

Documents for UNIX

VOLUME 1

T. A. Dolotta
S. B. Olsson
A. G. Petruccelli
Editors

January 1981

*Not for use or disclosure outside the
Bell System except under written agreement*

Laboratory 4517
Bell Telephone Laboratories, Incorporated
Murray Hill, NJ 07974

Copyright © 1981 Bell Telephone Laboratories, Inc.

UNIX is a trademark of Bell Telephone Laboratories, Inc.

*These documents were set on an AUTOLOGIC,
Inc. APS-5 phototypesetter driven by the TROFF
formatter operating under the UNIX system.*

ANNOTATED TABLE OF CONTENTS

NOTES: All the documents included here are supplements to the *UNIX User's Manual* (see G.1 below); the reader's attention is also drawn to documents G.2, G.3, and G.4.

Each document listed in Sections A through F below applies to UNIX Release 4.0, unless otherwise indicated after its title.

The number of pages in each document is given after the name(s) of its author(s).

VOLUME 1

A. OVERVIEWS

1. Overview and Synopsis

1. *UNIX—Overview and Synopsis of Facilities*

T. A. Dolotta, R. C. Haight, and A. G. Petrucci (p. 17)

A concise outline of the features and facilities of UNIX.

2. The UNIX Time-Sharing System

1. *The UNIX Time-Sharing System*

D. M. Ritchie and K. Thompson (p. 16)

The original, prize-winning UNIX paper, reprinted from G.5 below.

B. GETTING STARTED

1. Road Map

1. *UNIX Documentation Road Map*

G. A. Snyder and J. R. Mashey (p. 8)

A structured list of UNIX documents and information sources.

~~✗~~ A local section should be added to this document at each installation.

2. Editors

1. *A Tutorial Introduction to the UNIX Text Editor*

B. W. Kernighan (p. 11)

An easy way to get started with the text editor.

2. *Advanced Editing on UNIX*

B. W. Kernighan (p. 16)

A guide to the more advanced features of the text editor.

3. *SED—A Non-Interactive Text Editor*

L. E. McMahon (p. 10)

A variant of the text editor for stream editing.

3. UNIX for Beginners

1. *UNIX for Beginners (Second Edition)*

B. W. Kernighan (p. 13)

An introduction to some of the basic uses of UNIX.

4. Shell

1. *UNIX Shell Tutorial*

G. A. Snyder and J. R. Mashey (p. 36+ii)

An introduction to the various uses and facilities of the UNIX command language interpreter, with many examples.

2. *An Introduction to the UNIX Shell*

S. R. Bourne (p. 24)

Description of the UNIX command language interpreter.

C. DOCUMENT PREPARATION

1. NROFF/TROFF
 1. *A TROFF Tutorial*
B. W. Kernighan (p. 14)
A beginner's guide to phototypesetting with TROFF.
 2. *NROFF/TROFF User's Manual*
J. F. Ossanna (p. 37)
Reference manual for the UNIX text formatters.
2. Macros for NROFF/TROFF
 1. *MM—Memorandum Macros*
D. W. Smith and J. R. Mashey (p. 69+iv)
Reference manual for MM, the standard BTL text-formatting macros.
 2. *Typing Documents with MM*
D. W. Smith and E. M. Piskorik (p. 16)
A fold-out card that summarizes the MM macros; furnished separately.
 3. *A Macro Package for View Graphs and Slides*
T. A. Dolotta and D. W. Smith (p. 23)
A guide to making visual aids with TROFF.
3. TBL and EQN
 1. *TBL—A Program to Format Tables*
M. E. Lesk (p. 18)
An NROFF/TROFF preprocessor that permits easy formatting of tabular matter.
 2. *Typesetting Mathematics—User's Guide (Second Edition)*
B. W. Kernighan and L. L. Cherry (p. 11)
Manual for the EQN and NEQN preprocessors for TROFF and NROFF, respectively; these preprocessors allow one to specify, in an easy-to-learn language, how to typeset complex mathematical expressions.
 3. *A System for Typesetting Mathematics*
B. W. Kernighan and L. L. Cherry (p. 8)
A revision of the original EQN paper (*CACM* 18, March 1975), describing the principles behind the design of its input language and internal structure.

D. PROGRAMMING

1. C and LINT
 1. *The C Programming Language—Reference Manual*
D. M. Ritchie (p. 31)
Official statement of the syntax and semantics of C; supplemented by G.9 below.
 2. *A Guide to the C Library for UNIX Users*
C. D. Perez (p. 20)
An explanation of how to use the C library.
 3. *LINT, a C Program Checker*
S. C. Johnson (p. 11)
A program that checks C code for syntax errors, type violations, portability problems, and a variety of potential errors.

2. FORTRAN, RATFOR, and EFL
 1. *A Portable FORTRAN 77 Compiler*
S. I. Feldman and P. J. Weinberger (p. 19)
The FORTRAN 77 language and its interfaces with the operating system.
 2. *RATFOR—A Preprocessor for a Rational FORTRAN*
B. W. Kernighan (p. 12)
A preprocessor that endows FORTRAN with C-like control structures and input format.
 3. *The Programming Language EFL*
S. I. Feldman (p. 36)
A general-purpose computer language intended to encourage portable programming, while making use of the good features and facilities of FORTRAN.
3. UNIX Programming
 1. *UNIX Programming (Second Edition)*
B. W. Kernighan and D. M. Ritchie (p. 22)
A guide to writing programs that interface to the UNIX operating system, either directly or through the Standard I/O Library.
4. MAKE
 1. *MAKE—A Program for Maintaining Computer Programs*
S. I. Feldman (p. 9)
A tool for automating the recompilation of large programs.
 2. *An Augmented Version of MAKE*
E. G. Bradford (p. 16)
A discussion of how to use MAKE to its fullest advantage.
5. Debuggers
 1. *SDB—A Symbolic Debugger*
H. P. Katseff (p. 9)
A debugger that allows one to examine the “core image” of an aborted program.
 2. *A Tutorial Introduction to ADB*
J. F. Maranzano and S. R. Bourne (p. 27)
A guide to debugging crashed systems and programs; ADB is used mostly by system programmers.

VOLUME 2

E. SUPPORTING TOOLS AND LANGUAGES

1. LEX and YACC
 1. *LEX—A Lexical Analyzer Generator*
M. E. Lesk and E. Schmidt (p. 19)
A program that generates recognizers of sets of regular expressions; each regular expression can be followed by arbitrary C code that is executed when the regular expression is found.
 2. *YACC—Yet Another Compiler-Compiler*
S. C. Johnson (p. 33)
A converter from a BNF specification of a language and semantic actions written in C into a compiler for that language.
2. M4 Macro Processor
 1. *The M4 Macro Processor*
B. W. Kernighan and D. M. Ritchie (p. 6)
A macro processor, also useful as a front end for languages such as C and RATFOR.
3. AWK
 1. *AWK—A Pattern Scanning and Processing Language (Second Edition)*
A. V. Aho, B. W. Kernighan, and P. J. Weinberger (p. 8)
A language that makes it easy to specify many data selection and transformation operations.
4. SCCS
 1. *Source Code Control System User's Guide*
L. E. Bonanni and C. A. Salemi (p. 27)
A package for controlling access and changes to (possibly multiple versions of) source programs and text files.
 2. *Function and Use of an SCCS Interface Program*
L. E. Bonanni and A. Guyton (p. 3)
A discussion of how to control concurrent updates to SCCS files.
5. Calculators
 1. *BC—An Arbitrary Precision Desk-Calculator Language*
L. L. Cherry and R. Morris (p. 14)
A front end for DC (see below) that provides infix notation, flow control, and built-in functions.
 2. *DC—An Interactive Desk Calculator*
R. Morris and L. L. Cherry (p. 8)
An interactive desk calculator program that implements arbitrary-precision integer arithmetic.
6. Graphics
 1. *UNIX Graphics Overview*
A. R. Feuer (p. 7)
An introduction to the UNIX graphics facility.
 2. *A Tutorial Introduction to the Graphics Editor*
A. R. Feuer (p. 17)
A guide to making graphs, drawings, and pictures on Tektronix series 4010 terminals.

3. *STAT—A Tool for Analyzing Data*

A. R. Feuer and A. Guyton (p. 20)

A collection of programs that can be interconnected via the shell to analyze statistical data and display the results in graphical form.

4. *Administrative Information for the UNIX Graphics Package*

R. L. Chen, D. E. Pinkston, and A. Guyton (p. 6)

A reference guide for administrators of UNIX graphics facilities.

7. RJE and Networking**1. *UNIX Remote Job Entry User's Guide***

A. L. Sabsevitz and K. A. Kelleman (p. 7)

A guide to submitting jobs to an IBM system via the UNIX Remote Job Entry (RJE) facility.

2. *UNIX Remote Job Entry Administrator's Guide*

M. J. Fitton (p. 20)

A guide to setting up RJE on both UNIX and IBM systems, and to trouble-shooting when things go wrong.

3. *Release 1.0 of the UNIX Virtual Protocol Machine* (UNIX 3.0)

P. F. Long and C. Mee, III (p. 7)

A description of the first version of VPM; good background reading.

4. *Release 2.0 of the UNIX Virtual Protocol Machine* (UNIX 3.0)

P. F. Long and C. Mee, III (p. 20)

A newer release of VPM; supports bit-oriented, full-duplex protocols.

8. UUCP**1. *A Dial-up Network of UNIX Systems***

D. A. Nowitz and M. E. Lesk (p. 10)

Description of the design of a dial-up UNIX network called UUCP and used for transmission and distribution of programs and text files.

2. *UUCP Implementation Description*

D. A. Nowitz (p. 15)

A detailed description of UUCP for use by administrators of UNIX systems.

9. Printer Spooler**1. *The Implementation of the LP Spooling System***

J. R. Kliegman (p. 13)

Explanation of how the LP spooler works and how it can be used as a general-purpose spooler, as well as a line-printer spooler.

2. *LP Administrator's Guide*

J. R. Kliegman (p. 12)

A guide for those who oversee the operation of LP spoolers.

F. ADMINISTRATION, MAINTENANCE, AND IMPLEMENTATION**1. Operations and FSCK****1. *UNIX Operations Manual***

A. G. Petrucci (p. 24+ii)

Duties of a UNIX operator.

2. *FSCK—The UNIX File System Check Program*

T. J. Kowalski (p. 20)

A guide to checking and fixing UNIX file systems.

2. Accounting and System Activity
 1. *The UNIX Accounting System*
H. S. McCreary and A. G. Petrucelli (p. 19)
A guide to the use and management of the UNIX accounting system.
 2. *The UNIX System Activity Package*
T. W. Pao (p. 8)
A package that reports on processor utilization, terminal activity, disk and tape I/O, swapping, system calls, etc.
3. Stand-Alone I/O
 1. *A Stand-Alone Input/Output Library*
S. R. Eisen (p. 11)
A guide to the stand-alone library and the stand-alone shell (SASH).
4. ETP
 1. *The UNIX Equipment Test Package: Operational Procedures* (UNIX 3.0)
A. L. Chellis and T. J. Kowalski (p. 24)
The Equipment Test Package, a collection of UNIX hardware exercisers.
5. UNIX Internals
 1. *UNIX Implementation*
K. Thompson (p. 10)
An explanation of how UNIX works; reprinted from G.5 below.
 2. *The UNIX I/O System*
D. M. Ritchie (p. 7)
Guide for writers of UNIX device drivers.
 3. *UNIX on the PDP-11/23 and 11/34 Computers* (UNIX 3.0)
T. J. Kowalski (p. 7)
Description of what had to be done to UNIX to make it run on the PDP-11/23 and the PDP-11/34.
 4. *UNIX Assembler Reference Manual*
D. M. Ritchie (p. 12)
Describes the UNIX PDP-11 assembler; a tool of last resort.
6. C Internals
 1. *A Tour Through the Portable C Compiler*
S. C. Johnson (p. 25)
A description of how the portable C compiler works.
 2. *A Tour Through the UNIX C Compiler*
D. M. Ritchie (p. 15)
A description of how the PDP-11 C compiler works.
7. Security
 1. *On the Security of UNIX*
D. M. Ritchie (p. 3)
Hints on how to break UNIX and how to prevent it.
 2. *Password Security—A Case History*
R. Morris and K. Thompson (p. 6)
The story of how the bad guys used to be able to break the password algorithm and why they can't now, at least not so easily.

G. RECOMMENDED READING (not included)

1. ***UNIX User's Manual***—Release 3.0
T. A. Dolotta, S. B. Olsson, and A. G. Petrucci (eds.)
Bell Laboratories (June 1980).
The basic document for every UNIX user.
2. ***UNIX Reference Guide***
J. C. White (compiler) and P. V. Guidi (ed.)
Bell Laboratories (April 1981).
A pocket-size summary of UNIX commands, macro packages, etc.
3. ***Setting up UNIX***
R. C. Haight, M. J. Petrella, and L. A. Wehr
Bell Laboratories.
Procedures for installing UNIX; must reading for anyone who wants to configure and/or generate a UNIX system. (Because this document changes with each release of UNIX, it is not included here; it is distributed with each copy of the UNIX system itself.)
4. ***Administrative Advice for UNIX***
R. C. Haight
Bell Laboratories.
Hints for getting UNIX up, getting it going, and keeping it going, plus some information about hardware; must reading for UNIX system administrators. (This document is distributed just like G.3 above.)
5. ***The Bell System Technical Journal***
Vol. 57, No. 6, Part 2 (July-August 1978).
Special issue devoted to UNIX.
6. ***Using a Command Language as the Primary Programming Tool***
T. A. Dolotta and J. R. Mashey
In: Beech, D. (ed.), *Command Language Directions* (Proc. Second IFIP Working Conf. on Command Languages). Amsterdam: North Holland (1980), pp. 35-55.
A discussion of how to get the most out of the UNIX shell.
7. ***The UNIX Programming Environment***
B. W. Kernighan and J. R. Mashey
COMPUTER, Vol. 14, No. 4, pp. 12-24 (April 1981); an earlier version of this paper was published in *Software—Practice & Experience*, Vol. 9, No. 1, pp. 1-15 (Jan. 1979).
A discussion of what's good about UNIX.
8. ***Software Tools***
B. W. Kernighan and P. J. Plauger
Reading, MA: Addison-Wesley (1976).
A textbook for building good software tools similar to those available in UNIX.
9. ***The C Programming Language***
B. W. Kernighan and D. M. Ritchie
Englewood Cliffs, NJ: Prentice-Hall (1978).
The basic book for every C programmer; contains a tutorial and many examples.
10. ***Experiences with the UNIX Time-sharing System***
J. Lions
Software—Practice & Experience, Vol. 9, No. 9, pp. 701-709 (September 1979).
An enjoyable article that tells why they like UNIX in New South Wales.

11. The Evolution of the UNIX Time-sharing System

D. M. Ritchie

Proc. Symposium on Language Design and Programming Methodology, Sydney, Australia (September 1979).

Ten years later, one of the creators of UNIX looks back.

12. The Source Code Control System

M. J. Rochkind

IEEE Trans. Software Eng., Vol. SE-1, No. 4, pp. 364-370 (December 1975).

The motivation for, and the underlying design of, SCCS.

UNIX—Overview and Synopsis of Facilities

T. A. Dolotta
R. C. Haight
A. G. Petruccelli

Bell Laboratories
Murray Hill, New Jersey 07974

OVERVIEW

1. UNIX TIME-SHARING SYSTEM

The UNIX[†] Time-Sharing System is a general-purpose, multi-user, interactive operating system specifically engineered to make the designer's, programmer's, and documenter's computing environment simple, efficient, flexible, and productive. UNIX contains features such as:

- A hierarchical file system.
- A flexible, easy-to-use command language (can be "tailored" to meet specific user needs).
- Ability to execute sequential, asynchronous, and background processes.
- A powerful context editor.
- Very flexible document preparation and text processing systems.
- A high-level programming language conducive to structured programming (C).
- Other languages, including *FORTRAN 77*, *EFL*, and variants of *SNOBOL* and *BASIC*.
- Symbolic debugging systems.
- A variety of system programming tools (i.e., lexical analyzers, compiler-compilers, etc.).
- Sophisticated "desk-calculator" packages.
- Inter-machine communication by both hard-wired and dial-up facilities.
- A system designed to help control changes to source code and files of text (SCCS).
- A graphical plotting package.

Currently, UNIX runs on the Western Electric Co. 3B-20; Digital Equipment Corporation's (DEC) PDP-11/23, /34, /45, /70, VAX-11/780, and VAX-11/750; and IBM System/370 and equivalent. The cost per user-hour of UNIX is significantly lower than that of most other interactive computer systems; UNIX typically runs unattended.

The UNIX file system consists of a highly-uniform set of directories and files arranged in a tree-like hierarchical structure. Some of its features are:

- Simple and consistent naming conventions; names can be absolute, or relative to any directory in the file system hierarchy.
- Mountable and de-mountable file systems and volumes.
- File linking across directories.
- Automatic file space allocation and de-allocation that is invisible to users.
- A complete set of flexible directory and file protection modes, allowing all combinations of *read*, *write*, and *execute* access, independently for the owner of each file or directory, for a group of users (e.g., all members of a project), and for all other users; protection modes can be set dynamically.
- Facilities for creating, accessing, moving, and processing files, directories, or sets of these in a simple, uniform, and natural way.
- Each physical I/O device, from interactive terminals to main memory, is treated like a file, allowing uniform file and device I/O.

[†] UNIX is a trademark of Bell Laboratories.

2. UNIX COMMAND LANGUAGE

Unlike other interactive command languages, the UNIX shell is a full programming language. The shell provides variables, conditional and iterative constructs, and a user environment that can be tailored to an individual's or group's needs. Any user can create new commands simply by writing shell scripts.

3. DOCUMENT PREPARATION AND TEXT PROCESSING

In a software development project of any appreciable size, the production of usable, accurate documentation may well consume more effort than the production of the software itself. Several years of experience with many projects that use UNIX have shown that document preparation should not be separated from software development, and that the combination of a flexible operating system, a powerful command language, and good text processing facilities permit quick and convenient production of many kinds of documentation that might be otherwise unobtainable, impractical, or very expensive.

In UNIX, one also obtains a very useful "word processing" system—an editing system, text formatting systems, a typesetting system, and spelling and typographical error-detection facilities. The document preparation and text processing facilities of UNIX include commands that automatically control pagination, style of paragraphs, line justification, hyphenation, multi-column pages, footnote placement, generation of marginal revision bars, generation of tables of contents, etc., for specialized documents such as program run books, or for general documents such as letters, memoranda, legal briefs, etc. There are also excellent facilities for formatting and typesetting complex tables and equations. This document was produced in its entirety by these facilities.

4. REMOTE JOB ENTRY

The RJE facility provides for the submission and retrieval of jobs from an IBM host system (e.g., a System/360 or System/370 computer using HASP, ASP, JES2, or JES3). To the host system, RJE appears to be a System/360 work station.

At the request of a UNIX user, RJE gathers the job control statements and source code from files created and stored on UNIX, sends them to the host IBM system and, subsequently, retrieves from the host the resulting output, either placing it in a convenient UNIX file for later perusal, or using that output as the standard input to a specified shell procedure. Automatic notification of the output's arrival is also available.

5. SOURCE CODE CONTROL SYSTEM

The UNIX Source Code Control System (SCCS) is an integrated set of commands designed to help software development projects control changes to source code and to files of text (e.g., manuals). It provides facilities for storing, updating, and retrieving, by version number or date, all versions of source code modules or of documents, and for recording who made each software change, when it was made, and why. SCCS is designed to solve most of the source code and documentation control problems that software development projects encounter when customer support, system testing, and development are all proceeding simultaneously. Some of the main characteristics of SCCS are:

- The exact source code or text, as it existed at any point of development or maintenance, can be recreated at any later time.
- All releases and versions of a source code module or document are stored together, so the common code or text is stored only once.
- Releases in production or system test status can be protected from unauthorized changes.
- Enough identifying information can be automatically inserted into source code modules to enable one to identify the exact version and release of any such module, given only the corresponding load module or its memory dump.

SOFTWARE, FACILITIES, AND DOCUMENTATION

Often-used UNIX commands are listed below. Every command, including all its options, is issued as a single line, unless specifically described below as being “interactive.” Interactive programs can be made to run from a prepared “script” simply by redirecting their input. All commands are fully described in the *UNIX User's Manual* (see Section 6.1 below). Commands for which additional manuals and tutorials are provided are marked with [m] and [t], respectively. All indicated manuals and tutorials are listed in Section 6.2 below.

File processing commands that go from standard input to standard output are called “filters” and are marked with [f]. The “pipe” facility of the shell may be used to connect filters directly to the input or output of other filters and programs thus creating a “pipeline.”

Almost all of UNIX is written in C. UNIX is totally self-supporting: it contains all the software that is needed to generate it, maintain it, and modify it. Source code is included except as noted below.

1. BASIC SOFTWARE

Included are the operating system with utilities, an assembler, and a compiler for the programming language C—enough software to regenerate, maintain, and modify UNIX itself, and to write and run new applications. Due to hardware constraints, not all the commands listed below will work on all the supported hardware configurations.

1.1. Operating System

- UNIX [m] This is the basic resident code, also known as the kernel, on which everything else depends. It executes the system calls, maintains the file system, and manages the system's resources; it contains device drivers, I/O buffers, and other system information. A general description of UNIX design philosophy and system facilities appeared in the *Communications of the ACM*. A more extensive survey is in the *Bell System Technical Journal* for July-August 1978. Further capabilities include:
 - Automatically-supported reentrant code.
 - Separation of instruction and data spaces (machine dependent).
 - Timer-interrupt sampling and interprocess monitoring for debugging and measurement.
- Devices [m] All I/O is logically synchronous. Normally, automatic buffering by the system makes the physical record structure invisible and exploits the hardware's ability to do overlapped I/O. Unbuffered physical record I/O is available for unusual applications. Software drivers are provided for many devices; others can be easily written.

1.2. User Access Control

- LOGIN Signs on a new user:
 - Adapts to characteristics of terminal.
 - Verifies password and establishes user's individual and group (project) identity.
 - Establishes working directory.
 - Publishes message of the day.
 - Announces presence of mail.
 - Lists unseen news items.
 - Executes an optional user-specified profile.
 - Starts command interpreter (shell) or other user-specified program.
- PASSWD Changes a password:
 - User can change own password.
 - Passwords are kept encrypted for security.

- **SU** Assume the permissions and privileges of another user or root (super-user) provided that the proper password is supplied.
- **NEWGRP** Changes working group (project ID). This provides access with protection for groups of related users.
- **STTY** Sets up options for optimal control of a terminal. In so far as they are deducible from the input, these options are set automatically by LOGIN:
 - Speed.
 - Parity.
 - Mapping of upper-case characters to lower case.
 - Carriage-return plus line-feed versus new-line.
 - Interpretation of tab characters.
 - Delays for tab, new-line, and carriage-return characters.
 - Raw versus edited input.
- **TABS** Sets terminal's tab stops. Knows several "standard" formats.

1.3. Manipulation of Files and Directories

- **ED [m,t]** Interactive line-oriented context editor. Random access to all lines of a file. It can:
 - Find lines by number or pattern (regular expressions). Patterns can include: specified characters, "don't care" characters, choices among characters, (specified numbers of) repetitions of these constructs, beginning of line, end of line.
 - Add, delete, change, copy, or move lines.
 - Permute contents of a line.
 - Replace one or more instances of a pattern within a line.
 - Combine or split lines.
 - Combine or split files.
 - Do any of above operations on every line (in a given range) that matches a pattern.
 - Escape to the shell (UNIX command language) during editing.
- **SED [f,m]** A stream (one-pass) editor with facilities similar to those of ED.
- **CAT [f]** Concatenates one or more files onto standard output. Mostly used for unadorned printing, for inserting data into a "pipe," and for buffering output that comes in dribs and drabs.
- **PR [f]** Prints files with title, date, and page number on every page:
 - Multi-column output.
 - Parallel column merge of several files.
- **SPLIT** Splits a large file into more manageable pieces.
- **CSPLIT** Like SPLIT, with the splitting controlled by context.
- **SUM** Computes the check sum of a file.
- **DD [f]** Physical file format translator, for exchanging data with non-UNIX systems, especially OS/360, VSI, MVS, etc.
- **CP** Copies one file to another or many files to a directory. Works on any file regardless of its contents.
- **LN** Links another name (alias) to an existing file.
- **MV** Moves one or more files. Usually used for renaming files or directories.

- **RM** Removes one or more files. If any names are linked to the file, only the name being removed goes away.
- **CHMOD** Changes access permissions on a file(s). Executable by the owner of the file(s), or by the super-user.
- **CHOWN** Changes owner of a file(s).
- **MKDIR** Makes one or more new directories.
- **RMDIR** Removes one or more (empty) directories.
- **CD** Changes working (i.e., current) directory.
- **FIND** Searches the directory hierarchy for, and performs specified commands on, every file that meets given criteria:
 - File name matches a given pattern.
 - Modified date in given range.
 - Date of last use in given range.
 - Given permissions.
 - Given owner.
 - Given special file characteristics.
 - Any logical combination of the above.
 - Any directory can be the starting "node."
- **CPIO [f]** Copies a sub-tree of the file system (directories, links, and all) to another place in the file system. Can also copy a sub-tree onto a tape, and later recreate it from tape. Often used with the **FIND** command.
- **SCCS [m]** SCCS (Source Code Control System) is a collections of UNIX commands (some interactive) for controlling changes to files of text (typically the source code of programs or the text of documents). It provides facilities for:
 - Storing, updating, and retrieving any version of any source or text file.
 - Controlling updating privileges.
 - Identifying both source and object (or load) modules by version number.
 - Recording who made each change, when it was made, and why.

1.4. Execution of Programs

- **SH [f,m,t]** The shell, or command language interpreter, understands a set of constructs that constitute a full programming language; it allows a user or a command procedure to:
 - Supply arguments to and run any executable program.
 - Redirect standard input, standard output, and standard error files.
 - Pipes: simultaneous execution with output of one process connected to the input of another.
 - Compose compound commands using:
 - *if ... then ... else,*
 - *case switches,*
 - *while loops,*
 - *for loops over lists,*
 - *break, continue, and exit,*
 - *parentheses for grouping.*
 - Initiate background processes.
 - Perform shell procedures (i.e., command scripts with substitutable arguments).
 - Construct argument lists from all file names matching specified patterns.
 - Take user-specified action on traps and interrupts.

- Specify a search path for finding commands.
 - Upon login, automatically create a user-specifiable environment.
 - Optionally announce presence of mail as it arrives.
 - Provide variables and parameters with default settings.
- TEST Tests argument values in shell conditional constructs:
- String comparison.
 - File nature and accessibility.
 - Boolean combinations of the above.
- EXPR String computations for calculating command arguments:
- Integer arithmetic
 - Pattern matching
 - Like TEST above, EXPR can be used for conditional side-effect.
- ECHO Prints its arguments on the standard output. Useful for diagnostics or prompts in shell procedures, or for inserting data into a “pipe.”
- RSH Restricted shell; restricts a user to a subset of UNIX commands. The system administrator may construct different levels of restriction.
- SLEEP Suspends execution for a specified time.
- WAIT Waits for termination of a specific or all processes that are running in the background.
- NOHUP Runs a command immune to interruption from “hanging up” the terminal.
- NICE Runs a command at low (or high) priority.
- KILL Terminates named process(es).
- CRON Performs actions at specified times:
- Actions are arbitrary shell procedures or executable programs.
 - Times are conjunctions of month, day of month, day of week, hour, and minute. Ranges are specifiable for each.
- TEE [f] Passes data between processes (like a “pipe”), but also diverts copies into one or more files.
- HELP Explains error messages from certain other programs.

1.5. Status Inquiries

- LS Lists the names of one, several, or all files in one or more directories:
- Alphabetic or chronological sorting, up or down.
 - Optional information: size, owner, group, date last modified, date last accessed, permissions.
- FILE Tries to determine what kind of information is in a file by consulting the file system index and by reading the file itself.
- DATE Print current date and time. Has considerable knowledge of calendrical and horologic peculiarities; can be used to set UNIX’s idea of date and time. (As yet, cannot cope with Daylight Saving Time in the Southern Hemisphere.)
- DF Reports amount of free space in file system.
- DU Prints a summary of total space occupied by all files in a hierarchy.
- TTY Prints the “name” of your terminal (i.e., the name of the port to which your terminal is connected).

- WHO Tells who is logged onto the system:
 - Lists logged-in users, their ports, and time they logged in.
 - Optional history of all logins and logouts.
 - Tells you who you are logged in as.
- PS Reports on active processes:
 - Lists your own or everybody's processes.
 - Tells what commands are being executed at the moment.
 - Optional status information: state and scheduling information, priority, attached terminal, what the process is waiting for, its size, etc.
- ACCTCOM [f] Reports a chronological history of all process that have terminated. Information includes:
 - User and system times and sizes.
 - Start and end real times.
 - Owner and terminal line associated with process.
 - System exit status.
- PWD Prints name of your working (i.e., current) directory.
- RJESTAT Reports on the status of the Remote Job Entry (RJE) interface(s) to an IBM host.
- WHAT Prints informational lines found in files usually inserted by SCCS.

1.6. Inter-User Communication

- MAIL Mails a message to one or more users. Also used to read and dispose of incoming mail. The presence of mail is announced by LOGIN.
- NEWS Prints out current general information and announcement files.
- CALENDAR An automatic reminder service.
- WRITE Establishes direct, interactive terminal-to-terminal communication with another user.
- WALL Broadcasts a message to all users who are logged in.
- MSG Inhibits or permits receipt of messages from WRITE and WALL.

1.7. Inter-Machine Communication

- UUCP [m] Sends files back and forth between UNIX machines.
- SEND [m] Collects files together to be sent as a "job" to an IBM host.
- FSEND Sends files to the HONEYWELL 6000.
- FGET Retrieves files from the HONEYWELL 6000.
- CU Dials a phone number and attempts to make an interactive connection with another machine.
- CT Dials the phone number of a modem that is attached to a terminal, and spawns a LOGIN process to that terminal.
- VPM [m] A software package for implementing communications protocols. It consists of a protocol script interpreter that runs in a front-end microprocessor, allowing a variety of different protocols to be implemented with the same hardware.
- BX.25 A superset of the international X.25 communications protocol; it is implemented using VPM.

1.8. Program Development Package

A kit of fundamental programming tools. Some of these are used as integral parts of the higher-level languages described in Section 2 below.

- AR Maintains library archives, especially useful with LD. Combines several files into one for housekeeping efficiency:
 - Creates new archive.
 - Updates archive by date.
 - Replaces or deletes files.
 - Prints table of contents.
 - Retrieves from archive.

- Libraries [m] Basic run-time libraries. They are used freely by all system software:
 - Number conversions.
 - Time conversions.
 - Mathematical functions: *sin, cos, log, exp, atan, sqrt, gamma*.
 - Buffered character-by-character I/O.
 - Random number generator.
 - An elaborate library for formatted I/O.
 - Password encryption.

- ADB [t] Interactive debugger:
 - Postmortem dumping.
 - Examination of arbitrary files, with no limit on size.
 - Interactive breakpoint debugging; the debugger is a separate process.
 - Symbolic reference to local and global variables.
 - Stack trace for C programs.
 - Output formats:
 - 1-, 2-, or 4-byte integers in octal, decimal, or hex
 - single and double floating point
 - character and string
 - disassembled machine instructions
 - Patching.
 - Searching for integer, character, or floating patterns.
 - Handles separated instruction and data space.

- OD [f] Dumps any file:
 - Output options include: octal or decimal by words, octal by bytes, ASCII, operation codes, hexadecimal, or any combination thereof.
 - Range of dumping is controllable.

- SDB [m] Symbolic debugger for C and F77 programs.
- LD Linkage editor. Combines relocatable object files. Inserts required routines from specified libraries; resulting code:
 - Can be made sharable.
 - Can be made to have separate instruction and data spaces.

- NM Prints the *namelist* (symbol table) of an object program. Provides control over the style and order of names that are printed.

- SIZE Reports the main memory requirements of one or more object files.

- STRIP Removes the relocation and symbol table information from an object file to save file space.

- PROF Constructs a profile of time spent in each routine from data gathered by time-sampling the execution of a program; gives subroutine call frequencies and average times for C programs.

- MAKE [m] Controls creation of large programs. Uses a control file specifying source file dependencies to make new version; uses time last changed to deduce minimum amount of work necessary. Knows about SCCS, CC, YACC, LEX, etc.

1.9. Utilities

- CXREF Makes cross-reference listings of a set of C source files. The listing contains all symbols in each file separately or, optionally, in combination. An asterisk appears before a symbol's declaration.
- SORT [f] Merges and/or sorts ASCII files line-by-line:
 - In ascending or descending order.
 - Lexicographically or on numeric key.
 - On multiple keys located by delimiters or by position.
 - Can fold upper-case characters together with lower-case into dictionary order.
- UNIQ [f] Deletes successive duplicate lines in a file:
 - Prints lines that were originally unique, duplicated, or both.
 - Can give redundancy count for each line.
- TR [f] Does character translation according to an arbitrary code:
 - Can "squeeze out" repetitions of selected characters.
 - Can delete selected characters.
- DIFF [f] Reports line changes, additions, and deletions necessary to bring two files into agreement; can produce an editor script to convert one file into another.
- COMM [f] Identifies common lines in two sorted files. Output in up to 3 columns shows lines present in first file only, present in second file only, and/or present in both.
- CMP Compares two files and reports disagreeing bytes.
- GREP [f] Prints all lines in one or more files that match a pattern of the kind used by ED (the editor):
 - Can print all lines that fail to match.
 - Can print count of "hits."
- WC [f] Counts lines and "words" (strings separated by blanks or tab characters) in a file.
- TIME Runs a command and reports timing information about it.

2. PROGRAMMING LANGUAGES

2.1. The Programming Language C

- CC [m,t] Compiles and/or link-edits programs in the C language. The UNIX operating system, almost all of its subsystems, and C itself are written in C:
 - General-purpose language designed for structured programming.
 - Data types:
 - Character.
 - Short.
 - Integer.
 - Long integer.
 - Floating-point.
 - Double.
 - Pointers to all types.

- Functions returning all types.
- Arrays of any type.
- Structures containing various types.
- Provides machine-independent control of all machine facilities, including to-memory operations and pointer arithmetic.
- Macro-preprocessor for parameterized code and for the inclusion of other files.
- All procedures recursive, with parameters passed by value.
- Run-time library gives access to all system facilities.

■ PCC [m]

Portable version of CC for a variety of computers.

■ CB [f]

C beautifier: gives a C program that well-groomed, structured, indented look.

2.2. FORTRAN

■ F77 [m]

A full compiler for ANSI Standard *FORTRAN 77*:

- Compatible with C and supporting tools at object level.
- Optional source compatibility with *FORTRAN 66*.
- Free format source.
- Optional subscript-range checking, detection of uninitialized variables.
- All widths of arithmetic: 2- and 4-byte integer; 4- and 8-byte real; 8- and 16-byte complex.

■ RATFOR [m]

Ratfor adds rational control structure à la C to *FORTRAN*:

- Compound statements.
- *If-else, do, for, while, repeat-until, break, next* statements.
- Symbolic constants.
- File insertion.
- Free format source
- Translation of relationals like $>$, $>=$, etc.
- Produces genuine *FORTRAN* to carry away.
- May be used with F77.

■ EFL [m]

Compiles a program written in the *EFL* Language into clean *FORTRAN* on the standard output. It provides the C-like control constructs of RATFOR.

2.3. Other Algorithmic Languages

■ AWK [m]

Pattern scanning and processing language. Searches input for patterns, and performs actions on each line of input that satisfies the pattern:

- Patterns include regular expressions, arithmetic and lexicographic conditions, boolean combinations and ranges of these.
- Data treated as string or numeric as appropriate.
- Can break input into fields; fields are variables.
- Variables and arrays (with non-numeric subscripts).
- Full set of arithmetic operators and control flow.
- Multiple output streams to files and pipes.
- Output can be formatted as desired.
- Multi-line capabilities.

■ BS

An interactive interpreter, containing features of both *BASIC* and *SNOBOL4*:

- Statements include:
 - *for/while ... next*
 - *goto*
 - *if ... else ... fi*
 - *trace*
 - *symbolic dump*

- All numeric calculations in double precision.
 - Recursive function defining and calling.
 - Built-in functions include *log*, *exp*, *sin*, *cos*, *atan*, *ceil*, *floor*, *sqrt*, *abs*, *rand*.
 - String operations include regular expression pattern matching.
 - Very general I/O (including pipes to commands) is provided.
- DC [m] Interactive programmable desk calculator. Has named storage locations, as well as conventional stack for holding integers and programs:
- Arbitrary-precision decimal arithmetic.
 - Appropriate treatment of decimal fractions.
 - Arbitrary input and output radices, in particular binary, octal, decimal, and hexadecimal.
 - Postfix (“Reverse Polish”) operators:
 + - * /
 remainder, power, square root
 load, store, duplicate, clear
 print, enter program text, execute
- BC [m] A C-like interactive interface to the desk calculator DC:
- All the capabilities of DC with a high-level syntax.
 - Arrays and recursive functions.
 - Immediate evaluation of expressions and evaluation of functions upon call.
 - Arbitrary-precision elementary functions: *exp*, *sin*, *cos*, *atan*.
 - Goto-less programming.
- SNO An interpreter very similar to *SNOBOL 3*; its limitations are:
- Function definitions are static.
 - Pattern matches are always anchored.
 - No built-in functions.

2.4. Macro-Processors and Compiler-Compilers

- M4 [f,m] A general-purpose macro-processor:
- Stream-oriented, recognizes macros anywhere in text.
 - Integer arithmetic.
 - String and substring capabilities.
 - Condition testing, file manipulation, arguments.
- YACC [m] An LALR(1)-based compiler-writing system. During execution of resulting parsers, arbitrary C functions can be called to do code generation or take semantic actions:
- BNF syntax specifications.
 - Precedence relations.
 - Accepts formally ambiguous grammars with non-BNF resolution rules.
- LEX [m] LEX helps write programs whose control flow is directed by instances of regular expressions in the input stream. It is well suited for editor-script type transformations and for segmenting input in preparation for a parsing routine.

3. TEXT PROCESSING

3.1. Formatters

High-level formatting macros have been developed to ease the preparation of documents with NROFF and TROFF, as well as to exploit their more complex formatting capabilities.

- NROFF [f,m,t] Advanced formatter for terminals. Capable of many elaborate feats:
 - Justification of either or both margins.
 - Automatic hyphenation.
 - Generalized page headers and footers, automatic page numbering, with even-odd page differentiation capability, etc.
 - Hanging indents and one-line indents.
 - Absolute and relative parameter settings.
 - Optional legal-style numbering of output lines.
 - Nested or chained input files.
 - Complete page format control, keyed to dynamically-planted "traps" at specified lines.
 - Several separately-definable formatting environments (e.g., one for regular text, one for footnotes, and one for "floating" tables and displays).
 - Macros with substitutable arguments.
 - Conditional execution of macros.
 - Conditional insertion or deletion of text.
 - String variables that can be invoked in mid-line.
 - Computation and printing of numerical quantities.
 - String-width computations for unusually-difficult layout problems.
 - Positions and distances expressible in inches, centimeters, ems, ens, line spaces, points, picas, machine units, and arithmetic combinations thereof.
 - Dynamic (relative or absolute) positioning.
 - Horizontal and vertical line drawing.
 - Multi-column output on terminals capable of reverse line-feed, or through the postprocessor COL.

- TROFF [f,m,t] This formatter generates output on a phototypesetter. It provides facilities that are upward-compatible with NROFF, but with the following additions:
 - Vocabulary of several 102-character fonts (any 4 simultaneously) in 15 different point sizes.
 - Character-width and string-width computations for unusually difficult layout problems.
 - Overstrikes and built-up brackets.
 - Dynamic (relative or absolute) point size selection, globally or at the character level.
 - Terminal output for rough sampling of the product.

☞ *This entire document was typeset by TROFF, assisted by MM, TBL, and EQN.* ☞

- EQN [f,m] A mathematical preprocessor for TROFF. Translates in-line or displayed formulae from a very easy-to-type form into detailed typesetting instructions. For example:

$\sigma^2 = \frac{1}{N} \sum_{j=1}^N (x_j - \bar{x})^2$

produces:

$$\sigma^2 = \frac{1}{N} \sum_{j=1}^N (x_j - \bar{x})^2$$

- Automatic calculation of point size changes for subscripts, superscripts,
- Full vocabulary of Greek letters, such as γ , Π , Γ , α .
- Automatic calculation of the size of large brackets.

- Vertical “piling” of formulae for matrices, conditional alternatives, etc.
- Integrals, sums, etc., with arbitrarily complex limits.
- Diacriticals: dots, double dots, hats, bars, etc.

Formulae can appear within tables to be formatted by TBL (see below).

- NEQN [f,m] A mathematical preprocessor for NROFF with the same facilities as EQN, except for the limitations imposed by the graphic capabilities of the terminal being used. Prepares formulae for display on various Diablo-mechanism terminals, etc.
- MM [m] A standardized manuscript layout macro package for use with NROFF/TROFF. Provides a flexible, user-oriented interface to these two formatters; designed to be:
 - Robust in face of user errors.
 - Adaptable to a wide range of output styles.
 - Can be extended by users familiar with the formatter.
 - Compatible with both NROFF and TROFF.

Some of its features are:

- Page numbers and draft dates.
- Cover sheets and title pages.
- Automatically-numbered or “lettered” headings.
- Automatically-numbered or “lettered” lists.
- Automatically-numbered figure and table captions.
- Automatically-numbered and positioned footnotes.
- Single- or double-column text.
- Paragraphing, displays, and indentation.
- Automatic table of contents.

- MV [m] A TROFF macro package that makes it easy to typeset professional-looking projection foils and slides.
- TBL [f,m] A preprocessor for NROFF that translates simple descriptions of table layouts and contents into detailed formatting instructions:
 - Computes appropriate column widths.
 - Handles left- and right-justified columns, centered columns, and decimal-point aligned columns.
 - Places column titles; spans these titles, as appropriate.

For example:

Composition of Foods			
Food	Percent by Weight		
	Protein	Fat	Carbo- hydrate
Apples	.4	.5	13.0
Halibut	18.4	5.2	...
Lima beans	7.5	.8	22.0
Milk	3.3	4.0	5.0
Mushrooms	3.5	.4	6.0
Rye bread	9.0	.6	52.7

- CW [f] A preprocessor for TROFF that prepares text to be displayed in a special “constant-width” typeface; this typeface is very useful for printing examples of computer output in, e.g., programming manuals.

3.2. Other Text Processing Tools

- SPELL [f] Finds spelling errors by looking up all uncommon words from a document in a large spelling list. Knows about prefixes and suffixes and can cope with such rotten spellings as "roted."
- PTX Generates a permuted index, like the one in the *UNIX User's Manual*.
- GRAPH [f] Given the coordinates of the points to be plotted, draws the corresponding graph; has many options for scaling, axes, grids, labeling, etc.
- TPLOT [f] Makes the output of GRAPH suitable for plotting on a Diablo-mechanism terminal.
- 300, 450 [f] Exploits the hardware facilities of GSI 300, DASI 450, and other Diablo-mechanism terminals:
 - Implements reverse line-feeds and forward and reverse fractional-line motions.
 - Allows any combination of 10- or 12-pitch printing with 6 or 8 lines/inch spacing.
 - Approximates Greek letters and other special characters by overstriking in plot mode.
- HP [f] Like 300, but for the Hewlett-Packard 2640 family of terminals.
- COL [f] Reformats files with reverse line-feeds so that they can be correctly printed on terminals that cannot reverse line-feed.
- Graphics [m,t] Graphics is the name of a collection of commands for manipulating and plotting statistical and graphical data on a Tektronix series 4010 terminal or a Hewlett-Packard 7221A Graphics Plotter. Its facilities include:
 - A sophisticated graphical editor.
 - Pie and bar chart generators.
 - Built-in mathematical functions such as powers, roots, logarithms, and slope and intercept generation.
 - Histograms.
 - Additive sequence, prime number, and random sequence generators.
 - Table of contents generators.

4. SYSTEM ADMINISTRATION

4.1. Normal Day-to-Day Administration and Maintenance

- MOUNT Attaches a device containing a file system to the tree of directories. Protects against nonsense arrangements.
- UMount Removes the file system contained on a device from the tree of directories. Protects against removing a busy device.
- MKFS Makes a new file system on a device.
- MKNOD Makes a file system entry for a special file. Special files are physical devices, virtual devices, physical memory, etc.
- VOLCOPY File system backup/recovery system for disk/disk or disk/tape. Protective labeling of disks and tapes is included.
- FSCK [m] Used to check the consistency of file systems and directories and make interactive repairs:
 - Print statistics: number of files, space used, free space.
 - Report duplicate use of space.
 - Retrieve lost space.

- Report inaccessible files.
 - Check consistency of directories.
 - Reorganize free disk space for maximum operating efficiency.
- SYNC Forces all outstanding I/O on the system to completion. Used to shut down the system gracefully.
 - CONFIG Tailors device-dependent system code to a specific hardware configuration. As distributed, UNIX can be brought up directly on any supported computer equipped with an acceptable tape drive and disk, sufficient amount of main memory, a console terminal, and a clock.
 - CRASH Prints out tables and structures in the operating system. May be used on a running system, but more useful for examining operating system core dumps after a "crash."

4.2. System Monitoring Facilities

- Accounting [m] The process accounting package covers connect time accounting, command usage, command frequency, disk utilization, and line usage. All of these are summarized by user and by command on a daily, monthly, and fiscal basis. The system lends itself to local needs and modification.
- Error Logging The UNIX operating system incorporates continuous hardware error detection and reporting.
- Equipment Test Package [m] The Equipment Test Package (ETP—available on a separate tape) is a useful addition to a hardware supplier's diagnostic software. It is essentially a UNIX-based hardware exerciser and verifier.
- System Activity Report [m] The System Activity Report (SAR) package is a body of programs for sampling the behavior of the operating system. The sampling consists of several time counters, I/O activity counters, context-switching counters, system-call counters, and file-access counters. Reports can be generated on a daily basis, or as desired.
- Profiler The Profiler is another group of commands for studying the activity of the operating system. It reports the percentage of time that the operating system spends on user tasks, on system functions, and in being idle.

4.3. Installation, Administration, and Operation

- Installation [m] The *Setting up UNIX* document contains the procedures and advice for the first-time installation and for the periodic upgrading of the operating system.
- Administration [m] The *Administrative Advice for UNIX* document describes various problems that can occasionally arise during normal operation, and suggests possible solutions. Included are tips on data-set options, specifications for phototypesetter fonts and chemicals, for system tuning, security, troubleshooting, as well as other useful information.
- Operation [m] The *UNIX Operations Manual* contains a description of console operations, step-by-step operator functions, and operating system error messages and their meanings.

5. DEMONSTRATION AND TRAINING PROGRAMS

Unless otherwise indicated, source code for the following interactive programs is *not* included:

- QUIZ Tests your knowledge of Shakespeare, presidents, capitals, etc. Source code included.
- BJ A blackjack dealer.
- MOO A fascinating number-guessing game, rather like Mastermind®.
- CAL Prints a calendar of specified month or year between A.D. 1 and 9999. Source code included.
- UNITS Converts quantities between different scales of measurement. Knows hundreds of units; for example, how many kilometers/second (or furlongs/fortnight) is a parsec/megayear? Source code included.
- TTT A traditional 3×3 tic-tac-toe program that learns. It never makes the same mistake twice, unless you make it forget what it has learned.
- BACK The game of Backgammon.
- HANGMAN Children's "guess the word" game.
- WUMP Thrilling hunt for the mighty wumpus in a dangerous cave.

6. USER DOCUMENTATION

6.1. UNIX User's Manual

- MAN [m] On-line and hard-copy versions are provided. The manual contains:
 - A system overview.
 - Commands.
 - System calls.
 - Subroutines in the C, math, standard I/O, and specialized libraries.
 - File formats for most files known to the system software.
 - etc.

6.2. Documents For UNIX

This two-volume collection contains documents that supplement the information in the *UNIX User's Manual*. It contains:

- OVERVIEWS
 - UNIX—Overview and Synopsis of Facilities
 - The UNIX Time-Sharing System
- GETTING STARTED
 - UNIX Documentation Road Map
 - A Tutorial Introduction to the UNIX Text Editor
 - Advanced Editing on UNIX
 - SED—A Non-Interactive Text Editor
 - UNIX for Beginners (Second Edition)
 - UNIX Shell Tutorial
 - An Introduction to the UNIX Shell
- DOCUMENT PREPARATION
 - A TROFF Tutorial
 - NROFF/TROFF User's Manual
 - MM—Memorandum Macros
 - Typing Documents with MM
 - A Macro Package for View Graphs and Slides
 - TBL—A Program to Format Tables

- Typesetting Mathematics—User's Guide (Second Edition)
- A System for Typesetting Mathematics

■ PROGRAMMING

- The C Programming Language—Reference Manual
- A Guide to the C Library for UNIX Users
- LINT, a C Program Checker
- A Portable FORTRAN 77 Compiler
- RATFOR—A Preprocessor for a Rational FORTRAN
- The Programming Language EFL
- UNIX Programming (Second Edition)
- MAKE—A Program for Maintaining Computer Programs
- An Augmented Version of MAKE
- SDB—A Symbolic Debugger
- A Tutorial Introduction to ADB

■ SUPPORTING TOOLS AND LANGUAGES

- LEX—A Lexical Analyzer Generator
- YACC—Yet Another Compiler-Compiler
- The M4 Macro Processor
- AWK—A Pattern Scanning and Processing Language (Second Edition)
- Source Code Control System User's Guide
- Function and Use of an SCCS Interface Program
- BC—An Arbitrary Precision Desk-Calculator Language
- DC—An Interactive Desk Calculator
- UNIX Graphics Overview
- A Tutorial Introduction to the Graphics Editor
- STAT—A Tool for Analyzing Data
- Administrative Information for the UNIX Graphics Package
- UNIX Remote Job Entry User's Guide
- UNIX Remote Job Entry Administrator's Guide
- Release 1.0 of the UNIX Virtual Protocol Machine
- Release 2.0 of the UNIX Virtual Protocol Machine
- A Dial-up Network of UNIX Systems
- UUCP Implementation Description
- The Implementation of the LP Spooling System
- LP Administrator's Guide

■ ADMINISTRATION, MAINTENANCE, AND IMPLEMENTATION

- UNIX Operations Manual
- FSCK—The UNIX File System Check Program
- The UNIX Accounting System
- The UNIX System Activity Package
- A Stand-Alone Input/Output Library
- The UNIX Equipment Test Package: Operational Procedures
- UNIX Implementation
- The UNIX I/O System
- UNIX on the PDP-11/23 and 11/34 Computers
- UNIX Assembler Reference Manual
- A Tour Through the Portable C Compiler
- A Tour Through the UNIX C Compiler
- On the Security of UNIX
- Password Security—A Case History

The UNIX Time-Sharing System*

D. M. Ritchie

K. Thompson

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

UNIX[†] is a general-purpose, multi-user, interactive operating system for the larger Digital Equipment Corporation PDP-11 and the Interdata 8/32 computers. It offers a number of features seldom found even in larger operating systems, including

- A hierarchical file system incorporating demountable volumes,
- Compatible file, device, and inter-process I/O,
- The ability to initiate asynchronous processes,
- System command language selectable on a per-user basis,
- Over 100 subsystems including a dozen languages,
- High degree of portability.

This paper discusses the nature and implementation of the file system and of the user command interface.

I. INTRODUCTION

There have been four versions of the UNIX time-sharing system. The earliest (circa 1969-70) ran on the Digital Equipment Corporation PDP-7 and -9 computers. The second version ran on the unprotected PDP-11/20 computer. The third incorporated multiprogramming and ran on the PDP-11/34, /40, /45, /60, and /70 computers; it is the one described in the previously published version of this paper, and is also the most widely used today. This paper describes only the fourth, current system that runs on the PDP-11/70 and the Interdata 8/32 computers. In fact, the differences among the various systems is rather small; most of the revisions made to the originally published version of this paper, aside from those concerned with style, had to do with details of the implementation of the file system.

Since PDP-11 UNIX became operational in February, 1971, over 600 installations have been put into service. Most of them are engaged in applications such as computer science education, the preparation and formatting of documents and other textual material, the collection and processing of trouble data from various switching machines within the Bell System, and recording and checking telephone service orders. Our own installation is used mainly for research in operating systems, languages, computer networks, and other topics in computer science, and also for document preparation.

* Copyright 1974, Association for Computing Machinery, Inc., reprinted by permission. This is a revised version of an article that appeared in *Communications of the ACM*, 17, No. 7 (July 1974), pp. 365-375. That article was a revised version of a paper presented at the Fourth ACM Symposium on Operating Systems Principles, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, October 15-17, 1973.

[†] UNIX is a trademark of Bell Laboratories.

Perhaps the most important achievement of UNIX is to demonstrate that a powerful operating system for interactive use need not be expensive either in equipment or in human effort: it can run on hardware costing as little as \$40,000, and less than two man-years were spent on the main system software. We hope, however, that users find that the most important characteristics of the system are its simplicity, elegance, and ease of use.

Besides the operating system proper, some major programs available under UNIX are

- C compiler
- Text editor based on QED¹
- Assembler, linking loader, symbolic debugger
- Phototypesetting and equation setting programs^{2,3}
- Dozens of languages including Fortran 77, Basic, Snobol, APL, Algol 68, M6, TMG, Pascal

There is a host of maintenance, utility, recreation and novelty programs, all written locally. The UNIX user community, which numbers in the thousands, has contributed many more programs and languages. It is worth noting that the system is totally self-supporting. All UNIX software is maintained on the system; likewise, this paper and all other documents in this issue were generated and formatted by the UNIX editor and text formatting programs.

II. HARDWARE AND SOFTWARE ENVIRONMENT

The PDP-11/70 on which the Research UNIX system is installed is a 16-bit word (8-bit byte) computer with 768K bytes of core memory; the system kernel occupies 90K bytes about equally divided between code and data tables. This system, however, includes a very large number of device drivers and enjoys a generous allotment of space for I/O buffers and system tables; a minimal system capable of running the software mentioned above can require as little as 96K bytes of core altogether. There are even larger installations; see the description of the PWB/UNIX systems,^{4,5} for example. There are also much smaller, though somewhat restricted, versions of the system.⁶

Our own PDP-11 has two 200-Mb moving-head disks for file system storage and swapping. There are 20 variable-speed communications interfaces attached to 300- and 1,200-baud data sets, and an additional 12 communication lines hard-wired to 9,600-baud terminals and satellite computers. There are also several 2,400- and 4,800-baud synchronous communication interfaces used for machine-to-machine file transfer. Finally, there is a variety of miscellaneous devices including nine-track magnetic tape, a line printer, a voice synthesizer, a phototypesetter, a digital switching network, and a chess machine.

The preponderance of UNIX software is written in the abovementioned C language.⁷ Early versions of the operating system were written in assembly language, but during the summer of 1973, it was rewritten in C. The size of the new system was about one-third greater than that of the old. Since the new system not only became much easier to understand and to modify but also included many functional improvements, including multiprogramming and the ability to share reentrant code among several user programs, we consider this increase in size quite acceptable.

III. THE FILE SYSTEM

The most important role of the system is to provide a file system. From the point of view of the user, there are three kinds of files: ordinary disk files, directories, and special files.

3.1 Ordinary files

A file contains whatever information the user places on it, for example, symbolic or binary (object) programs. No particular structuring is expected by the system. A file of text consists simply of a string of characters, with lines demarcated by the new-line character. Binary programs are sequences of words as they will appear in core memory when the program starts executing. A few user programs manipulate files with more structure; for example, the

assembler generates, and the loader expects, an object file in a particular format. However, the structure of files is controlled by the programs that use them, not by the system.

3.2 Directories

Directories provide the mapping between the names of files and the files themselves, and thus induce a structure on the file system as a whole. Each user has a directory of his own files; he may also create subdirectories to contain groups of files conveniently treated together. A directory behaves exactly like an ordinary file except that it cannot be written on by unprivileged programs, so that the system controls the contents of directories. However, anyone with appropriate permission may read a directory just like any other file.

The system maintains several directories for its own use. One of these is the **root** directory. All files in the system can be found by tracing a path through a chain of directories until the desired file is reached. The starting point for such searches is often the **root**. Other system directories contain all the programs provided for general use; that is, all the *commands*. As will be seen, however, it is by no means necessary that a program reside in one of these directories for it to be executed.

Files are named by sequences of 14 or fewer characters. When the name of a file is specified to the system, it may be in the form of a *path name*, which is a sequence of directory names separated by slashes, “/”, and ending in a file name. If the sequence begins with a slash, the search begins in the root directory. The name **/alpha/beta/gamma** causes the system to search the root for directory **alpha**, then to search **alpha** for **beta**, finally to find **gamma** in **beta**. **gamma** may be an ordinary file, a directory, or a special file. As a limiting case, the name “/” refers to the root itself.

A path name not starting with “/” causes the system to begin the search in the user’s current directory. Thus, the name **alpha/beta** specifies the file named **beta** in subdirectory **alpha** of the current directory. The simplest kind of name, for example, **alpha**, refers to a file that itself is found in the current directory. As another limiting case, the null file name refers to the current directory.

The same non-directory file may appear in several directories under possibly different names. This feature is called *linking*; a directory entry for a file is sometimes called a link. The UNIX system differs from other systems in which linking is permitted in that all links to a file have equal status. That is, a file does not exist within a particular directory; the directory entry for a file consists merely of its name and a pointer to the information actually describing the file. Thus a file exists independently of any directory entry, although in practice a file is made to disappear along with the last link to it.

Each directory always has at least two entries. The name “.” in each directory refers to the directory itself. Thus a program may read the current directory under the name “.” without knowing its complete path name. The name “..” by convention refers to the parent of the directory in which it appears, that is, to the directory in which it was created.

The directory structure is constrained to have the form of a rooted tree. Except for the special entries “.” and “..”, each directory must appear as an entry in exactly one other directory, which is its parent. The reason for this is to simplify the writing of programs that visit subtrees of the directory structure, and more important, to avoid the separation of portions of the hierarchy. If arbitrary links to directories were permitted, it would be quite difficult to detect when the last connection from the root to a directory was severed.

3.3 Special files

Special files constitute the most unusual feature of the UNIX file system. Each supported I/O device is associated with at least one such file. Special files are read and written just like ordinary disk files, but requests to read or write result in activation of the associated device. An entry for each special file resides in directory **/dev**, although a link may be made to one of these files just as it may to an ordinary file. Thus, for example, to write on a magnetic tape one

may write on the file `/dev/mt`. Special files exist for each communication line, each disk, each tape drive, and for physical main memory. Of course, the active disks and the memory special file are protected from indiscriminate access.

There is a threefold advantage in treating I/O devices this way: file and device I/O are as similar as possible; file and device names have the same syntax and meaning, so that a program expecting a file name as a parameter can be passed a device name; finally, special files are subject to the same protection mechanism as regular files.

3.4 Removable file systems

Although the root of the file system is always stored on the same device, it is not necessary that the entire file system hierarchy reside on this device. There is a `mount` system request with two arguments: the name of an existing ordinary file, and the name of a special file whose associated storage volume (e.g., a disk pack) should have the structure of an independent file system containing its own directory hierarchy. The effect of `mount` is to cause references to the heretofore ordinary file to refer instead to the root directory of the file system on the removable volume. In effect, `mount` replaces a leaf of the hierarchy tree (the ordinary file) by a whole new subtree (the hierarchy stored on the removable volume). After the `mount`, there is virtually no distinction between files on the removable volume and those in the permanent file system. In our installation, for example, the root directory resides on a small partition of one of our disk drives, while the other drive, which contains the user's files, is mounted by the system initialization sequence. A mountable file system is generated by writing on its corresponding special file. A utility program is available to create an empty file system, or one may simply copy an existing file system.

There is only one exception to the rule of identical treatment of files on different devices: no link may exist between one file system hierarchy and another. This restriction is enforced so as to avoid the elaborate bookkeeping that would otherwise be required to assure removal of the links whenever the removable volume is dismounted.

3.5 Protection

Although the access control scheme is quite simple, it has some unusual features. Each user of the system is assigned a unique user identification number. When a file is created, it is marked with the user ID of its owner. Also given for new files is a set of ten protection bits. Nine of these specify independently read, write, and execute permission for the owner of the file, for other members of his group, and for all remaining users.

If the tenth bit is on, the system will temporarily change the user identification (hereafter, user ID) of the current user to that of the creator of the file whenever the file is executed as a program. This change in user ID is effective only during the execution of the program that calls for it. The set-user-ID feature provides for privileged programs that may use files inaccessible to other users. For example, a program may keep an accounting file that should neither be read nor changed except by the program itself. If the set-user-ID bit is on for the program, it may access the file although this access might be forbidden to other programs invoked by the given program's user. Since the actual user ID of the invoker of any program is always available, set-user-ID programs may take any measures desired to satisfy themselves as to their invoker's credentials. This mechanism is used to allow users to execute the carefully written commands that call privileged system entries. For example, there is a system entry invocable only by the "super-user" (below) that creates an empty directory. As indicated above, directories are expected to have entries for `."` and `.."`. The command which creates a directory is owned by the super-user and has the set-user-ID bit set. After it checks its invoker's authorization to create the specified directory, it creates it and makes the entries for `."` and `.."`.

Because anyone may set the set-user-ID bit on one of his own files, this mechanism is generally available without administrative intervention. For example, this protection scheme easily solves the MOO accounting problem posed by "Aleph-null."⁸

The system recognizes one particular user ID (that of the “super-user”) as exempt from the usual constraints on file access; thus (for example), programs may be written to dump and reload the file system without unwanted interference from the protection system.

3.6 I/O calls

The system calls to do I/O are designed to eliminate the differences between the various devices and styles of access. There is no distinction between “random” and “sequential” I/O, nor is any logical record size imposed by the system. The size of an ordinary file is determined by the number of bytes written on it; no predetermination of the size of a file is necessary or possible.

To illustrate the essentials of I/O, some of the basic calls are summarized below in an anonymous language that will indicate the required parameters without getting into the underlying complexities. Each call to the system may potentially result in an error return, which for simplicity is not represented in the calling sequence.

To read or write a file assumed to exist already, it must be opened by the following call:

```
filep = open ( name, flag )
```

where **name** indicates the name of the file. An arbitrary path name may be given. The **flag** argument indicates whether the file is to be read, written, or “updated,” that is, read and written simultaneously.

The returned value **filep** is called a *file descriptor*. It is a small integer used to identify the file in subsequent calls to read, write, or otherwise manipulate the file.

To create a new file or completely rewrite an old one, there is a **create** system call that creates the given file if it does not exist, or truncates it to zero length if it does exist; **create** also opens the new file for writing and, like **open**, returns a file descriptor.

The file system maintains no locks visible to the user, nor is there any restriction on the number of users who may have a file open for reading or writing. Although it is possible for the contents of a file to become scrambled when two users write on it simultaneously, in practice difficulties do not arise. We take the view that locks are neither necessary nor sufficient, in our environment, to prevent interference between users of the same file. They are unnecessary because we are not faced with large, single-file data bases maintained by independent processes. They are insufficient because locks in the ordinary sense, whereby one user is prevented from writing on a file that another user is reading, cannot prevent confusion when, for example, both users are editing a file with an editor that makes a copy of the file being edited.

There are, however, sufficient internal interlocks to maintain the logical consistency of the file system when two users engage simultaneously in activities such as writing on the same file, creating files in the same directory, or deleting each other’s open files.

Except as indicated below, reading and writing are sequential. This means that if a particular byte in the file was the last byte written (or read), the next I/O call implicitly refers to the immediately following byte. For each open file there is a pointer, maintained inside the system, that indicates the next byte to be read or written. If n bytes are read or written, the pointer advances by n bytes.

Once a file is open, the following calls may be used:

```
n = read ( filep, buffer, count )
n = write ( filep, buffer, count )
```

Up to **count** bytes are transmitted between the file specified by **filep** and the byte array specified by **buffer**. The returned value **n** is the number of bytes actually transmitted. In the **write** case, **n** is the same as **count** except under exceptional conditions, such as I/O errors or end of physical medium on special files; in a **read**, however, **n** may without error be less than **count**. If the read pointer is so near the end of the file that reading **count** characters would cause reading beyond the end, only sufficient bytes are transmitted to reach the end of the file; also,

typewriter-like terminals never return more than one line of input. When a **read** call returns with **n** equal to zero, the end of the file has been reached. For disk files this occurs when the read pointer becomes equal to the current size of the file. It is possible to generate an end-of-file from a terminal by use of an escape sequence that depends on the device used.

Bytes written affect only those parts of a file implied by the position of the write pointer and the count; no other part of the file is changed. If the last byte lies beyond the end of the file, the file is made to grow as needed.

To do random (direct-access) I/O it is only necessary to move the read or write pointer to the appropriate location in the file.

```
location = lseek ( filep, offset, base )
```

The pointer associated with **filep** is moved to a position **offset** bytes from the beginning of the file, from the current position of the pointer, or from the end of the file, depending on **base**. **offset** may be negative. For some devices (e.g., paper tape and terminals) seek calls are ignored. The actual offset from the beginning of the file to which the pointer was moved is returned in **location**.

There are several additional system entries having to do with I/O and with the file system that will not be discussed. For example: close a file, get the status of a file, change the protection mode or the owner of a file, create a directory, make a link to an existing file, delete a file.

IV. IMPLEMENTATION OF THE FILE SYSTEM

As mentioned in Section 3.2 above, a directory entry contains only a name for the associated file and a pointer to the file itself. This pointer is an integer called the *i-number* (for index number) of the file. When the file is accessed, its *i-number* is used as an index into a system table (the *i-list*) stored in a known part of the device on which the directory resides. The entry found thereby (the file's *i-node*) contains the description of the file:

- the user and group-ID of its owner
- its protection bits
- the physical disk or tape addresses for the file contents
- its size
- time of creation, last use, and last modification
- the number of links to the file, that is, the number of times it appears in a directory
- a code indicating whether the file is a directory, an ordinary file, or a special file.

The purpose of an **open** or **create** system call is to turn the path name given by the user into an *i-number* by searching the explicitly or implicitly named directories. Once a file is open, its device, *i-number*, and read/write pointer are stored in a system table indexed by the file descriptor returned by the **open** or **create**. Thus, during a subsequent call to read or write the file, the descriptor may be easily related to the information necessary to access the file.

When a new file is created, an *i-node* is allocated for it and a directory entry is made that contains the name of the file and the *i-node* number. Making a link to an existing file involves creating a directory entry with the new name, copying the *i-number* from the original file entry, and incrementing the link-count field of the *i-node*. Removing (deleting) a file is done by decrementing the link-count of the *i-node* specified by its directory entry and erasing the directory entry. If the link-count drops to 0, any disk blocks in the file are freed and the *i-node* is de-allocated.

The space on all disks that contain a file system is divided into a number of 512-byte blocks logically addressed from 0 up to a limit that depends on the device. There is space in the *i-node* of each file for 13 device addresses. For nonspecial files, the first 10 device addresses point at the first 10 blocks of the file. If the file is larger than 10 blocks, the 11 device address points to an indirect block containing up to 128 addresses of additional blocks in the

file. Still larger files use the twelfth device address of the i-node to point to a double-indirect block naming 128 indirect blocks, each pointing to 128 blocks of the file. If required, the thirteenth device address is a triple-indirect block. Thus files may conceptually grow to $[(10+128+128^2+128^3)\times 512]$ bytes. Once opened, bytes numbered below 5,120 can be read with a single disk access; bytes in the range 5,120 to 70,656 require two accesses; bytes in the range 70,656 to 8,459,264 require three accesses; bytes from there to the largest file (1,082,201,088) require four accesses. In practice, a device cache mechanism (see below) proves effective in eliminating most of the indirect fetches.

The foregoing discussion applies to ordinary files. When an I/O request is made to a file whose i-node indicates that it is special, the last 12 device address words are immaterial, and the first specifies an internal *device name*, which is interpreted as a pair of numbers representing, respectively, a device type and subdevice number. The device type indicates which system routine will deal with I/O on that device; the subdevice number selects, for example, a disk drive attached to a particular controller or one of several similar terminal interfaces.

In this environment, the implementation of the **mount** system call (Section 3.4) is quite straightforward. **mount** maintains a system table whose argument is the i-number and device name of the ordinary file specified during the **mount**, and whose corresponding value is the device name of the indicated special file. This table is searched for each i-number/device pair that turns up while a path name is being scanned during an **open** or **create**; if a match is found, the i-number is replaced by the i-number of the root directory and the device name is replaced by the table value.

To the user, both reading and writing of files appear to be synchronous and unbuffered. That is, immediately after return from a **read** call the data are available; conversely, after a **write** the user's workspace may be reused. In fact, the system maintains a rather complicated buffering mechanism that reduces greatly the number of I/O operations required to access a file. Suppose a **write** call is made specifying transmission of a single byte. The system will search its buffers to see whether the affected disk block currently resides in main memory; if not, it will be read in from the device. Then the affected byte is replaced in the buffer and an entry is made in a list of blocks to be written. The return from the **write** call may then take place, although the actual I/O may not be completed until a later time. Conversely, if a single byte is read, the system determines whether the secondary storage block in which the byte is located is already in one of the system's buffers; if so, the byte can be returned immediately. If not, the block is read into a buffer and the byte picked out.

The system recognizes when a program has made accesses to sequential blocks of a file, and asynchronously pre-reads the next block. This significantly reduces the running time of most programs while adding little to system overhead.

A program that reads or writes files in units of 512 bytes has an advantage over a program that reads or writes a single byte at a time, but the gain is not immense; it comes mainly from the avoidance of system overhead. If a program is used rarely or does no great volume of I/O, it may quite reasonably read and write in units as small as it wishes.

The notion of the i-list is an unusual feature of UNIX. In practice, this method of organizing the file system has proved quite reliable and easy to deal with. To the system itself, one of its strengths is the fact that each file has a short, unambiguous name related in a simple way to the protection, addressing, and other information needed to access the file. It also permits a quite simple and rapid algorithm for checking the consistency of a file system, for example, verification that the portions of each device containing useful information and those free to be allocated are disjoint and together exhaust the space on the device. This algorithm is independent of the directory hierarchy, because it need only scan the linearly organized i-list. At the same time the notion of the i-list induces certain peculiarities not found in other file system organizations. For example, there is the question of who is to be charged for the space a file occupies, because all directory entries for a file have equal status. Charging the owner of a file is unfair in general, for one user may create a file, another may link to it, and the first user may delete the file. The first user is still the owner of the file, but it should be charged to the

second user. The simplest reasonably fair algorithm seems to be to spread the charges equally among users who have links to a file. Many installations avoid the issue by not charging any fees at all.

V. PROCESSES AND IMAGES

An *image* is a computer execution environment. It includes a memory image, general register values, status of open files, current directory and the like. An image is the current state of a pseudo-computer.

A *process* is the execution of an image. While the processor is executing on behalf of a process, the image must reside in main memory; during the execution of other processes it remains in main memory unless the appearance of an active, higher-priority process forces it to be swapped out to the disk.

The user-memory part of an image is divided into three logical segments. The program text segment begins at location 0 in the virtual address space. During execution, this segment is write-protected and a single copy of it is shared among all processes executing the same program. At the first hardware protection byte boundary above the program text segment in the virtual address space begins a non-shared, writable data segment, the size of which may be extended by a system call. Starting at the highest address in the virtual address space is a stack segment, which automatically grows downward as the stack pointer fluctuates.

5.1 Processes

Except while the system is bootstrapping itself into operation, a new process can come into existence only by use of the **fork** system call:

```
processid = fork ( )
```

When **fork** is executed, the process splits into two independently executing processes. The two processes have independent copies of the original memory image, and share all open files. The new processes differ only in that one is considered the parent process: in the parent, the returned **processid** actually identifies the child process and is never 0, while in the child, the returned value is always 0.

Because the values returned by **fork** in the parent and child process are distinguishable, each process may determine whether it is the parent or child.

5.2 Pipes

Processes may communicate with related processes using the same system **read** and **write** calls that are used for file-system I/O. The call:

```
filep = pipe ( )
```

returns a file descriptor **filep** and creates an inter-process channel called a *pipe*. This channel, like other open files, is passed from parent to child process in the image by the **fork** call. A **read** using a pipe file descriptor waits until another process writes using the file descriptor for the same pipe. At this point, data are passed between the images of the two processes. Neither process need know that a pipe, rather than an ordinary file, is involved.

Although inter-process communication via pipes is a quite valuable tool (see Section 6.2), it is not a completely general mechanism, because the pipe must be set up by a common ancestor of the processes involved.

5.3 Execution of programs

Another major system primitive is invoked by

```
execute ( file, arg1, arg2, . . . , argn )
```

which requests the system to read in and execute the program named by **file**, passing it string

arguments **arg₁**, **arg₂**, ..., **arg_n**. All the code and data in the process invoking **execute** is replaced from the **file**, but open files, current directory, and inter-process relationships are unaltered. Only if the call fails, for example because **file** could not be found or because its execute-permission bit was not set, does a return take place from the **execute** primitive; it resembles a "jump" machine instruction rather than a subroutine call.

5.4 Process synchronization

Another process control system call:

```
processid = wait ( status )
```

causes its caller to suspend execution until one of its children has completed execution. Then **wait** returns the **processid** of the terminated process. An error return is taken if the calling process has no descendants. Certain status from the child process is also available.

5.5 Termination

Lastly:

```
exit ( status )
```

terminates a process, destroys its image, closes its open files, and generally obliterates it. The parent is notified through the **wait** primitive, and **status** is made available to it. Processes may also terminate as a result of various illegal actions or user-generated signals (Section VII below).

VI. THE SHELL

For most users, communication with the system is carried on with the aid of a program called the shell. The shell is a command-line interpreter: it reads lines typed by the user and interprets them as requests to execute other programs. (The shell is described fully elsewhere,⁹ so this section will discuss only the theory of its operation.) In simplest form, a command line consists of the command name followed by arguments to the command, all separated by spaces:

```
command arg1 arg2 ... argn
```

The shell splits up the command name and the arguments into separate strings. Then a file with name **command** is sought; **command** may be a path name including the "/" character to specify any file in the system. If **command** is found, it is brought into memory and executed. The arguments collected by the shell are accessible to the command. When the command is finished, the shell resumes its own execution, and indicates its readiness to accept another command by typing a prompt character.

If file **command** cannot be found, the shell generally prefixes a string such as **/bin/** to **command** and attempts again to find the file. Directory **/bin** contains commands intended to be generally used. (The sequence of directories to be searched may be changed by user request.)

6.1 Standard I/O

The discussion of I/O in Section III above seems to imply that every file used by a program must be opened or created by the program in order to get a file descriptor for the file. Programs executed by the shell, however, start off with three open files with file descriptors 0, 1, and 2. As such a program begins execution, file 1 is open for writing, and is best understood as the standard output file. Except under circumstances indicated below, this file is the user's terminal. Thus programs that wish to write informative information ordinarily use file descriptor 1. Conversely, file 0 starts off open for reading, and programs that wish to read messages typed by the user read this file.

The shell is able to change the standard assignments of these file descriptors from the user's terminal printer and keyboard. If one of the arguments to a command is prefixed by

“>”, file descriptor 1 will, for the duration of the command, refer to the file named after the “>”. For example:

```
ls
```

ordinarily lists, on the typewriter, the names of the files in the current directory. The command:

```
ls >there
```

creates a file called **there** and places the listing there. Thus the argument **>there** means “place output on **there**.” On the other hand:

```
ed
```

ordinarily enters the editor, which takes requests from the user via his keyboard. The command

```
ed <script
```

interprets **script** as a file of editor commands; thus **<script** means “take input from **script**.”

Although the file name following “<” or “>” appears to be an argument to the command, in fact it is interpreted completely by the shell and is not passed to the command at all. Thus no special coding to handle I/O redirection is needed within each command; the command need merely use the standard file descriptors 0 and 1 where appropriate.

File descriptor 2 is, like file 1, ordinarily associated with the terminal output stream. When an output-diversion request with “>” is specified, file 2 remains attached to the terminal, so that commands may produce diagnostic messages that do not silently end up in the output file.

6.2 Filters

An extension of the standard I/O notion is used to direct output from one command to the input of another. A sequence of commands separated by vertical bars causes the shell to execute all the commands simultaneously and to arrange that the standard output of each command be delivered to the standard input of the next command in the sequence. Thus in the command line:

```
ls | pr -2 | opr
```

ls lists the names of the files in the current directory; its output is passed to **pr**, which paginates its input with dated headings. (The argument “-2” requests double-column output.) Likewise, the output from **pr** is input to **opr**; this command spools its input onto a file for off-line printing.

This procedure could have been carried out more clumsily by:

```
ls >templ
pr -2 <templ >temp2
opr <temp2
```

followed by removal of the temporary files. In the absence of the ability to redirect output and input, a still clumsier method would have been to require the **ls** command to accept user requests to paginate its output, to print in multi-column format, and to arrange that its output be delivered off-line. Actually it would be surprising, and in fact unwise for efficiency reasons, to expect authors of commands such as **ls** to provide such a wide variety of output options.

A program such as **pr** which copies its standard input to its standard output (with processing) is called a *filter*. Some filters that we have found useful perform character transliteration, selection of lines according to a pattern, sorting of the input, and encryption and decryption.

6.3 Command separators; multitasking

Another feature provided by the shell is relatively straightforward. Commands need not be on different lines; instead they may be separated by semicolons:

```
ls; ed
```

will first list the contents of the current directory, then enter the editor.

A related feature is more interesting. If a command is followed by “&,” the shell will not wait for the command to finish before prompting again; instead, it is ready immediately to accept a new command. For example:

```
as source >output &
```

causes **source** to be assembled, with diagnostic output going to **output**; no matter how long the assembly takes, the shell returns immediately. When the shell does not wait for the completion of a command, the identification number of the process running that command is printed. This identification may be used to wait for the completion of the command or to terminate it. The “&” may be used several times in a line:

```
as source >output & ls >files &
```

does both the assembly and the listing in the background. In these examples, an output file other than the terminal was provided; if this had not been done, the outputs of the various commands would have been intermingled.

The shell also allows parentheses in the above operations. For example:

```
(date; ls) >x &
```

writes the current date and time followed by a list of the current directory onto the file **x**. The shell also returns immediately for another request.

6.4 The shell as a command; command files

The shell is itself a command, and may be called recursively. Suppose file **tryout** contains the lines:

```
as source
mv a.out testprog
testprog
```

The **mv** command causes the file **a.out** to be renamed **testprog**. **a.out** is the (binary) output of the assembler, ready to be executed. Thus if the three lines above were typed on the keyboard, **source** would be assembled, the resulting program renamed **testprog**, and **testprog** executed. When the lines are in **tryout**, the command:

```
sh <tryout
```

would cause the shell **sh** to execute the commands sequentially.

The shell has further capabilities, including the ability to substitute parameters and to construct argument lists from a specified subset of the file names in a directory. It also provides general conditional and looping constructions.

6.5 Implementation of the shell

The outline of the operation of the shell can now be understood. Most of the time, the shell is waiting for the user to type a command. When the new-line character ending the line is typed, the shell's **read** call returns. The shell analyzes the command line, putting the arguments in a form appropriate for **execute**. Then **fork** is called. The child process, whose code of course is still that of the shell, attempts to perform an **execute** with the appropriate arguments. If successful, this will bring in and start execution of the program whose name was given. Meanwhile, the other process resulting from the **fork**, which is the parent process, waits for the

child process to die. When this happens, the shell knows the command is finished, so it types its prompt and reads the keyboard to obtain another command.

Given this framework, the implementation of background processes is trivial; whenever a command line contains “&,” the shell merely refrains from waiting for the process that it created to execute the command.

Happily, all of this mechanism meshes very nicely with the notion of standard input and output files. When a process is created by the **fork** primitive, it inherits not only the memory image of its parent but also all the files currently open in its parent, including those with file descriptors 0, 1, and 2. The shell, of course, uses these files to read command lines and to write its prompts and diagnostics, and in the ordinary case its children—the command programs—inherit them automatically. When an argument with “<” or “>” is given, however, the offspring process, just before it performs **execute**, makes the standard I/O file descriptor (0 or 1, respectively) refer to the named file. This is easy because, by agreement, the smallest unused file descriptor is assigned when a new file is **opened** (or **created**); it is only necessary to close file 0 (or 1) and open the named file. Because the process in which the command program runs simply terminates when it is through, the association between a file specified after “<” or “>” and file descriptor 0 or 1 is ended automatically when the process dies. Therefore the shell need not know the actual names of the files that are its own standard input and output, because it need never reopen them.

Filters are straightforward extensions of standard I/O redirection with pipes used instead of files.

In ordinary circumstances, the main loop of the shell never terminates. (The main loop includes the branch of the return from **fork** belonging to the parent process; that is, the branch that does a **wait**, then reads another command line.) The one thing that causes the shell to terminate is discovering an end-of-file condition on its input file. Thus, when the shell is executed as a command with a given input file, as in:

```
sh <comfile
```

the commands in **comfile** will be executed until the end of **comfile** is reached; then the instance of the shell invoked by **sh** will terminate. Because this shell process is the child of another instance of the shell, the **wait** executed in the latter will return, and another command may then be processed.

6.6 Initialization

The instances of the shell to which users type commands are themselves children of another process. The last step in the initialization of the system is the creation of a single process and the invocation (via **execute**) of a program called **init**. The role of **init** is to create one process for each terminal channel. The various subinstances of **init** open the appropriate terminals for input and output on files 0, 1, and 2, waiting, if necessary, for carrier to be established on dial-up lines. Then a message is typed out requesting that the user log in. When the user types a name or other identification, the appropriate instance of **init** wakes up, receives the log-in line, and reads a password file. If the user's name is found, and if he is able to supply the correct password, **init** changes to the user's default current directory, sets the process's user ID to that of the person logging in, and performs an **execute** of the shell. At this point, the shell is ready to receive commands and the logging-in protocol is complete.

Meanwhile, the mainstream path of **init** (the parent of all the subinstances of itself that will later become shells) does a **wait**. If one of the child processes terminates, either because a shell found an end of file or because a user typed an incorrect name or password, this path of **init** simply recreates the defunct process, which in turn reopens the appropriate input and output files and types another log-in message. Thus a user may log out simply by typing the end-of-file sequence to the shell.

6.7 Other programs as shell

The shell as described above is designed to allow users full access to the facilities of the system, because it will invoke the execution of any program with appropriate protection mode. Sometimes, however, a different interface to the system is desirable, and this feature is easily arranged for.

Recall that after a user has successfully logged in by supplying a name and password, **init** ordinarily invokes the shell to interpret command lines. The user's entry in the password file may contain the name of a program to be invoked after log-in instead of the shell. This program is free to interpret the user's messages in any way it wishes.

For example, the password file entries for users of a secretarial editing system might specify that the editor **ed** is to be used instead of the shell. Thus when users of the editing system log in, they are inside the editor and can begin work immediately; also, they can be prevented from invoking programs not intended for their use. In practice, it has proved desirable to allow a temporary escape from the editor to execute the formatting program and other utilities.

Several of the games (e.g., chess, blackjack, 3D tic-tac-toe) available on the system illustrate a much more severely restricted environment. For each of these, an entry exists in the password file specifying that the appropriate game-playing program is to be invoked instead of the shell. People who log in as a player of one of these games find themselves limited to the game and unable to investigate the (presumably more interesting) offerings of the UNIX system as a whole.

VII. TRAPS

The PDP-11 hardware detects a number of program faults, such as references to non-existent memory, unimplemented instructions, and odd addresses used where an even address is required. Such faults cause the processor to trap to a system routine. Unless other arrangements have been made, an illegal action causes the system to terminate the process and to write its image on file **core** in the current directory. A debugger can be used to determine the state of the program at the time of the fault.

Programs that are looping, that produce unwanted output, or about which the user has second thoughts may be halted by the use of the **interrupt** signal, which is generated by typing the "delete" character. Unless special action has been taken, this signal simply causes the program to cease execution without producing a **core** file. There is also a **quit** signal used to force an image file to be produced. Thus programs that loop unexpectedly may be halted and the remains inspected without prearrangement.

The hardware-generated faults and the interrupt and quit signals can, by request, be either ignored or caught by a process. For example, the shell ignores quits to prevent a quit from logging the user out. The editor catches interrupts and returns to its command level. This is useful for stopping long printouts without losing work in progress (the editor manipulates a copy of the file it is editing). In systems without floating-point hardware, unimplemented instructions are caught and floating-point instructions are interpreted.

VIII. PERSPECTIVE

Perhaps paradoxically, the success of the UNIX system is largely due to the fact that it was not designed to meet any predefined objectives. The first version was written when one of us (Thompson), dissatisfied with the available computer facilities, discovered a little-used PDP-7 and set out to create a more hospitable environment. This (essentially personal) effort was sufficiently successful to gain the interest of the other author and several colleagues, and later to justify the acquisition of the PDP-11/20, specifically to support a text editing and formatting system. When in turn the 11/20 was outgrown, the system had proved useful enough to persuade management to invest in the PDP-11/45, and later in the PDP-11/70 and Interdata 8/32 machines, upon which it developed to its present form. Our goals throughout the effort, when

articulated at all, have always been to build a comfortable relationship with the machine and to explore ideas and inventions in operating systems and other software. We have not been faced with the need to satisfy someone else's requirements, and for this freedom we are grateful.

Three considerations that influenced the design of UNIX are visible in retrospect.

First: because we are programmers, we naturally designed the system to make it easy to write, test, and run programs. The most important expression of our desire for programming convenience was that the system was arranged for interactive use, even though the original version only supported one user. We believe that a properly designed interactive system is much more productive and satisfying to use than a "batch" system. Moreover, such a system is rather easily adaptable to noninteractive use, while the converse is not true.

Second: there have always been fairly severe size constraints on the system and its software. Given the partially antagonistic desires for reasonable efficiency and expressive power, the size constraint has encouraged not only economy, but also a certain elegance of design. This may be a thinly disguised version of the "salvation through suffering" philosophy, but in our case it worked.

Third: nearly from the start, the system was able to, and did, maintain itself. This fact is more important than it might seem. If designers of a system are forced to use that system, they quickly become aware of its functional and superficial deficiencies and are strongly motivated to correct them before it is too late. Because all source programs were always available and easily modified on-line, we were willing to revise and rewrite the system and its software when new ideas were invented, discovered, or suggested by others.

The aspects of UNIX discussed in this paper exhibit clearly at least the first two of these design considerations. The interface to the file system, for example, is extremely convenient from a programming standpoint. The lowest possible interface level is designed to eliminate distinctions between the various devices and files and between direct and sequential access. No large "access method" routines are required to insulate the programmer from the system calls; in fact, all user programs either call the system directly or use a small library program, less than a page long, that buffers a number of characters and reads or writes them all at once.

Another important aspect of programming convenience is that there are no "control blocks" with a complicated structure partially maintained by and depended on by the file system or other system calls. Generally speaking, the contents of a program's address space are the property of the program, and we have tried to avoid placing restrictions on the data structures within that address space.

Given the requirement that all programs should be usable with any file or device as input or output, it is also desirable to push device-dependent considerations into the operating system itself. The only alternatives seem to be to load, with all programs, routines for dealing with each device, which is expensive in space, or to depend on some means of dynamically linking to the routine appropriate to each device when it is actually needed, which is expensive either in overhead or in hardware.

Likewise, the process-control scheme and the command interface have proved both convenient and efficient. Because the shell operates as an ordinary, swappable user program, it consumes no "wired-down" space in the system proper, and it may be made as powerful as desired at little cost. In particular, given the framework in which the shell executes as a process that spawns other processes to perform commands, the notions of I/O redirection, background processes, command files, and user-selectable system interfaces all become essentially trivial to implement.

Influences

The success of UNIX lies not so much in new inventions but rather in the full exploitation of a carefully selected set of fertile ideas, and especially in showing that they can be keys to the implementation of a small yet powerful operating system.

The **fork** operation, essentially as we implemented it, was present in the GENIE time-sharing system.¹⁰ On a number of points we were influenced by Multics, which suggested the particular form of the I/O system calls¹¹ and both the name of the shell and its general functions. The notion that the shell should create a process for each command was also suggested to us by the early design of Multics, although in that system it was later dropped for efficiency reasons. A similar scheme is used by TENEX.¹²

IX. STATISTICS

The following numbers are presented to suggest the scale of the Research UNIX operation. Those of our users not involved in document preparation tend to use the system for program development, especially language work. There are few important "applications" programs.

Overall, we have today:

125	user population
33	maximum simultaneous users
1,630	directories
28,300	files
301,700	512-byte secondary storage blocks used

There is a "background" process that runs at the lowest possible priority; it is used to soak up any idle CPU time. It has been used to produce a million-digit approximation to the constant e , and other semi-infinite problems. Not counting this background work, we average daily:

13,500	commands
9.6	CPU hours
230	connect hours
62	different users
240	log-ins

X. ACKNOWLEDGEMENTS

The contributors to UNIX are, in the traditional but here especially apposite phrase, too numerous to mention. Certainly, collective salutes are due to our colleagues in the Computing Science Research Center. R. H. Canaday contributed much to the basic design of the file system. We are particularly appreciative of the inventiveness, thoughtful criticism, and constant support of R. Morris, M. D. McIlroy, and J. F. Ossanna.

REFERENCES

- [1] L. P. Deutsch and B. W. Lampson. An online editor, *CACM* **10**(12):793-99,803 (December 1967).
- [2] B. W. Kernighan and L. L. Cherry. A System for Typesetting Mathematics, *CACM* **18**(3):151-57 (March 1975).
- [3] B. W. Kernighan, M. E. Lesk, and J. F. Ossanna. UNIX Time-Sharing System: Document Preparation, *Bell Sys. Tech. J.* **57**(6):2115-35 (July-August 1978, Part 2).
- [4] T. A. Dolotta and J. R. Mashey. An Introduction to the Programmer's Workbench, *Proc. 2nd Int. Conf. on Software Engineering*, pp. 164-68 (October 13-15, 1976).
- [5] T. A. Dolotta, R. C. Haight, and J. R. Mashey. UNIX Time-Sharing System: The Programmer's Workbench, *Bell Sys. Tech. J.* **57**(6):2177-2200 (July-August 1978, Part 2).

- [6] H. Lycklama. UNIX Time-Sharing System: UNIX on a Microprocessor, *Bell Sys. Tech. J.* **57**(6):2087-2101 (July-August 1978, Part 2).
- [7] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey (1978).
- [8] Aleph-null. Computer Recreations, *Software Practice & Experience* **1**(2):210-4 (April-June 1971).
- [9] S. R. Bourne. UNIX Time-Sharing System: The UNIX Shell, *Bell Sys. Tech. J.* **57**(6):1971-90 (July-August 1978, Part 2).
- [10] L. P. Deutsch and B. W. Lampson. SDS 930 Time-Sharing System Preliminary Reference Manual, Doc. 30.10.10, Project GENIE, Univ. Cal. at Berkeley (April 1965).
- [11] R. J. Feiertag and E. I. Organick. The Multics Input-Output System, *Proc. Third Symposium on Operating Systems Principles*, pp. 35-41 (October 18-20, 1971).
- [12] D. G. Bobrow, J. D. Burchfiel, D. L. Murphy, and R. S. Tomlinson. TENEX, a Paged Time Sharing System for the PDP-10, *CACM* **15**(3):135-43 (March 1972).

January 1981

UNIX Documentation Road Map

G. A. Snyder
J. R. Mashey

Bell Laboratories
Murray Hill, New Jersey 07974

1. INTRODUCTION

A great deal of documentation exists for the UNIX† time-sharing system. New users are often overcome by the volume and distributed nature of the documentation. This “road map” attempts to be a terse, up-to-date outline of important documents and information sources.

✗ The information in this document applies only to UNIX Release 4.0.

1.1 Things to Do

See a local UNIX “system administrator” to obtain a “login name” and get other appropriate system information. See also Section 12 below.

1.2 Notation Used in This Road Map (B.1.1) ●●

- {N} → Section N in this road map.
- → Item required for everyone.
- → Item recommended for most users.

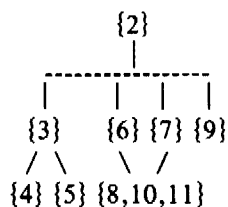
All other items are optional and depend on specific interests. If the name of a document mentioned here is followed by a number such as “(A.1.1),” then that document can be found in *Documents for UNIX*. Examine Section G of the *Annotated Table of Contents* in that volume for additional sources of information.

Entries in Section *n* of the *UNIX User's Manual* are referred to by *name(n)*.

1.3 List of Following Sections ●●

- {2} BASIC INFORMATION
- {3} BASIC TEXT PROCESSING AND DOCUMENT PREPARATION
- {4} SPECIALIZED TEXT PROCESSING
- {5} ADVANCED TEXT PROCESSING
- {6} COMMAND LANGUAGE (SHELL) PROGRAMMING
- {7} FILE MANIPULATION
- {8} C PROGRAMMING
- {9} NUMERICAL COMPUTATION
- {10} SOURCE CODE CONTROL SYSTEM
- {11} INTER-SYSTEM COMMUNICATION
- {12} LOCAL INFORMATION

1.4 Prerequisite Structure of Following Sections ●●



† UNIX is a trademark of Bell Laboratories.

Each section contains a list of relevant documents and a list of pertinent manual entries; some sections also contain a list of suggested things to do.

Only the manual entries for the most frequently used commands are listed here; other relevant entries may be found by consulting the *Table of Contents* and the *Permuted Index* of the *UNIX User's Manual* {2.1}; it is also wise to periodically scan Section 1 of that manual—you will often discover new uses for commands.

2. BASIC INFORMATION

You won't be able to do much until you have learned most of the material in {2.1}, {2.2}, and {2.3}. You must know how to log into the system, make your terminal work correctly, enter and edit files, and perform basic operations on directories and files. Get the *UNIX Programming Starter Package* from your local Computer Information Service Library.

2.1 UNIX User's Manual ●●

- Read *Introduction* and *How to Get Started*.
- Read the *intro* entry in each section.
- Look through Section 1 to become familiar with command names.
- Get into the habit of using the *Table of Contents* and the *Permuted Index*.

Section 1 will be especially needed for reference use.

2.2 UNIX for Beginners (Second Edition) (B.3.1) ●●

2.3 A Tutorial Introduction to the UNIX Text Editor (B.2.1) ●●

2.4 Advanced Editing on UNIX (B.2.2) ●

2.5 The Bell System Technical Journal, Vol. 57, No. 6, Part 2 ●

Contains several articles on UNIX. In particular, the first paper gives a good overview of UNIX.

2.6 Things to Do

- Do all the exercises found in {2.2} and {2.3}, and maybe {2.4}.
- If you want some sequence of commands to be executed each time you log in, create a file named `.profile` in your login directory.¹ A sample `.profile` can be found in *profile* (5).
- Files in directory `/usr/news` contain recent information on various topics. To print all the news items that have been added since you last looked, type:

```
news
```

2.7 Manual Entries

The following commands are described in Section 1 of the *UNIX User's Manual* and are used for creating, editing, moving (i.e., renaming), and removing files:

<code>cat(1)</code>	concatenate and print files (no pagination).
<code>cd(1)</code>	change working (current) directory.
<code>chmod(1)</code>	change the mode of a file.
<code>cp(1)</code>	copy (<i>cp</i>), move (<i>mv</i>) or link (<i>ln</i>) files.
<code>ed(1)</code>	edit a file.
<code>ls(1)</code>	list a directory; file names beginning with <code>.</code> are not listed unless the <code>-a</code> flag is used.
<code>mkdir(1)</code>	make a (new) directory.

1. The directory you are in when you log into the system.

pr(1)	print files (paginated listings).
pwd(1)	print working directory.
rm(1)	remove (delete) file(s); <i>rmdir</i> removes the named directories, which must be empty.

The following help you communicate with other users, make proper use of different kinds of terminals, and print manual entries on-line:

login(1)	sign on.
mail(1)	send mail to other users or inspect mail from them.
man(1)	print entries of <i>UNIX User's Manual</i> .
mesg(1)	permit or deny messages to your terminal.
news(1)	print news items: <i>news -n</i> prints a list of recent items.
passwd(1)	change your login password.
stty(1)	set terminal options; i.e., inform the system about the hardware characteristics of your terminal.
tabs(1)	set tab stops on your terminal.
term(7)	a list of commonly-used terminals.
who(1)	print list of currently logged-in users.
write(1)	communicate with another (logged-in) user.

Several useful status commands also exist:

date(1)	print time and date.
du(1)	summarize disk usage.
ps(1)	report active process status.

3. BASIC TEXT PROCESSING AND DOCUMENT PREPARATION

You should read this section if you want to *use* existing text processing tools to write letters, memoranda, manuals, etc. Get the *UNIX Text Editing and Phototypesetting Starter Package* from your local Computer Information Service Library.

3.1 MM—Memorandum Macros (C.2.1) ●●

This is a reference manual that can be moderately heavy going for a beginner. Try out some of the examples and stick close to the default options.

3.2 Typing Documents with MM (C.2.2) ●●

A handy fold-out.

3.3 A TROFF Tutorial (C.1.1) ●

An introduction to formatting text with the phototypesetter.

3.4 NROFF/TROFF User's Manual (C.1.2) ●

Describes the text formatting language in great detail; look at the *SUMMARY AND INDEX*, but don't try to digest the whole manual on first reading.

3.5 Manual Entries

mm(1)	print a document using the memorandum macros.
troff(1)	typeset or format (<i>nroff</i>) text files; read this to become familiar with options.
spell(1)	identify possible spelling errors.

To obtain some special functions (e.g., reverse paper motion, subscripts, superscripts), you must either indicate the terminal type to *nroff* or post-process *nroff* output through one of the following:

col(1)	process text for terminals lacking physical reverse vertical motion, such as the Texas Instruments 700 series, Model 43 <i>TELETYPE</i> [®] , etc.
greek(1)	handle special functions for many terminals, such as DASI 300, Tektronix 4014, Diablo 1620, Hewlett-Packard 2645, etc.
tc(1)	simulate phototypesetter output on a Tektronix 4014 terminal.

4. SPECIALIZED TEXT PROCESSING

The tools listed here are of a more specialized nature than those in {3}.

4.1 TBL—A Program to Format Tables (C.3.1) ●

Great help in formatting tabular data (see also *tbl(1)*).

4.2 Typesetting Mathematics—User's Guide (Second Edition) (C.3.2) ●

Read this if you need to produce mathematical equations. It describes the use of the equation-setting command *eqn(1)*.

4.3 A Macro Package for View Graphs and Slides (C.2.3)

Tells how to prepare typeset visuals.

4.4 UNIX Graphics Overview (E.6.1)

Describes the Graphics sub-system of UNIX.

4.5 Manual Entries

cw(1)	use a special constant-width "example" font.
diffmk(1)	mark changes between versions of a file, using output of <i>diff(1)</i> to produce "revision bars" in the right margin.
eqn(1)	preprocessor for mathematical equations.
eqnchar(7)	special character definitions for <i>eqn(1)</i> .
graphics(1G)	get into the graphics sub-system.
mmt(1)	typeset documents, view graphs, and slides.
tbl(1)	preprocessor for tabular data.

5. ADVANCED TEXT PROCESSING

You should read this section if you need to *design* your own package of formatting macros or perform other actions beyond the capabilities of existing tools; {3} is a prerequisite, and familiarity with {4} is very helpful, as is an experienced advisor.

5.1 NROFF/TROFF User's Manual (C.1.2) ●●

Look at this in detail and try modifying the examples. Read *A TROFF Tutorial* {3.3}.

5.2 Things to Do

It is fairly easy to use the text formatters for simple purposes. A typical application is that of writing simple macros that print standard headings in order to eliminate repetitive keying of such headings. It is extremely difficult to set up general-purpose macro packages for use by large numbers of people. Don't re-invent what you can borrow from an existing package (such as MM—see {3.1} and {3.2}).

5.3 Manual Entries

All entries mentioned in {3.5} and {4.5}.

6. COMMAND LANGUAGE (SHELL) PROGRAMMING

The shell provides a powerful programming language for combining existing commands. This section should be especially useful to those who want to automate manual procedures and build data bases.

6.1 The UNIX Time-Sharing System (A.1.2) ●●

6.2 UNIX Shell Tutorial (B.4.1) ●●

6.3 An Introduction to the UNIX Shell (B.4.2)

6.4 Things to Do

If you want to create your own library of commands, for example `/usr/gas/bin`, set the `PATH` parameter in your `.profile` so that your own library is searched when a command is invoked. For example:

```
PATH=:$HOME/bin:/bin:/usr/bin
```

The `HOME` parameter is described in `sh(1)`.

6.5 Manual Entries

Read `sh(1)` first; the following entries give further details on commands that are most frequently used within command language programs:

<code>echo(1)</code>	echo arguments (typically to terminal).
<code>env(1)</code>	set environment for command execution.
<code>expr(1)</code>	evaluate an algebraic expression; includes some string operations.
<code>line(1)</code>	read a line from the standard input.
<code>nohup(1)</code>	run a command immune to communications line hang-up.
<code>sh(1)</code>	shell (command interpreter and programming language).
<code>test(1)</code>	evaluate a logical expression.

7. FILE MANIPULATION

In addition to the basic commands of {2}, many UNIX commands exist to perform various kinds of file manipulation. Small data bases can often be managed quite simply by combining text processing {5}, shell programming {6}, and the commands listed below in {7.3}.

7.1 SED—A Non-Interactive Text Editor (B.2.3)

7.2 AWK—A Pattern Scanning and Processing Language (E.3.1)

7.3 Manual Entries

The starred (*) items below are especially useful for dealing with “fielded data,” i.e., data where each line is a sequence of delimited fields. The following are used to search or edit files in a single pass:

<code>awk(1)*</code>	perform actions on lines matching specified patterns.
<code>grep(1)</code>	search a file for a pattern; more powerful and specialized versions include <i>egrep</i> and <i>fgrep</i> .
<code>sed(1)*</code>	stream editor.
<code>tr(1)</code>	transliterate (substitute or delete specified characters).

The following compare files in different ways:

<code>cmp(1)</code>	compare files (byte by byte).
<code>comm(1)</code>	print lines common to and/or different in two files.
<code>diff(1)</code>	differential file comparator (minimal editing for conversion).

The following combine files and/or split them apart:

ar(1)	archiver and library maintainer.
cpio(1)	general file copying and archiving.
cut(1)*	cut out selected fields of each line of a file.
join(1)	join two relations specified by the lines of two files.
paste(1)*	merge lines from several files.
split(1)	split file into chunks of specified size.

The following interrogate files and print information about them:

file(1)	determine file type (best guess).
od(1)	octal dump (and other kinds also).
sum(1)	sum and count blocks in a file.
wc(1)	word (and line and character) count.

Miscellaneous commands:

find(1)	search directory structure for specified kinds of files.
sort(1)*	sort or merge files.
tail(1)	print the last part of a file.
tee(1)	copy single input to several output files.
uniq(1)*	report repeated lines in a file, or obtain unique ones.

8. C PROGRAMMING

Try to use existing tools first, before writing C programs at all.

8.1 The C Programming Language

A book written by B. W. Kernighan and D. M. Ritchie; published by Prentice Hall (1978). It contains comprehensive text and includes a tutorial and a reference manual. Read the tutorial; try the examples. Check for updates to the reference manual {8.2} from time to time.

8.2 The C Programming Language—Reference Manual (D.1.1) ●●

8.3 UNIX Programming (D.3.1) ●

8.4 A Guide to the C Library for UNIX Users (D.1.2) ●

8.5 SDB—A Symbolic Debugger (D.5.1)

8.6 YACC—Yet Another Compiler-Compiler (E.1.2)

8.7 LEX—A Lexical Analyzer Generator (E.1.1)

8.8 LINT, a C Program Checker (D.1.3)

8.9 MAKE—A Program for Maintaining Computer Programs (D.4.1)

8.10 An Augmented Version of MAKE (D.4.2)

8.11 Things to Do

Read {8.1} and do some of the exercises. A good way to become familiar with C is to look at the source code of existing programs, especially ones whose functions are well known to you. Much code can be found in directory `/usr/src`. In particular, the directory `cmd` contains the source for most of the commands. Also, investigate directory `/usr/include`.

8.12 Manual Entries

ar(1)	archive and library maintainer.
cc(1)	compile C programs.
ld(1)	link edit object files; you must know about some of its flags.

lex(1)	generate lexical analyzers.
lint(1)	verify C programs.
lorder(1)	find ordering relation for an object library.
make(1)	automate program (re)generation procedures.
nm(1)	print name (i.e., symbol) list.
prof(1)	display profile data; used for program optimization.
ps(1)	report active process status.
sdb(1)	debug C and F77 programs symbolically on the VAX 11/780.
strip(1)	remove symbols and relocation bits from executable files.
time(1)	time a command.
yacc(1)	parser generator.

9. NUMERICAL COMPUTATION

9.1 DC—An Interactive Desk Calculator (E.5.2)

9.2 BC—An Arbitrary Precision Desk-Calculator Language (E.5.1)

9.3 AWK—A Pattern Scanning and Processing Language (E.3.1)

9.4 A Portable FORTRAN 77 Compiler (D.2.1)

9.5 RATFOR—A Preprocessor for a Rational FORTRAN (D.2.2)

9.6 SDB—A Symbolic Debugger (D.5.1)

9.7 Manual Entries

awk(1)	perform actions on lines matching specified patterns.
bc(1)	an interactive language, acts as front end for <i>dc</i> (1).
bs(1)	a compiler/interpreter for modest-sized programs.
dc(1)	a desk calculator.
f77(1)	a FORTRAN compiler.
ratfor(1)	a rational FORTRAN dialect.
sdb(1)	debug C and F77 programs symbolically on the VAX 11/780.

10. SOURCE CODE CONTROL SYSTEM

10.1 Source Code Control System User's Guide (E.4.1) ●

10.2 Manual Entries

admin(1)	create and administer SCCS files.
cdc(1)	change the delta commentary of an SCCS file.
comb(1)	combine deltas of an SCCS file.
delta(1)	create a new version or delta of a file under SCCS control.
get(1)	get a particular version of an SCCS file, usually for editing.
help(1)	print helpful error messages and information about a command.
prs(1)	print delta information of an SCCS file in a specified format.
rmdel(1)	remove a delta.
sact(1)	print current SCCS file editing activity.
sccsdiff(1)	print the different lines between two deltas of an SCCS file.
unset(1)	undo the version control mechanism created by a <i>get</i> for editing.
val(1)	validate an SCCS file.
what(1)	print out embedded information lines placed in a file by SCCS.

11. INTER-SYSTEM COMMUNICATION

11.1 A Dial-up Network of UNIX Systems (E.8.1) ●

11.2 UNIX Remote Job Entry User's Guide (E.7.1) ●

11.3 Manual Entries

The following commands (most of which are site-dependent) are useful in communicating with other systems:

cu(1C)	call another system.
dpr(1C)	print files off-line at a specified destination.
fget(1C)	retrieve files from the HONEYWELL 6000.
fsend(1C)	send files to the HONEYWELL 6000.
gcat(1C)	send phototypesetter output to the HONEYWELL 6000.
send(1C)	send files to an IBM host for execution using Remote Job Entry.
uucp(1C)	copy files from one UNIX system to another.
uux(1C)	execute command(s) on another UNIX system.

12. LOCAL INFORMATION

☞ This section should be provided by each individual UNIX installation.

January 1981

A Tutorial Introduction to the UNIX Text Editor

Brian W. Kernighan

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Almost all text input on the UNIX† operating system is done with the text-editor *ed*. This memorandum is a tutorial guide to help beginners get started with text editing.

Although it does not cover everything, it does discuss enough for most users' day-to-day needs. This includes printing, appending, changing, deleting, moving and inserting entire lines of text; reading and writing files; context searching and line addressing; the substitute command; the global commands; and the use of special characters for advanced editing.

Introduction

Ed is a "text editor", that is, an interactive program for creating and modifying "text", using directions provided by a user at a terminal. The text is often a document like this one, or a program or perhaps data for a program.

This introduction is meant to simplify learning *ed*. The recommended way to learn *ed* is to read this document, simultaneously using *ed* to follow the examples, then to read the description in Section 1 of the *UNIX User's Manual*, all the while experimenting with *ed*. (Solicitation of advice from experienced users is also useful.)

Do the exercises! They cover material not completely discussed in the actual text. An appendix summarizes the commands.

Disclaimer

This is an introduction and a tutorial. For this reason, no attempt is made to cover more than a part of the facilities that *ed* offers (although this fraction includes the most useful and frequently used parts). When you have mastered the Tutorial, try *Advanced Editing on UNIX*. Also, there is not enough space to explain basic UNIX procedures. We will assume that you know how to log on to UNIX, and that you have at least a vague understanding of what a file is. For more on that, read *UNIX for Beginners*.

You must also know what character to type as the end-of-line on your particular terminal. This character is the RETURN key on most terminals. Throughout, we will refer to this character, whatever it is, as RETURN.

Getting Started

We'll assume that you have logged in to your system and it has just printed the prompt character, usually either a \$ or a %. The easiest way to get *ed* is to type

```
ed      (followed by a return)
```

You are now ready to go — *ed* is waiting for you to tell it what to do.

Creating Text — The Append Command "a"

As your first problem, suppose you want to create some text starting from scratch. Perhaps you are typing the very first draft of a paper; clearly it will have to start somewhere, and undergo modifications later. This section will show how to get some text in, just to get started. Later we'll talk about how to change it.

When *ed* is first started, it is rather like working with a blank piece of paper — there is no text or information present. This must be supplied by the person using *ed*; it is usually done by typing in the text, or by reading it into *ed* from a file. We will start by typing in some text, and return shortly to how to read files.

† UNIX is a trademark of Bell Laboratories.

First a bit of terminology. In *ed* jargon, the text being worked on is said to be "kept in a buffer." Think of the buffer as a work space, if you like, or simply as the information that you are going to be editing. In effect the buffer is like the piece of paper, on which we will write things, then change some of them, and finally file the whole thing away for another day.

The user tells *ed* what to do to his text by typing instructions called "commands." Most commands consist of a single letter, which must be typed in lower case. Each command is typed on a separate line. (Sometimes the command is preceded by information about what line or lines of text are to be affected — we will discuss these shortly.) *Ed* makes no response to most commands — there is no prompting or typing of messages like "ready". (This silence is preferred by experienced users, but sometimes a hangup for beginners.)

The first command is *append*, written as the letter

a

all by itself. It means "append (or add) text lines to the buffer, as I type them in." Appending is rather like writing fresh material on a piece of paper.

So to enter lines of text into the buffer, just type an **a** followed by a RETURN, followed by the lines of text you want, like this:

a

Now is the time
for all good men
to come to the aid of their party.

The only way to stop appending is to type a line that contains only a period. The "." is used to tell *ed* that you have finished appending. (Even experienced users forget that terminating "." sometimes. If *ed* seems to be ignoring you, type an extra line with just "." on it. You may then find you've added some garbage lines to your text, which you'll have to take out later.)

After the append command has been done, the buffer will contain the three lines

Now is the time
for all good men
to come to the aid of their party.

The "a" and "." aren't there, because they are not text.

To add more text to what you already have, just issue another **a** command, and continue typing.

Error Messages — "?"

If at any time you make an error in the commands you type to *ed*, it will tell you by typing

?

This is about as cryptic as it can be, but with practice, you can usually figure out how you goofed. In some versions of *ed*, you can get a brief explanation of the error by typing

h

Writing Text out as a File — The Write Command "w"

It's likely that you'll want to save your text for later use. To write out the contents of the buffer onto a file, use the *write* command

w

followed by the file name you want to write on. This will copy the buffer's contents onto the specified file (destroying any previous information on the file). To save the text on a file named **junk**, for example, type

w junk

Leave a space between **w** and the file name. *Ed* will respond by printing the number of characters it wrote out. In this case, *ed* would respond with

68

(Remember that blanks and the return character at the end of each line are included in the character count.) Writing a file just makes a copy of the text — the buffer's contents are not disturbed, so you can go on adding lines to it. This is an important point. *Ed* at all times works on a copy of a file, not the file itself. No change in the contents of a file takes place until you give a **w** command. (Writing out the text onto a file from time to time as it is being created is a good idea, since if the system crashes or if you make some horrible mistake, you will lose all the text in the buffer but any text that was written onto a file is relatively safe.)

Leaving *ed* — The Quit Command "q"

To terminate a session with *ed*, first save your text by writing it onto a file using the **w** command, and then type the *quit* command

q

The system will respond with the prompt character (\$ or %). At this point your buffer vanishes, with all its text, which is why you want to write it out before quitting.*

* Actually, *ed* will print ? if you try to quit without writing. At that point, write if you want; if not, another **q** will get you out regardless.

Exercise 1:

Enter *ed* and create some text using

```
a
... text ...
.
```

Write it out using *w*. Then leave *ed* with the *q* command, and print the file, to see that everything worked. (To print a file, say

```
pr file_name
```

or

```
cat file_name
```

in response to the prompt character. Try both.)

Reading Text From a File — The Edit Command “e”

A common way to get text into the buffer is to read it from a file in the file system. This is what you do to edit text that you saved with the *w* command in a previous session. The *edit* command *e* fetches the entire contents of a file into the buffer. So if you had saved the three lines “Now is the time”, etc., with a *w* command in an earlier session, the *ed* command

```
e junk
```

would fetch the entire contents of the file *junk* into the buffer, and respond

```
68
```

which is the number of characters in *junk*. *If anything was already in the buffer, it is deleted first.*

If you use the *e* command to read a file into the buffer, then you need not use a file name after a subsequent *w* command; *ed* remembers the last file name used in an *e* command, and *w* will write on this file. Thus a good way to operate is

```
ed
e file
[editing session]
w
q
```

This way, you can simply say *w* from time to time, and be secure in the knowledge that if you got the file name right at the beginning, you are writing into the proper file each time.

You can find out at any time what file name *ed* is remembering by typing the *file* command *f*. In this example, if you typed

```
f
```

ed would reply

```
junk
```

Reading Text From a File — The Read Command “r”

Sometimes you want to read a file into the buffer without destroying anything that is already there. This is done by the *read* command *r*. The command

```
r junk
```

will read the file *junk* into the buffer; it adds it to the end of whatever is already in the buffer. So if you do a read after an edit:

```
e junk
r junk
```

the buffer will contain *two* copies of the text (six lines).

```
Now is the time
for all good men
to come to the aid of their party.
Now is the time
for all good men
to come to the aid of their party.
```

Like the *w* and *e* commands, *r* prints the number of characters read in, after the reading operation is complete.

Generally speaking, *r* is much less used than *e*.

Exercise 2:

Experiment with the *e* command — try reading and printing various files. You may get an error *?name*, where *name* is the name of a file; this means that the file doesn't exist, typically because you spelled the file name wrong, or perhaps that you are not allowed to read or write it. Try alternately reading and appending to see that they work similarly. Verify that

```
ed file_name
```

is exactly equivalent to

```
ed
e file_name
```

What does

```
f file_name
```

do?

Printing the Contents of the Buffer — The Print Command “p”

To *print* or list the contents of the buffer (or parts of it) on the terminal, use the print command

```
p
```

The way this is done is as follows. Specify the lines where you want printing to begin and where

you want it to end, separated by a comma, and followed by the letter **p**. Thus to print the first two lines of the buffer, for example, (that is, lines 1 through 2) say

1,2p (starting line=1, ending line=2 p)

Ed will respond with

Now is the time
for all good men

Suppose you want to print *all* the lines in the buffer. You could use **1,3p** as above if you knew there were exactly 3 lines in the buffer. But in general, you don't know how many there are, so what do you use for the ending line number? *Ed* provides a shorthand symbol for "line number of last line in buffer" — the dollar sign **\$**. Use it this way:

1,\$p

This will print *all* the lines in the buffer (line 1 to last line); **1,\$p** can be abbreviated **.p**. If you want to stop the printing before it is finished, push the DEL or Delete key; *ed* will type

?

and wait for the next command.

To print the *last* line of the buffer, you could use

,\$p

but *ed* lets you abbreviate this to

\$p

You can print any single line by typing the line number followed by a **p**. Thus

1p

produces the response

Now is the time

which is the first line of the buffer.

In fact, *ed* lets you abbreviate even further: you can print any single line by typing *just* the line number — no need to type the letter **p**. So if you say

\$

ed will print the last line of the buffer.

You can also use **\$** in combinations like

\$-1,\$p

which prints the last two lines of the buffer. This helps when you want to see how far you got in typing.

Exercise 3:

As before, create some text using the **a** command and experiment with the **p** command. You will find, for example, that you can't print line 0 or a line beyond the end of the buffer, and that attempts to print a buffer in reverse order by saying

3,1p

don't work.

The Current Line — "Dot" or "."

Suppose your buffer still contains the six lines as above, that you have just typed

1,3p

and *ed* has printed the three lines for you. Try typing just

p (no line numbers)

This will print

to come to the aid of their party.

which is the third line of the buffer. In fact it is the last (most recent) line that you have done anything with. (You just printed it!) You can repeat this **p** command without line numbers, and it will continue to print line 3.

The reason is that *ed* maintains a record of the last line that you did anything to (in this case, line 3, which you just printed) so that it can be used instead of an explicit line number. This most recent line is referred to by the shorthand symbol

(pronounced "dot").

Dot is a line number in the same way that **\$** is; it means exactly "the current line", or loosely, "the line you most recently did something to." You can use it in several ways — one possibility is to say

.,\$p

This will print all the lines from (including) the current line to the end of the buffer. In our example these are lines 3 through 6.

Some commands change the value of dot, while others do not. The **p** command sets dot to the number of the last line printed; the last command will set both **.** and **\$** to 6.

Dot is most useful when used in combinations like this one:

+.1 (or equivalently, **+.1p**)

This means "print the next line" and is a handy way to step slowly through a buffer. You can also say

`.-1` (or `.-1p`)

which means "print the line *before* the current line." This enables you to go backwards if you wish. Another useful one is something like

`.-3,.-1p`

which prints the previous three lines.

Don't forget that all of these change the value of dot. You can find out what dot is at any time by typing

`.=`

Ed will respond by printing the value of dot.

Let's summarize some things about the `p` command and dot. Essentially `p` can be preceded by 0, 1, or 2 line numbers. If there is no line number given, it prints the "current line", the line that dot refers to. If there is one line number given (with or without the letter `p`), it prints that line (and dot is set there); and if there are two line numbers, it prints all the lines in that range (and sets dot to the last line printed.) If two line numbers are specified the first can't be bigger than the second (see Exercise 2.)

Typing a single return will cause printing of the next line — it's equivalent to `+.1p`. Try it. Try typing a `-`; you will find that it's equivalent to `.-1p`.

Deleting Lines — The Delete Command "d"

Suppose you want to get rid of the three extra lines in the buffer. This is done by the *delete* command

`d`

Except that `d` deletes lines instead of printing them, its action is similar to that of `p`. The lines to be deleted are specified for `d` exactly as they are for `p`:

starting line, ending line d

Thus the command

`4,$d`

deletes lines 4 through the end. There are now three lines left, as you can check by using

`1,$p`

And notice that `$` now is line 3! Dot is set to the next line after the last line deleted, unless the last line deleted is the last line in the buffer. In that case, dot is set to `$`.

Exercise 4:

Experiment with `a`, `e`, `r`, `w`, `p` and `d` until you are sure that you know what they do, and until you understand how dot, `$`, and line numbers are used.

If you are adventurous, try using line numbers with `a`, `r` and `w` as well. You will find that `a` will append lines *after* the line number that you specify (rather than after dot); that `r` reads a file in *after* the line number you specify (not necessarily at the end of the buffer); and that `w` will write out exactly the lines you specify, not necessarily the whole buffer. These variations are sometimes handy. For instance you can insert a file at the beginning of a buffer by saying

Or `file_name`

and you can enter lines at the beginning of the buffer by saying

```
0a
... text ...
.
```

Notice that `.w` is *very* different from

```
.
w
```

Modifying Text — The Substitute Command "s"

We are now ready to try one of the most important of all commands — the substitute command

`s`

This is the command that is used to change individual words or letters within a line or group of lines. It is what you use, for example, for correcting spelling mistakes and typing errors.

Suppose that, because of a typing error, line 1 says

`Now is th time`

namely, the `e` has been left off the `the`. You can use `s` to fix this up as follows:

`1s/th/the/`

This says: "in line 1, substitute for the characters `th` the characters `the`." To verify that it works (*ed* will not print the result automatically) say

`p`

and get

`Now is the time`

which is what you wanted. Notice that dot must have been set to the line where the substitution

took place, since the **p** command printed that line. Dot is always set this way with the **s** command.

The general way to use the substitute command is

starting-line, ending-line s/change this/to this/

Whatever string of characters is between the first pair of slashes is replaced by whatever is between the second pair, in *all* the lines between *starting-line* and *ending-line*. Only the first occurrence on each line is changed, however. If you want to change *every* occurrence, see Exercise 5. The rules for line numbers are the same as those for **p**, except that dot is set to the last line changed. (But there is a trap for the unwary: if no substitution took place, dot is *not* changed. This causes an error ? as a warning.)

Thus you can say

1,\$s/speling/spelling/

and correct the first spelling mistake on each line in the text. (This is useful for people who are consistent misspellers!)

If no line numbers are given, the **s** command assumes we mean "make the substitution on line dot", so it changes things only on the current line. This leads to the very common sequence

s/something/something else/p

which makes some correction on the current line, and then prints it, to make sure it worked out right. If it didn't, you can try again. (Notice that there is a **p** on the same line as the **s** command. With few exceptions, **p** can follow any command; no other multi-command lines are legal.)

It's also legal to say

s/...//

which means "change the first string of characters to *"nothing"*, i.e., remove them. This is useful for deleting extra words in a line or removing extra letters from words. For instance, if you had

Nowxx is the time

you can say

s/xx//p

to get

Now is the time

Notice that **//** (two adjacent slashes) means "no characters", not a blank. There *is* a difference! (See below for another meaning of **//**.)

Exercise 5:

Experiment with the substitute command. See what happens if you substitute for some word on a line with several occurrences of that word. For example, do this:

```
a
the other side of the coin
.
s/the/on the/p
```

You will get

```
on the other side of the coin
```

A substitute command changes only the first occurrence of the first string. You can change all occurrences by adding a **g** (for "global") to the **s** command, like this:

```
s/.../.../gp
```

Try other characters instead of slashes to delimit the two sets of characters in the **s** command — anything should work except blanks or tabs.

(If you get funny results using any of the characters

```
^ . $ [ * \ &
```

read the section on "Special Characters".)

Context Searching — **"/.../"**

With the substitute command mastered, you can move on to another highly important idea of *ed* — context searching.

Suppose you have the original three line text in the buffer:

```
Now is the time
for all good men
to come to the aid of their party.
```

Suppose you want to find the line that contains *their* so you can change it to *the*. Now with only three lines in the buffer, it's pretty easy to keep track of what line the word *their* is on. But if the buffer contained several hundred lines, and you'd been making changes, deleting and rearranging lines, and so on, you would no longer really know what this line number would be. Context searching is simply a method of specifying the desired line, regardless of what its number is, by specifying some context on it.

The way to say "search for a line that contains this particular string of characters" is to type

```
/string of characters we want to find/
```

For example, the *ed* command

```
/their/
```

is a context search which is sufficient to find the desired line — it will locate the next occurrence of the characters between slashes (“their”). It also sets dot to that line and prints the line for verification:

to come to the aid of their party.

“Next occurrence” means that *ed* starts looking for the string at line `.+1`, searches to the end of the buffer, then continues at line 1 and searches to line dot. (That is, the search “wraps around” from `$` to 1.) It scans all the lines in the buffer until it either finds the desired line or gets back to dot again. If the given string of characters can’t be found in any line, *ed* types the error message

?

Otherwise it prints the line it found.

You can do both the search for the desired line *and* a substitution all at once, like this:

`/their/s/their/the/p`

which will yield

to come to the aid of the party.

There were three parts to that last command: context search for the desired line, make the substitution, print the line.

The expression `/their/` is a context search expression. In their simplest form, all context search expressions are like this — a string of characters surrounded by slashes. Context searches are interchangeable with line numbers, so they can be used by themselves to find and print a desired line, or as line numbers for some other command, like `s`. They were used both ways in the examples above.

Suppose the buffer contains the three familiar lines

Now is the time
for all good men
to come to the aid of their party.

Then the *ed* line numbers

`/Now/+1`
`/good/`
`/party/-1`

are all context search expressions, and they all refer to the same line (line 2). To make a change in line 2, you could say

`/Now/+1s/good/bad/`

or

`/good/s/good/bad/`

or

`/party/-1s/good/bad/`

The choice is dictated only by convenience. You could print all three lines by, for instance

`/Now/,/party/p`

or

`/Now/,/Now/+2p`

or by any number of similar combinations. The first one of these might be better if you don’t know how many lines are involved. (Of course, if there were only three lines in the buffer, you’d use

`1,$p`

but not if there were several hundred.)

The basic rule is: a context search expression is *the same as* a line number, so it can be used wherever a line number is needed.

Exercise 6:

Experiment with context searching. Try a body of text with several occurrences of the same string of characters, and scan through it using the same context search.

Try using context searches as line numbers for the substitute, print and delete commands. (They can also be used with `r`, `w`, and `a`.)

Try context searching using `?text?` instead of `/text/`. This scans lines in the buffer in reverse order rather than normal. This is sometimes useful if you go too far while looking for some string of characters — it’s an easy way to back up.

(If you get funny results with any of the characters

`^ . $ [* \ &`

read the section on “Special Characters”.)

Ed provides a shorthand for repeating a context search for the same string. For example, the *ed* line number

`/string/`

will find the next occurrence of `string`. It often happens that this is not the desired line, so the search must be repeated. This can be done by typing merely

`//`

This shorthand stands for “the most recently used context search expression.” It can also be used as the first string of the substitute command, as in

`/string1/s//string2/`

which will find the next occurrence of **string1** and replace it by **string2**. This can save a lot of typing. Similarly

??

means "scan backwards for the same expression."

Change and Insert — The "c" and "i" Commands

This section discusses the *change* command

c

which is used to change or replace a group of one or more lines, and the *insert* command

i

which is used for inserting a group of one or more lines.

"Change", written as

c

is used to replace a number of lines with different lines, which are typed in at the terminal. For example, to change lines **.+1** through **\$** to something else, type

```
.+1,$c
... type the lines of text you want here ...
```

The lines you type between the **c** command and the **.** will take the place of the original lines between start line and end line. This is most useful in replacing a line or several lines which have errors in them.

If only one line is specified in the **c** command, then just that line is replaced. (You can type in as many replacement lines as you like.) Notice the use of **.** to end the input — this works just like the **.** in the append command and must appear by itself on a new line. If no line number is given, line dot is replaced. The value of dot is set to the last line you typed in.

"Insert" is similar to append — for instance

```
/string/i
... type the lines to be inserted here ...
```

will insert the given text *before* the next line that contains "string". The text between **i** and **.** is *inserted before* the specified line. If no line number is specified dot is used. Dot is set to the last line inserted.

Exercise 7:

"Change" is rather like a combination of delete followed by insert. Experiment to verify that

```
start, end d
i
... text ...
.
```

is almost the same as

```
start, end c
... text ...
.
```

These are not *precisely* the same if line **\$** gets deleted. Check this out. What is dot?

Experiment with **a** and **i**, to see that they are similar, but not the same. You will observe that

```
line-number a
... text ...
.
```

appends *after* the given line, while

```
line-number i
... text ...
.
```

inserts *before* it. Observe that if no line number is given, **i** inserts before line dot, while **a** appends after line dot.

Moving Text Around — The "m" Command

The move command **m** is used for cutting and pasting — it lets you move a group of lines from one place to another in the buffer. Suppose you want to put the first three lines of the buffer at the end instead. You could do it by saying:

```
1,3w temp
$r temp
1,3d
```

(Do you see why?) but you can do it a lot easier with the **m** command:

```
1,3m$
```

The general case is

```
start line, end line m after this line
```

Notice that there is a third line to be specified — the place where the moved stuff gets put. Of course the lines to be moved can be specified by context searches; if you had

```

First paragraph
...
end of first paragraph.
Second paragraph
...
end of second paragraph.

```

you could reverse the two paragraphs like this:

```
/Second/./end of second/m/First/-1
```

Notice the `-1`: the moved text goes *after* the line mentioned. Dot is set to the last line moved.

The Global Commands “g” and “v”

The *global* command `g` is used to execute one or more *ed* commands on all those lines in the buffer that match some specified string. For example

```
g/peling/p
```

prints all lines that contain `peling`. More usefully,

```
g/peling/s//pelling/gp
```

makes the substitution everywhere on the line, then prints each corrected line. Compare this to

```
1,$s/peling/pelling/gp
```

which only prints the last line substituted. Another subtle difference is that the `g` command does not give a `?` if `peling` is not found where the `s` command will.

There may be several commands (including `a`, `c`, `i`, `r`, `w`, but not `g`); in that case, every line except the last must end with a backslash `\`:

```

g/xxx/.-1s/abc/def/\
.+2s/ghi/jkl/\
.-2,.p

```

makes changes in the lines before and after each line that contains `xxx`, then prints all three lines.

The `v` command is the same as `g`, except that the commands are executed on every line that does *not* match the string following `v`:

```
v/ /d
```

deletes every line that does not contain a blank.

Special Characters

You may have noticed that things just don't work right when you used some characters like `.`, `*`, `$`, and others in context searches and the substitute command. The reason is rather complex, although the cure is simple. Basically, *ed* treats these characters as special, with special meanings. For instance, *in a context search or the first string of the substitute command only*, `.` means

“any character,” not a period, so

```
/x.y/
```

means “a line with an `x`, *any character*, and a `y`,” *not* just “a line with an `x`, a period, and a `y`.” A complete list of the special characters that can cause trouble is the following:

```
^ . $ [ * \
```

Warning: The backslash character `\` is special to *ed*. For safety's sake, avoid it where possible. If you have to use one of the special characters in a substitute command, you can turn off its magic meaning temporarily by preceding it with the backslash. Thus

```
s/\\.\*/backslash dot star/
```

will change `\.*` into “backslash dot star”.

Here is a hurried synopsis of the other special characters. First, the circumflex `^` signifies the beginning of a line. Thus

```
/^ string/
```

finds `string` only if it is at the beginning of a line: it will find

```
string
```

but not

```
the string ...
```

The dollar-sign `$` is just the opposite of the circumflex; it means the end of a line:

```
/string$/
```

will only find an occurrence of `string` that is at the end of some line. This implies, of course, that

```
/^ string$/
```

will find only a line that contains just `string`, and

```
/^ ./
```

finds a line containing exactly one character.

The character `.`, as we mentioned above, matches anything;

```
/x.y/
```

matches any of

```

x+y
x-y
x y
x.y

```

This is useful in conjunction with `*`, which is a repetition character; `a*` is a shorthand for “any number of `a`'s,” so `.*` matches any number of anythings. This is used like this:

```
s./*/stuff/
```

which changes an entire line, or

```
s/.*//
```

which deletes all characters in the line up to and including the last comma. (Since `.*` finds the longest possible match, this goes up to the last comma.)

[is used with] to form “character classes”; for example,

```
/[0123456789]/
```

matches any single digit — any one of the characters inside the braces will cause a match. This can be abbreviated to `[0-9]`.

Finally, the `&` is another shorthand character — it is used only on the right-hand part of a substitute command where it means “whatever was matched on the left-hand side”. It is used to save typing. Suppose the current line contained

```
Now is the time
```

and you wanted to put parentheses around it. You could just retype the line, but this is tedious. Or you could say

```
s/^/(/
s/$)/
```

using your knowledge of `^` and `$`. But the easiest way uses the `&`:

```
s./*/(&)/
```

This says “match the whole line, and replace it by itself surrounded by parentheses.” The `&` can be used several times in a line; consider using

```
s./*/&? &!!/
```

to produce

```
Now is the time? Now is the time!!
```

You don’t have to match the whole line, of course: if the buffer contains

```
the end of the world
```

you could type

```
/world/s//& is at hand/
```

to produce

```
the end of the world is at hand
```

Observe this expression carefully, for it illustrates how to take advantage of `ed` to save typing. The string `/world/` found the desired line; the shorthand `//` found the same word in the line; and the `&` saves you from typing it again.

The `&` is a special character only within the replacement text of a substitute command, and

has no special meaning elsewhere. You can turn off the special meaning of `&` by preceding it with a `\`:

```
s/ampersand/\&/
```

will convert the word “ampersand” into the literal symbol `&` in the current line.

Summary of Commands and Line Numbers

The general form of `ed` commands is the command name, perhaps preceded by one or two line numbers, and, in the case of `e`, `r`, and `w`, followed by a file name. Only one command is allowed per line, but a `p` command may follow any other command (except for `e`, `r`, `w` and `q`).

a: Append, that is, add lines to the buffer (at line dot, unless a different line is specified). Appending continues until `.` is typed on a new line. Dot is set to the last line appended.

c: Change the specified lines to the new text which follows. The new lines are terminated by a `.`, as with `a`. If no lines are specified, replace line dot. Dot is set to last line changed.

d: Delete the lines specified. If none are specified, delete line dot. Dot is set to the first undeleted line, unless `$` is deleted, in which case dot is set to `$`.

e: Edit new file. Any previous contents of the buffer are thrown away, so issue a `w` beforehand.

f: Print remembered file name. If a name follows `f` the remembered name will be set to it.

g: The command

```
g/---/commands
```

will execute the commands on those lines that contain `---`, which can be any context search expression.

i: Insert lines before specified line (or dot) until a `.` is typed on a new line. Dot is set to last line inserted.

m: Move lines specified to after the line named after `m`. Dot is set to the last line moved.

p: Print specified lines. If none specified, print line dot. A single line number is equivalent to *line-number* `p`. A single return prints `.+1`, the next line.

q: Quit `ed`. Wipes out all text in buffer if you give it twice in a row without first giving a `w` command.

r: Read a file into buffer (at end unless specified elsewhere.) Dot is set to last line read.

s: The command

```
s/string1/string2/
```

substitutes the characters **string1** into **string2** in the specified lines. If no lines are specified, make the substitution in line dot. Dot is set to last line in which a substitution took place, which means that if no substitution took place, dot is not changed. **s** changes only the first occurrence of **string1** on a line; to change all of them, type a **g** after the final slash.

v: The command

v/---/commands

executes **commands** on those lines that *do not* contain ---.

w: Write out buffer onto a file. Dot is not changed.

.=: Print value of dot. (= by itself prints the value of **\$**.)

!: The line

!command-line

causes **command-line** to be executed as a UNIX command.

/-----/: Context search. Search for next line which contains this string of characters. Print it. Dot is set to the line where string was found. Search starts at **+.1**, wraps around from **\$** to 1, and continues to dot, if necessary.

?-----?: Context search in reverse direction. Start search at **.-1**, scan to 1, wrap around to **\$**.

January 1981

Advanced Editing on UNIX

Brian W. Kernighan

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This paper is meant to help secretaries, typists and programmers to make effective use of the UNIX† facilities for preparing and editing text. It provides explanations and examples of

- special characters, line addressing and global commands in the editor `ed`;
- commands for “cut and paste” operations on files and parts of files, including the `mv`, `cp`, `cat` and `rm` commands, and the `r`, `w`, `m` and `t` commands of the editor;
- editing scripts and editor-based programs like `grep` and `sed`.

Although the treatment is aimed at non-programmers, new UNIX users with any background should find helpful hints on how to get their jobs done more easily.

1. INTRODUCTION

Although UNIX provides remarkably effective tools for text editing, that by itself is no guarantee that everyone will automatically make the most effective use of them. In particular, people who are not computer specialists — typists, secretaries, casual users — often use the system less effectively than they might.

This document is intended as a sequel to *A Tutorial Introduction to the UNIX Text Editor* [1], providing explanations and examples of how to edit with less effort. (You should also be familiar with the material in *UNIX For Beginners* [2].) Further information on all commands discussed here can be found in the *UNIX User's Manual* [3].

Examples are based on observations of users and the difficulties they encounter. Topics covered include special characters in searches and substitute commands, line addressing, the global commands, and line moving and copying. There are also brief discussions of effective use of related tools, like those for file manipulation, and those based on `ed`, like `grep` and `sed`.

A word of caution. There is only one way to learn to use something, and that is to *use* it. Reading a description is no substitute for trying

something. A paper like this one should give you ideas about what to try, but until you actually try something, you will not learn it.

2. SPECIAL CHARACTERS

The editor `ed` is the primary interface to the system for many people, so it is worthwhile to know how to get the most out of `ed` for the least effort.

The next few sections will discuss shortcuts and labor-saving devices. Not all of these will be instantly useful to any one person, of course, but a few will be, and the others should give you ideas to store away for future use. And as always, until you try these things, they will remain theoretical knowledge, not something you have confidence in.

The List Command ‘l’

`ed` provides two commands for printing the contents of the lines you’re editing. Most people are familiar with `p`, in combinations like

```
1,$p
```

to print all the lines you’re editing, or

```
s/abc/def/p
```

to change ‘abc’ to ‘def’ on the current line. Less

† UNIX is a trademark of Bell Laboratories.

familiar is the *list* command **l** (the letter 'l'), which gives slightly more information than **p**. In particular, **l** makes visible characters that are normally invisible, such as tabs and backspaces. If you list a line that contains some of these, **l** will print each tab as \Rightarrow and each backspace as \Leftarrow . This makes it much easier to correct the sort of typing mistake that inserts extra spaces adjacent to tabs, or inserts a backspace followed by a space.

The **l** command also 'folds' long lines for printing — any line that exceeds 72 characters is printed on multiple lines; each printed line except the last is terminated by a backslash \backslash , so you can tell it was folded. This is useful for printing long lines on short terminals.

Occasionally the **l** command will print in a line a string of numbers preceded by a backslash, such as $\backslash 07$ or $\backslash 16$. These combinations are used to make visible characters that normally don't print, like form feed or vertical tab or bell. Each such combination is a single character. When you see such characters, be wary — they may have surprising meanings when printed on some terminals. Often their presence means that your finger slipped while you were typing; you almost never want them.

The Substitute Command 's'

Most of the next few sections will be taken up with a discussion of the substitute command **s**. Since this is the command for changing the contents of individual lines, it probably has the most complexity of any **ed** command, and the most potential for effective use.

As the simplest place to begin, recall the meaning of a trailing **g** after a substitute command. With

```
s/this/that/
```

and

```
s/this/that/g
```

the first one replaces the *first* 'this' on the line with 'that'. If there is more than one 'this' on the line, the second form with the trailing **g** changes *all* of them.

Either form of the **s** command can be followed by **p** or **l** to 'print' or 'list' (as described in the previous section) the contents of the line:

```
s/this/that/p
s/this/that/l
s/this/that/gp
s/this/that/gl
```

are all legal, and mean slightly different things. Make sure you know what the differences are.

Of course, any **s** command can be preceded by one or two 'line numbers' to specify that the substitution is to take place on a group of lines. Thus

```
1,$s/mispell/misspell/
```

changes the *first* occurrence of 'mispell' to 'misspell' on every line of the file. But

```
1,$s/mispell/misspell/g
```

changes *every* occurrence in every line (and this is more likely to be what you wanted in this particular case).

You should also notice that if you add a **p** or **l** to the end of any of these substitute commands, only the last line that got changed will be printed, not all the lines. We will talk later about how to print all the lines that were modified.

The Undo Command 'u'

Occasionally, you will make a substitution in a line, only to realize too late that it was a ghastly mistake. The 'undo' command **u** lets you 'undo' the last command, so that the last line that was substituted can be restored to its previous state by typing the command

```
u
```

The Metacharacter '.'

As you have undoubtedly noticed when you use **ed**, certain characters have unexpected meanings when they occur in the left side of a substitute command, or in a search for a particular line. In the next several sections, we will talk about these special characters, which are often called 'metacharacters'.

The first one is the period '.'. On the left side of a substitute command, or in a search with $\backslash \dots /$, '.' stands for *any* single character. Thus the search

```
/x.y/
```

finds any line where 'x' and 'y' occur separated by a single character, as in

```
x+y
x-y
x□y
x.y
```

and so on. (We will use \square to stand for a space whenever we need to make it visible.)

Since '.' matches a single character, that gives you a way to deal with funny characters printed by **l**. Suppose you have a line that, when printed with the **l** command, appears as

... th\07is ...

and you want to get rid of the \07 (which represents the bell character, by the way).

The most obvious solution is to try

```
s/\07//
```

but this will fail. (Try it.) The brute force solution, which most people would now take, is to re-type the entire line. This is guaranteed, and is actually quite a reasonable tactic if the line in question isn't too big, but for a very long line, re-typing is a bore. This is where the metacharacter '.' comes in handy. Since '\07' really represents a single character, if we say

```
s/th.is/this/
```

the job is done. The '.' matches the mysterious character between the 'h' and the 'i', *whatever it is*.

Bear in mind that since '.' matches any single character, the command

```
s/././
```

converts the first character on a line into a '.', which very often is not what you intended.

As is true of many characters in *ed*, the '.' has several meanings, depending on its context. This line shows all three:

```
.s/././
```

The first '.' is a line number, the number of the line we are editing, which is called 'line dot'. (We will discuss line dot more in Section 3.) The second '.' is a metacharacter that matches any single character on that line. The third '.' is the only one that really is an honest literal period. On the *right* side of a substitution, '.' is not special. If you apply this command to the line

```
Now is the time.
```

the result will be

```
.ow is the time.
```

which is probably not what you intended.

The Backslash '\'

Since a period means 'any character', the question naturally arises of what to do when you really want a period. For example, how do you convert the line

```
Now is the time.
```

into

```
Now is the time?
```

The backslash '\' does the job. A backslash

turns off any special meaning that the next character might have; in particular, '\.' converts the '.' from a 'match anything' into a period, so you can use it to replace the period in

```
Now is the time.
```

like this:

```
s/\./?/
```

The pair of characters '\.' is considered by *ed* to be a single real period.

The backslash can also be used when searching for lines that contain a special character. Suppose you are looking for a line that contains

```
.PP
```

The search

```
/.PP/
```

isn't adequate, for it will find a line like

```
THE APPLICATION OF ...
```

because the '.' matches the letter 'A'. But if you say

```
\/.PP/
```

you will find only lines that contain '.PP'.

The backslash can also be used to turn off special meanings for characters other than '.'. For example, consider finding a line that contains a backslash. The search

```
\/
```

won't work, because the '\' isn't a literal '\', but instead means that the second '/' no longer delimits the search. But by preceding a backslash with another one, you can search for a literal backslash. Thus

```
\/\
```

does work. Similarly, you can search for a forward slash '/' with

```
\/
```

The backslash turns off the meaning of the immediately following '/' so that it doesn't terminate the /.../ construction prematurely.

As an exercise, before reading further, find two substitute commands each of which will convert the line

```
\x\y
```

into the line

```
\x/y
```

Here are several solutions; verify that each works as advertised.

```
s/\.\./
s/x./x/
s/..y/y/
```

A couple of miscellaneous notes about backslashes and special characters. First, you can use any character to delimit the pieces of an `s` command: there is nothing sacred about slashes. (But you must use slashes for context searching.) For instance, in a line that contains a lot of slashes already, like

```
//exec //sys.fort.go // etc. ...
```

you could use a colon as the delimiter — to delete all the slashes, type

```
s/::g
```

Second, if `#` and `@` are your character erase and line kill characters, you have to type `\#` and `\@`; this is true whether you're talking to `ed` or any other program.

When you are adding text with `a` or `i` or `c`, backslash is not special, and you should only put in one backslash for each one you really want.

The Dollar Sign '\$'

The next metacharacter, the '\$', stands for 'the end of the line'. As its most obvious use, suppose you have the line

```
Now is the
```

and you wish to add the word 'time' to the end. Use the \$ like this:

```
s/$/□time/
```

to get

```
Now is the time
```

Notice that a space is needed before 'time' in the substitute command, or you will get

```
Now is thetime
```

As another example, replace the second comma in the following line with a period without altering the first:

```
Now is the time, for all good men,
```

The command needed is

```
s/,,$/./
```

The \$ sign here provides context to make specific which comma we mean. Without it, of course, the `s` command would operate on the first comma to produce

```
Now is the time. for all good men,
```

As another example, to convert

```
Now is the time.
```

```
into
```

```
Now is the time?
```

as we did earlier, we can use

```
s/.$/?/
```

Like '.', the '\$' has multiple meanings depending on context. In the line

```
$s/$$/
```

the first '\$' refers to the last line of the file, the second refers to the end of that line, and the third is a literal dollar sign, to be added to that line.

The Circumflex '^'

The circumflex (or hat or caret) '^' stands for the beginning of the line. For example, suppose you are looking for a line that begins with 'the'. If you simply say

```
/the/
```

you will in all likelihood find several lines that contain 'the' in the middle before arriving at the one you want. But with

```
/^the/
```

you narrow the context, and thus arrive at the desired one more easily.

The other use of '^' is of course to enable you to insert something at the beginning of a line:

```
s/^/□/
```

places a space at the beginning of the current line.

Metacharacters can be combined. To search for a line that contains *only* the characters

```
.PP
```

you can use the command

```
/^\.PP$/
```

The Star '*'

Suppose you have a line that looks like this:

```
text x          y text
```

where *text* stands for lots of text, and there are some indeterminate number of spaces between the *x* and the *y*. Suppose the job is to replace all the spaces between *x* and *y* by a single space. The line is too long to retype, and there are too many spaces to count. What now?

This is where the metacharacter '*' comes in handy. A character followed by a star stands for as many consecutive occurrences of that character as possible. To refer to all the spaces at once, say

```
s/x□*y/x□y/
```

The construction '□*' means 'as many spaces as possible'. Thus 'x□*y' means 'an x, as many spaces as possible, then a y'.

The star can be used with any character, not just space. If the original example was instead

```
text x-----y text
```

then all '-' signs can be replaced by a single space with the command

```
s/x-*y/x□y/
```

Finally, suppose that the line was

```
text x.....y text
```

Can you see what trap lies in wait for the unwary? If you blindly type

```
s/x.*y/x□y/
```

what will happen? The answer, naturally, is that it depends. If there are no other x's or y's on the line, then everything works, but it's blind luck, not good management. Remember that '.' matches *any* single character? Then '.'* matches as many single characters as possible, and unless you're careful, it can eat up a lot more of the line than you expected. If the line was, for example, like this:

```
text x text x.....y text y text
```

then saying

```
s/x.*y/x□y/
```

will take everything from the *first* 'x' to the *last* 'y', which, in this example, is undoubtedly more than you wanted.

The solution, of course, is to turn off the special meaning of '.' with '\.':

```
s/x\. *y/x□y/
```

Now everything works, for '\.*' means 'as many *periods* as possible'.

There are times when the pattern '.'* is exactly what you want. For example, to change

```
Now is the time for all good men ...
```

into

```
Now is the time.
```

use '.'* to eat up everything after the 'for':

```
s/□for.*./
```

There are a couple of additional pitfalls associated with '*' that you should be aware of. Most notable is the fact that 'as many as possible' means *zero* or more. The fact that zero is a legitimate possibility is sometimes rather surprising. For example, if our line contained

```
text xy text x      y text
```

and we said

```
s/x□*y/x□y/
```

the *first* 'xy' matches this pattern, for it consists of an 'x', zero spaces, and a 'y'. The result is that the substitute acts on the first 'xy', and does not touch the later one that actually contains some intervening spaces.

The way around this, if it matters, is to specify a pattern like

```
/x□□*y/
```

which says 'an x, a space, then as many more spaces as possible, then a y', in other words, one or more spaces.

The other startling behavior of '*' is again related to the fact that zero is a legitimate number of occurrences of something followed by a star. The command

```
s/x*/y/g
```

when applied to the line

```
abcdef
```

produces

```
yaybycydyeyfy
```

which is almost certainly not what was intended. The reason for this behavior is that zero is a legal number of matches, and there are no x's at the beginning of the line (so that gets converted into a 'y'), nor between the 'a' and the 'b' (so that gets converted into a 'y'), nor ... and so on. Make sure you really want zero matches; if not, in this case write

```
s/xx*/y/g
```

'xx*' is one or more x's.

The Brackets '[']'

Suppose that you want to delete any numbers that appear at the beginning of all lines of a file. You might first think of trying a series of commands like

```
1,$s/^1*//
```

```
1,$s/^2*//
```

```
1,$s/^3*//
```

and so on, but this is clearly going to take forever if the numbers are at all long. Unless you want to repeat the commands over and over until finally all numbers are gone, you must get all the digits on one pass. This is the purpose of the brackets [and].

The construction

```
[0123456789]
```

matches any single digit — the whole thing is called a ‘character class’. With a character class, the job is easy. The pattern ‘[0123456789]*’ matches zero or more digits (an entire number), so

```
1,$s/^ [0123456789]*//
```

deletes all digits from the beginning of all lines.

Any characters can appear within a character class, and just to confuse the issue there are essentially no special characters inside the brackets; even the backslash doesn’t have a special meaning. To search for special characters, for example, you can say

```
/[\.\$^ \]/
```

Within [...], the ‘[’ is not special. To get a ‘[’ into a character class, make it the first character.

It’s a nuisance to have to spell out the digits, so you can abbreviate them as [0–9]; similarly, [a–z] stands for the lower case letters, and [A–Z] for upper case.

As a final frill on character classes, you can specify a class that means ‘none of the following characters’. This is done by beginning the class with a ‘^’:

```
[^0–9]
```

stands for ‘any character *except* a digit’. Thus you might find the first line that doesn’t begin with a tab or space by a search like

```
/^[^(space)(tab)]/
```

Within a character class, the circumflex has a special meaning only if it occurs at the beginning. Just to convince yourself, verify that

```
/^[^ ^]/
```

finds a line that doesn’t begin with a circumflex.

The Ampersand ‘&’

The ampersand ‘&’ is used primarily to save typing. Suppose you have the line

```
Now is the time
```

and you want to make it

```
Now is the best time
```

Of course you can always say

```
s/the/the best/
```

but it seems silly to have to repeat the ‘the’. The ‘&’ is used to eliminate the repetition. On the *right* side of a substitute, the ampersand means ‘whatever was just matched’, so you can say

```
s/the/& best/
```

and the ‘&’ will stand for ‘the’. Of course this isn’t much of a saving if the thing matched is just ‘the’, but if it is something truly long or awful, or if it is something like ‘.*’ which matches a lot of text, you can save some tedious typing. There is also much less chance of making a typing error in the replacement text. For example, to parenthesize a line, regardless of its length,

```
s/.*/(&)/
```

The ampersand can occur more than once on the right side:

```
s/the/& best and & worst/
```

makes

```
Now is the best and the worst time
```

and

```
s/.*/&? &!!/
```

converts the original line into

```
Now is the time? Now is the time!!
```

To get a literal ampersand, naturally the backslash is used to turn off the special meaning:

```
s/ampersand/\&/
```

converts the word into the symbol. Notice that ‘&’ is not special on the left side of a substitute, only on the *right* side.

Substituting New-lines

ed provides a facility for splitting a single line into two or more shorter lines by ‘substituting in a new-line’. As the simplest example, suppose a line has gotten unmanageably long because of editing (or merely because it was unwisely typed). If it looks like

```
text xy text
```

you can break it between the ‘x’ and the ‘y’ like this:

```
s/xy/x\  
y/
```

This is actually a single command, although it is typed on two lines. Bearing in mind that ‘\’

turns off special meanings, it seems relatively intuitive that a '\ ' at the end of a line would make the new-line there no longer special.

You can in fact make a single line into several lines with this same mechanism. As a large example, consider underlining the word 'very' in a long line by splitting 'very' onto a separate line, and preceding it by the `nroff` formatting command '.ul'.

```
text a very big text
```

The command

```
s/□very□/\
.ul\
very\
/
```

converts the line into four shorter lines, preceding the word 'very' by the line '.ul', and eliminating the spaces around the 'very', all at the same time.

When a new-line is substituted in, dot is left pointing at the last line created.

Joining Lines

Lines may also be joined together, but this is done with the `j` command instead of `s`. Given the lines

```
Now is
□the time
```

and supposing that dot is set to the first of them, then the command

```
j
```

joins them together. No blanks are added, which is why we carefully showed a blank at the beginning of the second line.

All by itself, a `j` command joins line dot to line dot+1, but any contiguous set of lines can be joined. Just specify the starting and ending line numbers. For example,

```
1,$jp
```

joins all the lines into one big one and prints it. (More on line numbers in Section 3.)

Rearranging a Line with \ (... \)

(This section should be skipped on first reading.) Recall that '&' is a shorthand that stands for whatever was matched by the left side of an `s` command. In much the same way you can capture separate pieces of what was matched; the only difference is that you have to specify on the left side just what pieces you're interested in.

Suppose, for instance, that you have a file of lines that consist of names in the form

```
Smith, A. B.
Jones, C.
```

and so on, and you want the initials to precede the name, as in

```
A. B. Smith
C. Jones
```

It is possible to do this with a series of editing commands, but it is tedious and error-prone. (It is instructive to figure out how it is done, though.)

The alternative is to 'tag' the pieces of the pattern (in this case, the last name, and the initials), and then rearrange the pieces. On the left side of a substitution, if part of the pattern is enclosed between \ (and \), whatever matched that part is remembered, and available for use on the right side. On the right side, the symbol '\1' refers to whatever matched the first \ (... \) pair, '\2' to the second \ (... \), and so on.

The command

```
1,$s/^\([^,]*\),□*(.*)/2□\1/
```

although hard to read, does the job. The first \ (... \) matches the last name, which is any string up to the comma; this is referred to on the right side with '\1'. The second \ (... \) is whatever follows the comma and any spaces, and is referred to as '\2'.

Of course, with any editing sequence this complicated, it's foolhardy to simply run it and hope. The global commands `g` and `v` discussed in section 4 provide a way for you to print exactly those lines which were affected by the substitute command, and thus verify that it did what you wanted in all cases.

3. LINE ADDRESSING IN THE EDITOR

The next general area we will discuss is that of line addressing in `ed`, that is, how you specify what lines are to be affected by editing commands. We have already used constructions like

```
1,$s/x/y/
```

to specify a change on all lines. And most users are long since familiar with using a single new-line (or return) to print the next line, and with

```
/thing/
```

to find a line that contains 'thing'. Less familiar, surprisingly enough, is the use of

```
?thing?
```

to scan *backwards* for the previous occurrence of 'thing'. This is especially handy when you realize that the thing you want to operate on is back

up the page from where you are currently editing.

The slash and question mark are the only characters you can use to delimit a context search, though you can use essentially any character in a substitute command.

Address Arithmetic

The next step is to combine the line numbers like '\$', '/', '...' and '?...?' with '+' and '-'. Thus

`$-1`

is a command to print the next to last line of the current file (that is, one line before line '\$'). For example, to recall how far you got in a previous editing session,

`$-5,$p`

prints the last six lines. (Be sure you understand why it's six, not five.) If there aren't six, of course, you'll get an error message.

As another example,

`.-3,+.3p`

prints from three lines before where you are now (at line dot) to three lines after, thus giving you a bit of context. By the way, the '+' can be omitted:

`.-3,.3p`

is absolutely identical in meaning.

Another area in which you can save typing effort in specifying lines is to use '-' and '+' as line numbers by themselves.

—

by itself is a command to move back up one line in the file. In fact, you can string several minus signs together to move back up that many lines:

moves up three lines, as does '-3'. Thus

`-3,+3p`

is also identical to the examples above.

Since '-' is shorter than '-1', constructions like

`-.s/bad/good/`

are useful. This changes 'bad' to 'good' on the previous line and on the current line.

'+' and '-' can be used in combination with searches using '/.../' and '?...?', and with '\$'. The search

`/thing/--`

finds the line containing 'thing', and positions you two lines before it.

Repeated Searches

Suppose you ask for the search

`/horrible thing/`

and when the line is printed you discover that it isn't the horrible thing that you wanted, so it is necessary to repeat the search again. You don't have to re-type the search, for the construction

`//`

is a shorthand for 'the previous thing that was searched for', whatever it was. This can be repeated as many times as necessary. You can also go backwards:

`??`

searches for the same thing, but in the reverse direction.

Not only can you repeat the search, but you can use '/' as the left side of a substitute command, to mean 'the most recent pattern'.

`/horrible thing/`

`... ed prints line with 'horrible thing' ...`

`s//good/p`

To go backwards and change a line, say

`??s//good/`

Of course, you can still use the '&' on the right hand side of a substitute to stand for whatever got matched:

`//s//&&&/p`

finds the next occurrence of whatever you searched for last, replaces it by two copies of itself, then prints the line just to verify that it worked.

Default Line Numbers and the Value of Dot

One of the most effective ways to speed up your editing is always to know what lines will be affected by a command if you don't specify the lines it is to act on, and on what line you will be positioned (i.e., the value of dot) when a command finishes. If you can edit without specifying unnecessary line numbers, you can save a lot of typing.

As the most obvious example, if you issue a search command like

`/thing/`

you are left pointing at the next line that contains 'thing'. Then no address is required with commands like `s` to make a substitution on that line, or `p` to print it, or `l` to list it, or `d` to delete

it, or **a** to append text after it, or **c** to change it, or **i** to insert text before it.

What happens if there was no 'thing'? Then you are left right where you were — dot is unchanged. This is also true if you were sitting on the only 'thing' when you issued the command. The same rules hold for searches that use '?...?'; the only difference is the direction in which you search.

The delete command **d** leaves dot pointing at the line that followed the last deleted line. When line '\$' gets deleted, however, dot points at the *new* line '\$'.

The line-changing commands **a**, **c** and **i** by default all affect the current line — if you give no line number with them, **a** appends text after the current line, **c** changes the current line, and **i** inserts text before the current line.

a, **c**, and **i** behave identically in one respect — when you stop appending, changing or inserting, dot points at the last line entered. This is exactly what you want for typing and editing on the fly. For example, you can say

```
a
... text ...
... botch ...           (minor error)
.
s/botch/correct/       (fix botched line)
a
... more text ...
```

without specifying any line number for the substitute command or for the second append command. Or you can say

```
a
... text ...
... horrible botch ...  (major error)
.
c                       (replace entire line)
... fixed up line ...
```

You should experiment to determine what happens if you add *no* lines with **a**, **c** or **i**.

The **r** command will read a file into the text being edited, either at the end if you give no address, or after the specified line if you do. In either case, dot points at the last line read in. Remember that you can even say **Or** to read a file in at the beginning of the text. (You can also say **0a** or **1i** to start adding text at the beginning.)

The **w** command writes out the entire file. If you precede the command by one line number, that line is written, while if you precede it by two line numbers, that range of lines is written. The **w** command does *not* change dot: the current line remains the same, regardless of

what lines are written. This is true even if you say something like

```
/^\.AB/,/^\.AE/w abstract
```

which involves a context search.

Since the **w** command is so easy to use, you should save what you are editing regularly as you go along just in case the system crashes, or in case you do something foolish, like clobbering what you're editing.

The least intuitive behavior, in a sense, is that of the **s** command. The rule is simple — you are left sitting on the last line that got changed. If there were no changes, then dot is unchanged.

To illustrate, suppose that there are three lines in the buffer, and you are sitting on the middle one:

```
x1
x2
x3
```

Then the command

```
-,+s/x/y/p
```

prints the third line, which is the last one changed. But if the three lines had been

```
x1
y2
y3
```

and the same command had been issued while dot pointed at the second line, then the result would be to change and print only the first line, and that is where dot would be set.

Semicolon ';'

Searches with '/.../' and '?...?' start at the current line and move forward or backward respectively until they either find the pattern or get back to the current line. Sometimes this is not what is wanted. Suppose, for example, that the buffer contains lines like this:

```
:
:
ab
:
bc
:
```

Starting at line 1, one would expect that the command

```
/a/,b/p
```

prints all the lines from the 'ab' to the 'bc' inclusive. Actually this is not what happens. *Both* searches (for 'a' and for 'b') start from the same point, and thus they both find the line that

contains 'ab'. The result is to print a single line. Worse, if there had been a line with a 'b' in it before the 'ab' line, then the print command would be in error, since the second line number would be less than the first, and it is illegal to try to print lines in reverse order.

This is because the comma separator for line numbers doesn't set dot as each address is processed; each search starts from the same place. In `ed`, the semicolon ';' can be used just like comma, with the single difference that use of a semicolon forces dot to be set at that point as the line numbers are being evaluated. In effect, the semicolon 'moves' dot. Thus in our example above, the command

```
/a;/b/p
```

prints the range of lines from 'ab' to 'bc', because after the 'a' is found, dot is set to that line, and then 'b' is searched for, starting beyond that line.

This property is most often useful in a very simple situation. Suppose you want to find the *second* occurrence of 'thing'. You could say

```
/thing/  
//
```

but this prints the first occurrence as well as the second, and is a nuisance when you know very well that it is only the second one you're interested in. The solution is to say

```
/thing/;/
```

This says to find the first occurrence of 'thing', set dot to that line, then find the second and print only that.

Closely related is searching for the second previous occurrence of something, as in

```
?something?;??
```

Printing the third or fourth or ... in either direction is left as an exercise.

Finally, bear in mind that if you want to find the first occurrence of something in a file, starting at an arbitrary place within the file, it is not sufficient to say

```
1;/thing/
```

because this fails if 'thing' occurs on line 1. But it is possible to say

```
0;/thing/
```

(one of the few places where 0 is a legal line number), for this starts the search at line 1.

Interrupting the Editor

As a final note on what dot gets set to, you should be aware that if you hit the interrupt or delete or rubout or break key while `ed` is doing a command, things are put back together again and your state is restored as much as possible to what it was before the command began. Naturally, some changes are irrevocable — if you are reading or writing a file or making substitutions or deleting lines, these will be stopped in some clean but unpredictable state in the middle (which is why it is not usually wise to stop them). Dot may or may not be changed.

Printing is more clear cut. Dot is not changed until the printing is done. Thus if you print until you see an interesting line, then hit delete, you are *not* sitting on that line or even near it. Dot is left where it was when the `p` command was started.

4. GLOBAL COMMANDS

The global commands `g` and `v` are used to perform one or more editing commands on all lines that either contain (`g`) or don't contain (`v`) a specified pattern.

As the simplest example, the command

```
g/UNIX/p
```

prints all lines that contain the word 'UNIX'. The pattern that goes between the slashes can be anything that could be used in a line search or in a substitute command; exactly the same rules and limitations apply.

As another example, then,

```
g/^\. /p
```

prints all the formatting commands in a file (lines that begin with '.').

The `v` command is identical to `g`, except that it operates on those line that do *not* contain an occurrence of the pattern. (Don't look too hard for mnemonic significance to the letter 'v'.) So

```
v/^\. /p
```

prints all the lines that don't begin with '.' — the actual text lines.

The command that follows `g` or `v` can be anything:

```
g/^\. /d
```

deletes all lines that begin with '.', and

```
g/^ $/d
```

deletes all empty lines.

Probably the most useful command that can follow a global is the substitute command.

for this can be used to make a change and print each affected line for verification. For example, we could change the word 'Unix' to 'UNIX' everywhere, and verify that it really worked, with

```
g/Unix/s//UNIX/gp
```

Notice that we used '/' in the substitute command to mean 'the previous pattern', in this case, 'Unix'. The **p** command is done on every line that matches the pattern, not just those on which a substitution took place.

The global command operates by making two passes over the file. On the first pass, all lines that match the pattern are marked. On the second pass, each marked line in turn is examined, dot is set to that line, and the command executed. This means that it is possible for the command that follows a **g** or **v** to use addresses, set dot, and so on, quite freely.

```
g/^\.PP/+
```

prints the line that follows each 'PP' command (the signal for a new paragraph in some formatting packages). Remember that '+' means 'one line past dot'. And

```
g/topic/?^\.SH?1
```

searches for each line that contains 'topic', scans backwards until it finds a line that begins 'SH' (a section heading) and prints the line that follows that, thus showing the section headings under which 'topic' is mentioned. Finally,

```
g/^\.EQ/+,/^\.EN/-p
```

prints all the lines that lie between lines beginning with 'EQ' and 'EN' formatting commands.

The **g** and **v** commands can also be preceded by line numbers, in which case the lines searched are only those in the range specified.

Multi-line Global Commands

It is possible to do more than one command under the control of a global command, although the syntax for expressing the operation is not especially natural or pleasant. As an example, suppose the task is to change 'x' to 'y' and 'a' to 'b' on all lines that contain 'thing'. Then

```
g/thing/s/x/y\  
s/a/b/
```

is sufficient. The '\' signals the **g** command that the set of commands continues on the next line; it terminates on the first line that does not end with '\'. (As a minor blemish, you can't use a substitute command to insert a new-line within a **g** command.)

You should watch out for this problem: the command

```
g/x/s//y\  
s/a/b/
```

does *not* work as you expect. The remembered pattern is the last pattern that was actually executed, so sometimes it will be 'x' (as expected), and sometimes it will be 'a' (not expected). You must spell it out, like this:

```
g/x/s/x/y\  
s/a/b/
```

It is also possible to execute **a**, **c** and **i** commands under a global command; as with other multi-line constructions, all that is needed is to add a '\' at the end of each line except the last. Thus to add a '.nf' and '.sp' command before each '.EQ' line, type

```
g/^\.EQ/i\  
.nf\  
.sp
```

There is no need for a final line containing a '.' to terminate the **i** command, unless there are further commands being done under the global. On the other hand, it does no harm to put it in either.

5. CUT AND PASTE WITH UNIX COMMANDS

One editing area in which non-programmers seem not very confident is in what might be called 'cut and paste' operations — changing the name of a file, making a copy of a file somewhere else, moving a few lines from one place to another in a file, inserting one file in the middle of another, splitting a file into pieces, and splicing two or more files together.

Yet most of these operations are actually quite easy, if you keep your wits about you and go cautiously. The next several sections talk about cut and paste. We will begin with the UNIX commands for moving entire files around, then discuss **ed** commands for operating on pieces of files.

Changing the Name of a File

You have a file named 'memo' and you want it to be called 'paper' instead. How is it done?

The UNIX program that renames files is called **mv** (for 'move'); it 'moves' the file from one name to another, like this:

```
mv memo paper
```

That's all there is to it: **mv** from the old name to the new name.

```
mv old_name new_name
```

Warning: if there is already a file around with the new name, its present contents will be silently clobbered by the information from the other file. The one exception is that you can't move a file to itself —

```
mv x x
```

is illegal.

Making a Copy of a File

Sometimes what you want is a copy of a file — an entirely fresh version. This might be because you want to work on a file, and yet save a copy in case something gets fouled up, or just because you're paranoid.

In any case, the way to do it is with the **cp** command. (**cp** stands for 'copy'; the UNIX system is big on short command names, which are appreciated by heavy users, but sometimes a strain for novices.) Suppose you have a file called 'good' and you want to save a copy before you make some dramatic editing changes. Choose a name — 'save_good' might be acceptable — then type

```
cp good save_good
```

This copies 'good' onto 'save_good', and you now have two identical copies of the file 'good'. (If 'save_good' previously contained something, it gets overwritten.)

Now if you decide at some time that you want to get back to the original state of 'good', you can say

```
mv save_good good
```

(if you're not interested in 'save_good' any more), or

```
cp save_good good
```

if you still want to retain a safe copy.

In summary, **mv** just renames a file; **cp** makes a duplicate copy. Both of them clobber the 'target' file if it already exists, so you had better be sure that's what you want to do *before* you do it.

Removing a File

If you decide you are really done with a file forever, you can remove it with the **rm** command:

```
rm save_good
```

throws away (irrevocably) the file called 'save_good'.

Putting Two or More Files Together

The next step is the familiar one of collecting two or more files into one big one. This will be needed, for example, when the author of a paper decides that several sections need to be combined into one. There are several ways to do it, of which the cleanest, once you get used to it, is a program called **cat**. (Not *all* UNIX programs have two-letter names.) **cat** is short for 'concatenate', which is exactly what we want to do.

Suppose the job is to combine the files 'file1' and 'file2' into a single file called 'big_file'. If you say

```
cat file
```

the contents of 'file' will get printed on your terminal. If you say

```
cat file1 file2
```

the contents of 'file1' and then the contents of 'file2' will *both* be printed on your terminal, in that order. So **cat** combines the files, all right, but it's not much help to print them on the terminal — we want them in 'big_file'.

Fortunately, there is a way. You can tell the system that instead of printing on your terminal, you want the same information put in a file. The way to do it is to add to the command line the character **>** and the name of the file where you want the output to go. Then you can say

```
cat file1 file2 >big_file
```

and the job is done. (As with **cp** and **mv**, you're putting something into 'big_file', and anything that was already there is destroyed.)

This ability to 'capture' the output of a program is one of the most useful aspects of the UNIX system. Fortunately it's not limited to the **cat** program — you can use it with *any* program that prints on your terminal. We'll see some more uses for it in a moment.

Naturally, you can combine several files, not just two:

```
cat file1 file2 file3 ... >big_file
```

collects a whole bunch.

Question: is there any difference between

```
cp good save_good
```

and

```
cat good >save_good
```

Answer: for most purposes, no. You might reasonably ask why there are two programs in that case, since **cat** is obviously all you need. The answer is that **cp** will do some other things as

well, which you can investigate for yourself by reading the manual. For now we'll stick to simple usages.

Adding Something to the End of a File

Sometimes you want to add one file to the end of another. We have enough building blocks now that you can do it; in fact before reading further it would be valuable if you figured out how. To be specific, how would you use **cp**, **mv** and/or **cat** to add the file 'good1' to the end of the file 'good'?

You could try

```
cat good good1 >temp
mv temp good
```

which is probably most direct. You should also understand why

```
cat good good1 >good
```

doesn't work. (Don't practice with a good 'good'!)

The easy way is to use a variant of **>**, called **>>**. In fact, **>>** is identical to **>** except that instead of clobbering the old file, it simply tacks stuff on at the end. Thus you could say

```
cat good1 >>good
```

and 'good1' is added to the end of 'good'. (And if 'good' didn't exist, this makes a copy of 'good1' called 'good'.)

6. CUT AND PASTE WITH THE EDITOR

Now we move on to manipulating pieces of files — individual lines or groups of lines. This is another area where new users seem unsure of themselves.

File Names

The first step is to ensure that you know the **ed** commands for reading and writing files. Of course you can't go very far without knowing **r** and **w**. Equally useful, but less well known, is the 'edit' command **e**. Within **ed**, the command

```
e new_file
```

says 'I want to edit a new file called *new_file*, without leaving the editor.' The **e** command discards whatever you're currently working on and starts over on *new_file*. It's exactly the same as if you had quit with the **q** command, then re-entered **ed** with a new file name, except that if you have a pattern remembered, then a command like **//** will still work.

If you enter **ed** with the command

```
ed file
```

ed remembers the name of the file, and any subsequent **e**, **r** or **w** commands that don't contain a file name will refer to this remembered file. Thus

```
ed file1
... (editing) ...
w      (writes back in file1)
e file2 (edit new file, without leaving editor)
... (editing on file2) ...
w      (writes back on file2)
```

(and so on) does a series of edits on various files without ever leaving **ed** and without typing the name of any file more than once. (As an aside, if you examine the sequence of commands here, you can see why many UNIX systems use **e** as a synonym for **ed**.)

You can find out the remembered file name at any time with the **f** command; just type **f** without a file name. You can also change the name of the remembered file name with **f**; a useful sequence is

```
ed precious
f junk
... (editing) ...
```

which gets a copy of a precious file, then uses **f** to guarantee that a careless **w** command won't clobber the original.

Inserting One File into Another

Suppose you have a file called 'memo', and you want the file called 'table' to be inserted just after the reference to Table 1. That is, in 'memo' somewhere is a line that says

Table 1 shows that ...

and the data contained in 'table' has to go there, probably so it will be formatted properly by **nroff** or **troff**. Now what?

This one is easy. Edit 'memo', find 'Table 1', and add the file 'table' right there:

```
ed memo
/Table 1/
Table 1 shows that ... [response from ed]
.r table
```

The critical line is the last one. As we said earlier, the **r** command reads a file; here you asked for it to be read in right after line dot. An **r** command without any address adds lines at the end, so it is the same as **Sr**.

Writing out Part of a File

The other side of the coin is writing out part of the document you're editing. For example, maybe you want to split out into a separate file that table from the previous example, so it can be formatted and tested separately. Suppose that in the file being edited we have

```
.TS
... [lots of stuff]
.TE
```

which is the way a table is set up for the `tbl` program. To isolate the table in a separate file called 'table', first find the start of the table (the '.TS' line), then write out the interesting part:

```
/^\.TS/
.TS [ed prints the line it found]
./^\.TE/w table
```

and the job is done. If you are confident, you can do it all at once with

```
/^\.TS/;/^\.TE/w table
```

The point is that the `w` command can write out a group of lines, instead of the whole file. In fact, you can write out a single line if you like; just give one line number instead of two. For example, if you have just typed a horribly complicated line and you know that it (or something like it) is going to be needed later, then save it — don't re-type it. In the editor, say

```
a
... lots of stuff ...
... horrible line ...
.
.w temp
a
... more stuff ...
.
.r temp
a
... more stuff ...
.
```

This last example is worth studying, to be sure you appreciate what's going on.

Moving Lines Around

Suppose you want to move a paragraph from its present position in a paper to the end. How would you do it? As a concrete example, suppose each paragraph in the paper begins with the formatting command '.PP'. Think about it and write down the details before reading on.

The brute force way (not necessarily bad) is to write the paragraph onto a temporary file, delete it from its current position, then read in the temporary file at the end. Assuming that

you are sitting on the '.PP' command that begins the paragraph, this is the sequence of commands:

```
./^\.PP/-w temp
./-/d
$r temp
```

That is, from where you are now ('.') until one line before the next '.PP' ('/^\.PP/-') write onto 'temp'. Then delete the same lines. Finally, read 'temp' at the end.

As we said, that's the brute force way. The easier way (often) is to use the `move` command `m` that `ed` provides — it lets you do the whole set of operations at one crack, without any temporary file.

The `m` command is like many other `ed` commands in that it takes up to two line numbers in front that tell what lines are to be affected. It is also *followed* by a line number that tells where the lines are to go. Thus

```
line1, line2 m line3
```

says to move all the lines between 'line1' and 'line2' after 'line3'. Naturally, any of 'line1' etc., can be patterns between slashes, \$ signs, or other ways to specify lines.

Suppose again that you're sitting at the first line of the paragraph. Then you can say

```
./^\.PP/-m$
```

That's all.

As another example of a frequent operation, you can reverse the order of two adjacent lines by moving the first one to after the second. Suppose that you are positioned at the first. Then

```
m+
```

does it. It says to move line dot to after one line after line dot. If you are positioned on the second line,

```
m--
```

does the interchange.

As you can see, the `m` command is more succinct and direct than writing, deleting and re-reading. When is brute force better anyway? This is a matter of personal taste — do what you have most confidence in. The main difficulty with the `m` command is that if you use patterns to specify both the lines you are moving and the target, you have to take care that you specify them properly, or you may well not move the lines you thought you did. The result of a botched `m` command can be a ghastly mess. Doing the job a step at a time makes it easier for you to verify at each step that you accomplished what you wanted to. It's also a good idea to

issue a **w** command before doing anything complicated; then if you goof, it's easy to back up to where you were.

Marks

ed provides a facility for marking a line with a particular name so you can later reference it by name regardless of its actual line number. This can be handy for moving lines, and for keeping track of them as they move. The *mark* command is **k**; the command

```
kx
```

marks the current line with the name 'x'. If a line number precedes the **k**, that line is marked. (The mark name must be a single lower case letter.) Now you can refer to the marked line with the address

```
'x
```

Marks are most useful for moving things around. Find the first line of the block to be moved, and mark it with '*a*'. Then find the last line and mark it with '*b*'. Now position yourself at the place where the stuff is to go and say

```
'a,'bm.
```

Bear in mind that only one line can have a particular mark name associated with it at any given time.

Copying Lines

We mentioned earlier the idea of saving a line that was hard to type or used often, so as to cut down on typing time. Of course this could be more than one line; then the saving is presumably even greater.

ed provides another command, called **t** (for 'transfer') for making a copy of a group of one or more lines at any point. This is often easier than writing and reading.

The **t** command is identical to the **m** command, except that instead of moving lines it simply duplicates them at the place you named. Thus

```
1,$t$
```

duplicates the entire contents that you are editing. A more common use for **t** is for creating a series of lines that differ only slightly. For example, you can say

```
a
..... x ..... (long line)
.
t.           (make a copy)
s/x/y/      (change it a bit)
t.           (make third copy)
s/y/z/      (change it a bit)
```

and so on.

The Temporary Escape '!'

Sometimes it is convenient to be able to temporarily escape from the editor to do some other UNIX command, perhaps one of the file copy or move commands discussed in section 5, without leaving the editor. The 'escape' command **!** provides a way to do this. If you say

```
!any UNIX command
```

your current editing state is suspended, and the UNIX command you asked for is executed. When the command finishes, **ed** will signal you by printing another **!**; at that point you can resume editing.

You can really do *any* UNIX command, including another **ed**. (This is quite common, in fact.) In this case, you can even do another **!**.

7. SUPPORTING TOOLS

There are several tools and techniques that go along with the editor, all of which are relatively easy once you know how **ed** works, because they are all based on the editor. In this section we will give some fairly cursory examples of these tools, more to indicate their existence than to provide a complete tutorial. More information on each can be found in [3].

Grep

Sometimes you want to find all occurrences of some word or pattern in a set of files, to edit them or perhaps just to verify their presence or absence. It may be possible to edit each file separately and look for the pattern of interest, but if there are many files this can get very tedious, and if the files are really big, it may be impossible because of limits in **ed**.

The program **grep** was invented to get around these limitations. The search patterns that we have described in the paper are often called 'regular expressions', and 'grep' stands for

```
g/re/p
```

That describes exactly what **grep** does — it prints every line in a set of files that contains a particular pattern. Thus

```
grep 'thing' file1 file2 file3 ...
```

finds 'thing' wherever it occurs in any of the files 'file1', 'file2', etc. **grep** also indicates the file in which the line was found, so you can later edit it if you like.

The pattern represented by 'thing' can be any pattern you can use in the editor, since **grep** and **ed** use exactly the same mechanism for pattern searching. It is wisest always to enclose the pattern in the single quotes '...' if it contains any non-alphabetic characters, since many such characters also mean something special to the UNIX command interpreter (the 'shell'). If you don't quote them, the command interpreter will try to interpret them before **grep** gets a chance.

There is also a way to find lines that *don't* contain a pattern:

```
grep -v 'thing' file1 file2 ...
```

finds all lines that don't contain 'thing'. The **-v** must occur in the position shown. Given **grep** and **grep -v**, it is possible to do things like selecting all lines that contain some combination of patterns. For example, to get all lines that contain 'x' but not 'y':

```
grep x file ... | grep -v y
```

(The notation **|** is a 'pipe', which causes the output of the first command to be used as input to the second command; see [2].)

Editing Scripts

If a fairly complicated set of editing operations is to be done on a whole set of files, the easiest thing to do is to make up a 'script', i.e., a file that contains the operations you want to perform, then apply this script to each file in turn.

For example, suppose you want to change every 'Unix' to 'UNIX' and every 'Gcos' to 'GCOS' in a large number of files. Then put into the file 'script' the lines

```
g/Unix/s//UNIX/g
g/Gcos/s//GCOS/g
w
q
```

Now you can say

```
ed file1 <script
ed file2 <script
...
```

This causes **ed** to take its commands from the prepared script. Notice that the whole job has to be planned in advance.

And of course by using the UNIX command interpreter, you can cycle through a set of files automatically, with varying degrees of ease.

Sed

sed ('stream editor') is a version of the editor with restricted capabilities but which is capable of processing unlimited amounts of input. Basically **sed** copies its input to its output, applying one or more editing commands to each line of input.

As an example, suppose that we want to do the 'Unix' to 'UNIX' part of the example given above, but without rewriting the files. Then the command

```
sed 's/Unix/UNIX/g' file1 file2 ...
```

applies the command 's/Unix/UNIX/g' to all lines from 'file1', 'file2', etc., and copies all lines to the output. The advantage of using **sed** in such a case is that it can be used with input too large for **ed** to handle. All the output can be collected in one place, either in a file or perhaps piped into another program.

If the editing transformation is so complicated that more than one editing command is needed, commands can be supplied from a file, or on the command line, with a slightly more complex syntax. To take commands from a file, for example,

```
sed -f cmdfile input_files ...
```

sed has further capabilities, including conditional testing and branching, which we cannot go into here.

Acknowledgement

I am grateful to Ted Dolotta for his careful reading and valuable suggestions.

References

- [1] Brian W. Kernighan. *A Tutorial Introduction to the UNIX Text Editor*, Bell Laboratories.
- [2] Brian W. Kernighan. *UNIX For Beginners*, Bell Laboratories.
- [3] T. A. Dolotta, S. B. Olsson, and A. G. Petrucci (eds.). *UNIX User's Manual - Release 3.0*, Bell Laboratories (June 1980).

January 1981

SED—A Non-Interactive Text Editor

Lee E. McMahon

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Sed is a non-interactive context editor that runs on the UNIX[†] operating system. *Sed* is designed to be especially useful in three cases:

- 1) To edit files too large for comfortable interactive editing.
- 2) To edit any size file when the sequence of editing commands is too complicated to be comfortably typed in interactive mode.
- 3) To perform multiple 'global' editing functions efficiently in one pass through the input.

This memorandum constitutes a manual for users of *sed*.

INTRODUCTION

Sed is a non-interactive context editor designed to be especially useful in three cases:

- 1) To edit files too large for comfortable interactive editing.
- 2) To edit any size file when the sequence of editing commands is too complicated to be comfortably typed in interactive mode.
- 3) To perform multiple 'global' editing functions efficiently in one pass through the input.

Because only a few lines of the input reside in memory at one time, and no temporary files are used, the effective size of file that can be edited is limited only by the requirement that the input and output fit simultaneously into available secondary storage.

Complicated editing scripts can be created separately and given to *sed* as a command file. For complex edits, this saves considerable typing, and its attendant errors. *Sed* running from a command file is much more efficient than any interactive editor known to the author, even if that editor can be driven by a pre-written script.

The principal loss of functions compared to an interactive editor are lack of relative addressing (because of the line-at-a-time operation), and lack of immediate verification that a command has done what was intended.

Sed is a lineal descendant of the UNIX editor, *ed*. Because of the differences between interactive and non-interactive operation, considerable changes have been made between *ed* and *sed*; even confirmed users of *ed* will frequently be surprised (and probably chagrined), if they rashly use *sed* without reading Sections 2 and 3 of this document. The most striking family resemblance between the two editors is in the class of patterns ('regular expressions') they recognize; the code for matching patterns is copied almost verbatim from the code for *ed*, and the description of regular expressions in Section 2 is copied almost verbatim from the *UNIX Programmer's Manual* [1]. (Both code and description were written by Dennis M. Ritchie.)

[†] UNIX is a trademark of Bell Laboratories.

1. OVERALL OPERATION

Sed by default copies the standard input to the standard output, perhaps performing one or more editing commands on each line before writing it to the output. This behavior may be modified by flags on the command line; see Section 1.1 below.

The general format of an editing command is:

```
[address1,address2][function][arguments]
```

One or both addresses may be omitted; the format of addresses is given in Section 2. Any number of blanks or tabs may separate the addresses from the function. The function must be present; the available commands are discussed in Section 3. The arguments may be required or optional, according to which function is given; again, they are discussed in Section 3 under each individual function.

Tab characters and spaces at the beginning of lines are ignored.

1.1. Command-line Flags

Three flags are recognized on the command line:

- n tells *sed* not to copy all lines, but only those specified by *p* functions or *p* flags after *s* functions (see Section 3.3);
- e tells *sed* to take the next argument as an editing command;
- f tells *sed* to take the next argument as a file name; the file should contain editing commands, one to a line.

1.2. Order of Application of Editing Commands

Before any editing is done (in fact, before any input file is even opened), all the editing commands are compiled into a form which will be moderately efficient during the execution phase (when the commands are actually applied to lines of the input file). The commands are compiled in the order in which they are encountered; this is generally the order in which they will be attempted at execution time. The commands are applied one at a time; the input to each command is the output of all preceding commands.

The default linear order of application of editing commands can be changed by the flow-of-control commands, *t* and *b* (see Section 3). Even when the order of application is changed by these commands, it is still true that the input line to any command is the output of any previously applied command.

1.3. Pattern-space

The range of pattern matches is called the pattern space. Ordinarily, the pattern space is one line of the input text, but more than one line can be read into the pattern space by using the *N* command (Section 3.6.).

1.4. Examples

Examples are scattered throughout the text. Except where otherwise noted, the examples all assume the following input text:

```
In Xanadu did Kubla Khan
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless to man
Down to a sunless sea.
```

(In no case is the output of the *sed* commands to be considered an improvement on Coleridge.)

Example:

The command

```
2q
```

will quit after copying the first two lines of the input. The output will be:

```
In Xanadu did Kubla Khan
A stately pleasure dome decree:
```

2. ADDRESSES: SELECTING LINES FOR EDITING

Lines in the input file(s) to which editing commands are to be applied can be selected by addresses. Addresses may be either line numbers or context addresses.

The application of a group of commands can be controlled by one address (or address-pair) by grouping the commands with curly braces ('{')—see Section 3.6.).

2.1. Line-number Addresses

A line number is a decimal integer. As each line is read from the input, a line-number counter is incremented; a line-number address matches (selects) the input line which causes the internal counter to equal the address line-number. The counter runs cumulatively through multiple input files; it is not reset when a new input file is opened.

As a special case, the character \$ matches the last line of the last input file.

2.2. Context Addresses

A context address is a pattern ('regular expression') enclosed in slashes ('/'). The regular expressions recognized by *sed* are constructed as follows:

- 1) An ordinary character (not one of those discussed below) is a regular expression, and matches that character.
- 2) A circumflex '^' at the beginning of a regular expression matches the null character at the beginning of a line.
- 3) A dollar-sign '\$' at the end of a regular expression matches the null character at the end of a line.
- 4) The characters '\n' match an embedded new-line character, but not the new-line at the end of the pattern space.
- 5) A period '.' matches any character except the terminal new-line of the pattern space.
- 6) A regular expression followed by an asterisk '*' matches any number (including 0) of adjacent occurrences of the regular expression it follows.
- 7) A string of characters in square brackets '[' matches any character in the string, and no others. If, however, the first character of the string is circumflex '^', the regular expression matches any character *except* the characters in the string and the terminal new-line of the pattern space.
- 8) A concatenation of regular expressions is a regular expression which matches the concatenation of strings matched by the components of the regular expression.
- 9) A regular expression between the sequences '\(' and '\)' is identical in effect to the unadorned regular expression, but has side-effects which are described under the *s* command below and specification 10) immediately below.
- 10) The expression '\d' means the same string of characters matched by an expression enclosed in '\(' and '\)' earlier in the same pattern. Here *d* is a single digit; the string specified is that beginning with the *d*th occurrence of '\(' counting from the left. For example, the expression '^\(.*\)\1' matches a line beginning with two repeated occurrences of the same string.
- 11) The null regular expression standing alone (e.g., '/') is equivalent to the last regular expression compiled.

To use one of the special characters (^ \$. * [] \ /) as a literal (to match an occurrence of itself in the input), precede the special character by a backslash '\'.

For a context address to 'match' the input requires that the whole pattern within the address match some portion of the pattern space.

2.3. Number of Addresses

The commands in the next section can have 0, 1, or 2 addresses. Under each command the maximum number of allowed addresses is given. For a command to have more addresses than the maximum allowed is considered an error.

If a command has no addresses, it is applied to every line in the input.

If a command has one address, it is applied to all lines which match that address.

If a command has two addresses, it is applied to the first line which matches the first address, and to all subsequent lines until (and including) the first subsequent line which matches the second address. Then an attempt is made on subsequent lines to again match the first address, and the process is repeated.

Two addresses are separated by a comma.

Examples:

/an/	matches lines 1, 3, 4 in our sample text
/an.*an/	matches line 1
/^an/	matches no lines
./	matches all lines
^./	matches line 5
/r*an/	matches lines 1, 3, 4 (number = zero!)
^(an\).*\1/	matches line 1

3. FUNCTIONS

All functions are named by a single character. In the following summary, the maximum number of allowable addresses is given enclosed in parentheses, then the single character function name, possible arguments enclosed in angles (< >), an expanded English translation of the single-character name, and finally a description of what each function does. The angles around the arguments are *not* part of the argument, and should not be typed in actual editing commands.

3.1. Whole-line Oriented Functions

(2)d — delete lines

The *d* function deletes from the file (does not write to the output) all those lines matched by its address(es).

It also has the side effect that no further commands are attempted on the corpse of a deleted line; as soon as the *d* function is executed, a new line is read from the input, and the list of editing commands is re-started from the beginning on the new line.

(2)n — next line

The *n* function reads the next line from the input, replacing the current line. The current line is written to the output if it should be. The list of editing commands is continued following the *n* command.

(1)a <text> — append lines

The *a* function causes the argument <text> to be written to the output after

the line matched by its address. The *a* command is inherently multi-line; *a* must appear at the end of a line, and `<text>` may contain any number of lines. To preserve the one-command-to-a-line fiction, the interior new-lines must be hidden by a backslash character (`\`) immediately preceding the new-line. The `<text>` argument is terminated by the first unhidden new-line (the first one not immediately preceded by backslash).

Once an *a* function is successfully executed, `<text>` will be written to the output, regardless of what later commands do to the line which triggered it; that line may be deleted entirely; `<text>` will still be written to the output.

The `<text>` is not scanned for address matches, and no editing commands are attempted on it. It does not cause any change in the line-number counter.

(1)*i*\
`<text>` – insert lines

The *i* function behaves identically to the *a* function, except that `<text>` is written to the output *before* the matched line. All other comments about the *a* function apply to the *i* function as well.

(2)*c*\
`<text>` – change lines

The *c* function deletes the lines selected by its address(es), and replaces them with the lines in `<text>`. Like *a* and *i*, *c* must be followed by a new-line hidden by a backslash; and interior new lines in `<text>` must be hidden by backslashes.

The *c* command may have two addresses, and therefore select a range of lines. If it does, all the lines in the range are deleted, but only one copy of `<text>` is written to the output, *not* one copy per line deleted. As with *a* and *i*, `<text>` is not scanned for address matches, and no editing commands are attempted on it. It does not change the line-number counter.

After a line has been deleted by a *c* function, no further commands are attempted on the corpse.

If text is appended after a line by *a* or *r* functions, and the line is subsequently changed, the text inserted by the *c* function will be placed *before* the text of the *a* or *r* functions. (The *r* function is described in Section 3.4.)

Note: Within the text put in the output by these functions, leading blanks and tabs will disappear, as always in *sed* commands. To get leading blanks and tabs into the output, precede the first desired blank or tab by a backslash; the backslash will not appear in the output.

Example:

The list of editing commands:

```
n
a\
XXXX
d
```

applied to our standard input, produces:

```
In Xanadu did Kubla Khan
XXXX
Where Alph, the sacred river, ran
XXXX
Down to a sunless sea.
```

In this particular case, the same effect would be produced by either of the two following command lists:

```

n      n
i\    c\
XXXX  XXXX
d

```

3.2. Substitute Function

One very important function changes parts of lines selected by a context search within the line.

(2)s<pattern><replacement><flags> – substitute

The *s* function replaces *part* of a line (selected by <pattern>) with <replacement>. It can best be read:

Substitute for <pattern>, <replacement>

The <pattern> argument contains a pattern, exactly like the patterns in addresses (see Section 2.2 above). The only difference between <pattern> and a context address is that the context address must be delimited by slash ('/') characters; <pattern> may be delimited by any character other than space or new-line.

By default, only the first string matched by <pattern> is replaced, but see the *g* flag below.

The <replacement> argument begins immediately after the second delimiting character of <pattern>, and must be followed immediately by another instance of the delimiting character. (Thus there are exactly *three* instances of the delimiting character.)

The <replacement> is not a pattern, and the characters which are special in patterns do not have special meaning in <replacement>. Instead, other characters are special:

& is replaced by the string matched by <pattern>

d (where *d* is a single digit) is replaced by the *d*th substring matched by parts of <pattern> enclosed in '\(' and '\)'. If nested substrings occur in <pattern>, the *d*th is determined by counting opening delimiters ('\(').

As in patterns, special characters may be made literal by preceding them with backslash ('\).

The <flags> argument may contain the following flags:

g – substitute <replacement> for all (non-overlapping) instances of <pattern> in the line. After a successful substitution, the scan for the next instance of <pattern> begins just after the end of the inserted characters; characters put into the line from <replacement> are not rescanned.

p – print the line if a successful replacement was done. The *p* flag causes the line to be written to the output if and only if a substitution was actually made by the *s* function. Notice that if several *s* functions, each followed by a *p* flag, successfully substitute in the same input line, multiple copies of the line will be written to the output: one for each successful substitution.

`w <filename>` – write the line to a file if a successful replacement was done. The `w` flag causes lines which are actually substituted by the `s` function to be written to a file named by `<filename>`. If `<filename>` exists before `sed` is run, it is overwritten; if not, it is created.

A single space must separate `w` and `<filename>`.

The possibilities of multiple, somewhat different copies of one input line being written are the same as for `p`.

A maximum of 10 different file names may be mentioned after `w` flags and `w` functions (see below), combined.

Examples:

The following command, applied to our standard input,

```
s/to/by/w changes
```

produces, on the standard output:

```
In Xanadu did Kubla Khan
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless by man
Down by a sunless sea.
```

and, on the file 'changes':

```
Through caverns measureless by man
Down by a sunless sea.
```

If the no-copy option is in effect, the command:

```
s/[.,;?:]/*P&*/gp
```

produces:

```
A stately pleasure dome decree*P:*
Where Alph*P,* the sacred river*P,* ran
Down to a sunless sea*P.*
```

Finally, to illustrate the effect of the `g` flag, the command:

```
/X/s/an/AN/p
```

produces (assuming no-copy mode):

```
In XANadu did Kubla Khan
```

and the command:

```
/X/s/an/AN/gp
```

produces:

```
In XANadu did Kubla KhAN
```

3.3. Input-output Functions

(2)`p` – print

The print function writes the addressed lines to the standard output file. They are written at the time the `p` function is encountered, regardless of what succeeding editing commands may do to the lines.

(2)`w <filename>` – write on `<filename>`

The write function writes the addressed lines to the file named by `<filename>`. If the file previously existed, it is overwritten; if not, it is created. The lines

are written exactly as they exist when the write function is encountered for each line, regardless of what subsequent editing commands may do to them.

Exactly one space must separate the *w* and `<filename>`.

A maximum of ten different files may be mentioned in write functions and *w* flags after *s* functions, combined.

(1)*r* `<filename>` — read the contents of a file

The read function reads the contents of `<filename>`, and appends them after the line matched by the address. The file is read and appended regardless of what subsequent editing commands do to the line which matched its address. If *r* and *a* functions are executed on the same line, the text from the *a* functions and the *r* functions is written to the output in the order that the functions are executed.

Exactly one space must separate the *r* and `<filename>`. If a file mentioned by a *r* function cannot be opened, it is considered a null file, not an error, and no diagnostic is given.

NOTE: Since there is a limit to the number of files that can be opened simultaneously, care should be taken that no more than ten files be mentioned in *w* functions or flags; that number is reduced by one if any *r* functions are present. (Only one read file is open at one time.)

Examples:

Assume that the file 'note1' has the following contents:

Note: Kubla Khan (more properly Kublai Khan; 1216-1294) was the grandson and most eminent successor of Genghiz (Chingiz) Khan, and founder of the Mongol dynasty in China.

Then the following command:

```
/Kubla/r note1
```

produces:

In Xanadu did Kubla Khan

Note: Kubla Khan (more properly Kublai Khan; 1216-1294) was the grandson and most eminent successor of Genghiz (Chingiz) Khan, and founder of the Mongol dynasty in China.

A stately pleasure dome decree:

Where Alph, the sacred river, ran

Through caverns measureless to man

Down to a sunless sea.

3.4. Multiple Input-line Functions

Three functions, all spelled with capital letters, deal specially with pattern spaces containing embedded new-lines; they are intended principally to provide pattern matches across lines in the input.

(2)*N* — Next line

The next input line is appended to the current line in the pattern space; the two input lines are separated by an embedded new-line. Pattern matches may extend across the embedded new-line(s).

(2)*D* — Delete first part of the pattern space

Delete up to and including the first new-line character in the current pattern space. If the pattern space becomes empty (the only new-line was the terminal

new-line), read another line from the input. In any case, begin the list of editing commands again from its beginning.

(2)P — Print first part of the pattern space

Print up to and including the first new-line in the pattern space.

The *P* and *D* functions are equivalent to their lower-case counterparts if there are no embedded new-lines in the pattern space.

3.5. Hold and Get Functions

Four functions save and retrieve part of the input for possible later use.

(2)h — hold pattern space

The *h* function copies the contents of the pattern space into a hold area (destroying the previous contents of the hold area).

(2)H — Hold pattern space

The *H* function appends the contents of the pattern space to the contents of the hold area; the former and new contents are separated by a new-line.

(2)g — get contents of hold area

The *g* function copies the contents of the hold area into the pattern space (destroying the previous contents of the pattern space).

(2)G — Get contents of hold area

The *G* function appends the contents of the hold area to the contents of the pattern space; the former and new contents are separated by a new-line.

(2)x — exchange

The exchange command interchanges the contents of the pattern space and the hold area.

Example:

The commands

```
1h
1s/ did.*//
1x
G
s/^n/ :/
```

applied to our standard example, produce:

```
In Xanadu did Kubla Khan :In Xanadu
A stately pleasure dome decree: :In Xanadu
Where Alph, the sacred river, ran :In Xanadu
Through caverns measureless to man :In Xanadu
Down to a sunless sea. :In Xanadu
```

3.6. Flow-of-control Functions

These functions do no editing on the input lines, but control the application of functions to the lines selected by the address part.

(2)! — Don't.

The *Don't* command causes the next command (written on the same line), to

be applied to all and only those input lines *not* selected by the address part.

(2){ – Grouping

The grouping command '{' causes the next set of commands to be applied (or not applied) as a block to the input lines selected by the addresses of the grouping command. The first of the commands under control of the grouping may appear on the same line as the '{' or on the next line.

The group of commands is terminated by a matching '}' standing on a line by itself.

Groups can be nested.

(0):<label> – place a label

The label function marks a place in the list of editing commands which may be referred to by *b* and *t* functions. The <label> may be any sequence of eight or fewer characters; if two different colon functions have identical labels, a compile time diagnostic will be generated, and no execution attempted.

(2)b<label> – branch to label

The branch function causes the sequence of editing commands being applied to the current input line to be restarted immediately after the place where a colon function with the same <label> was encountered. If no colon function with the same label can be found after all the editing commands have been compiled, a compile time diagnostic is produced, and no execution is attempted.

A *b* function with no <label> is taken to be a branch to the end of the list of editing commands; whatever should be done with the current input line is done, and another input line is read; the list of editing commands is restarted from the beginning on the new line.

(2)t<label> – test substitutions

The *t* function tests whether *any* successful substitutions have been made on the current input line; if so, it branches to <label>; if not, it does nothing. The flag which indicates that a successful substitution has been executed is reset by:

- 1) reading a new input line, or
- 2) executing a *t* function.

3.7. Miscellaneous Functions

(1)= – equals

The = function writes to the standard output the line number of the line matched by its address.

(1)q – quit

The *q* function causes the current line to be written to the output (if it should be), any appended or read text to be written, and execution to be terminated.

REFERENCE

- [1] *UNIX Programmer's Manual*, Bell Laboratories, 1978.

January 1981

UNIX for Beginners (Second Edition)

Brian W. Kernighan

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This paper is meant to help new users get started on the UNIX[†] operating system. It includes:

- basics needed for day-to-day use of the system — typing commands, correcting typing mistakes, logging in and out, mail, inter-terminal communication, the file system, printing files, redirecting I/O, pipes, and the shell.
- document preparation — a brief discussion of the major formatting programs and macro packages, hints on preparing documents, and capsule descriptions of some supporting software.
- UNIX programming — using the editor, programming the shell, programming in C, other languages and tools.
- pointers to additional sources of information about UNIX.

INTRODUCTION

From the user's point of view, the UNIX operating system is easy to learn and use, and presents few of the usual impediments to getting the job done. It is hard, however, for the beginner to know where to start, and how to make the best use of the facilities available. The purpose of this introduction is to help new users get used to the main ideas of the UNIX system and start making effective use of it quickly.

You should have a couple of other documents with you for easy reference as you read this one. The most important is *UNIX User's Manual*; it's often easier to tell you to read about something in the manual than to repeat its contents here. The other useful document is *A Tutorial Introduction to the UNIX Text Editor*, which will tell you how to use the editor to get text — programs, data, documents — into the computer.

A word of warning: the UNIX system has become quite popular, and there are several major variants in widespread use. Of course details also change with time. So although the basic structure of UNIX and how to use it is common to all versions, there will certainly be a few

things which are different on your system from what is described here. We have tried to minimize the problem, but be aware of it. In cases of doubt, this paper describes Version 7 UNIX.

This paper has four main sections:

1. Getting Started: How to log in, how to type, what to do about mistakes in typing, how to log out. Some of this is dependent on which system you log into (phone numbers, for example) and what terminal you use, so this section must necessarily be supplemented by local information.
2. Day-to-day Use: Things you need every day to use the system effectively: generally useful commands; the file system.
3. Document Preparation: Preparing manuscripts is one of the most common uses for UNIX systems. This section contains advice, but not extensive instructions on any of the formatting tools.
4. Writing Programs: UNIX is an excellent system for developing programs. This section talks about some of the tools, but again is not a tutorial in any of the programming languages provided by the system.

[†] UNIX is a trademark of Bell Laboratories.

I. GETTING STARTED

Logging In

You must have a UNIX login name, which you can get from whoever administers your system. You also need to know the phone number, unless your system uses permanently connected terminals. The UNIX system is capable of dealing with a wide variety of terminals: Terminet 300s; Execuport, TI and similar portables; video (CRT) terminals like the HP 2640, etc.; high-priced graphics terminals like the Tektronix 4014; plotting terminals like those from GSI and DASI; and even the venerable Teletype in its various forms. But note: UNIX is strongly oriented towards devices with *lower case*. If your terminal produces only upper case (e.g., model 33 Teletype, some video and portable terminals), life will be so difficult that you should look for another terminal.

Be sure to set the switches appropriately on your device. Switches that might need to be adjusted include the speed, upper/lower case mode, full duplex, even parity, and any others that local wisdom advises. Establish a connection using whatever magic is needed for your terminal; this may involve dialing a telephone call or merely flipping a switch. In either case, UNIX should type "login:" at you. If it types garbage, you may be at the wrong speed; check the switches. If that fails, push the "break" or "interrupt" key a few times, slowly. If that fails to produce a login message, consult a guru.

When you get a login: message, type your login name *in lower case*. Follow it by a RETURN; the system will not do anything until you type a RETURN. If a password is required, you will be asked for it, and (if possible) printing will be turned off while you type it. Don't forget RETURN.

The culmination of your login efforts is a "prompt character," a single character that indicates that the system is ready to accept commands from you. The prompt character is usually a dollar sign \$ or a percent sign %. (You may also get a message of the day just before the prompt character, or a notification that you have mail.)

Typing Commands

Once you've seen the prompt character, you can type commands, which are requests that the system do something. Try typing

date

followed by RETURN. You should get back something like

Mon Jan 16 14:17:10 EST 1978

Don't forget the RETURN after the command, or nothing will happen. If you think you're being ignored, type a RETURN; something should happen. RETURN won't be mentioned again, but don't forget it — it has to be there at the end of each line.

Another command you might try is **who**, which tells you everyone who is currently logged in:

who

gives something like

mb	tty01	Jan 16	09:11
ski	tty05	Jan 16	09:33
gam	tty11	Jan 16	13:07

The time is when the user logged in; "ttyxx" is the system's idea of what terminal the user is on.

If you make a mistake typing the command name, and refer to a non-existent command, you will be told. For example, if you type

whom

you will be told

whom: not found

Of course, if you inadvertently type the name of some other command, it will run, with more or less mysterious results.

Strange Terminal Behavior

Sometimes you can get into a state where your terminal acts strangely. For example, each letter may be typed twice, or the RETURN may not cause a line-feed or a return to the left margin. You can often fix this by logging out and logging back in. Or you can read the description of the command **stty** in Section 1 of the *UNIX User's Manual*. To get intelligent treatment of tab characters (which are much used in UNIX) if your terminal doesn't have tabs, type

stty -tabs

and the system will convert each tab into the right number of blanks for you. If your terminal does have computer-settable tabs, the command **tabs** will set the stops correctly for you.

Mistakes in Typing

If you make a typing mistake, and see it before RETURN has been typed, there are two ways to recover. The sharp-character # erases the last character typed; in fact successive uses of # erase characters back to the beginning of the line (but not beyond). So if you type badly, you can correct as you go:

dd#atte##e

is the same as **date**.

The at-sign **@** erases all of the characters typed so far on the current input line, so if the line is irretrievably fouled up, type an **@** and start the line over.

What if you must enter a sharp or at-sign as part of the text? If you precede either **#** or **@** by a backslash ****, it loses its erase meaning. So to enter a sharp or at-sign in something, type **\#** or **\@**. The system will always echo a new-line at you after your at-sign, even if preceded by a backslash. Don't worry — the at-sign has been recorded.

To erase a backslash, you have to type two sharps or two at-signs, as in **\##**. The backslash is used extensively in UNIX to indicate that the following character is in some way special.

Read-ahead

UNIX has full read-ahead, which means that you can type as fast as you want, whenever you want, even when some command is typing at you. If you type during output, your input characters will appear intermixed with the output characters, but they will be stored away and interpreted in the correct order. So you can type several commands one after another without waiting for the first to finish or even begin.

Stopping a Program

You can stop most programs by typing the character "DEL" (perhaps called "delete" or "rubout" on your terminal). The "interrupt" or "break" key found on most terminals can also be used. In a few programs, like the text editor, DEL stops whatever the program is doing but leaves you in that program. Hanging up the phone will stop most programs.

Logging Out

The easiest way to log out is to hang up the phone. You can also type

login

and let someone else use the terminal you were on. It is usually not sufficient just to turn off the terminal. Most UNIX systems do not use a time-out mechanism, so you'll be there forever unless you hang up.

Mail

When you log in, you may sometimes get the message

You have mail.

UNIX provides a postal system so you can communicate with other users of the system. To read your mail, type the command

mail

Your mail will be printed, one message at a time, most recent message first. After each message, **mail** waits for you to say what to do with it. The two basic responses are **d**, which deletes the message, and RETURN, which does not (so it will still be there the next time you read your mailbox). Other responses are described in the manual. (Earlier versions of **mail** do not process one message at a time, but are otherwise similar.)

How do you send mail to someone else? Suppose it is to go to "joe" (assuming "joe" is someone's login name). The easiest way is this:

mail joe

*now type in the text of the letter
on as many lines as you like ...*

*After the last line of the letter
type the character "control-d",
that is, hold down "control" and type
a letter "d".*

And that's it. The "control-d" sequence, often called "EOF" for end-of-file, is used throughout the system to mark the end of input from a terminal, so you might as well get used to it.

For practice, send mail to yourself. (This isn't as strange as it might sound — mail to oneself is a handy reminder mechanism.)

There are other ways to send mail — you can send a previously prepared letter, and you can mail to a number of people all at once. For more details see **mail(1)**. (The notation **mail(1)** means the command **mail** in Section 1 of the *UNIX User's Manual*.)

Writing to other users

At some point, out of the blue will come a message like

Message from joe tty07 ...

accompanied by a startling beep. It means that Joe wants to talk to you, but unless you take explicit action you won't be able to talk back. To respond, type the command

write.joe

This establishes a two-way communication path. Now whatever Joe types on his terminal will appear on yours and vice versa. The path is slow, rather like talking to the moon. (If you are in the middle of something, you have to get to a state where you can type a command. Normally, whatever program you are running has to

terminate or be terminated. If you're editing, you can escape temporarily from the editor — read the editor tutorial.)

A protocol is needed to keep what you type from getting garbled up with what Joe types. Typically it's like this:

Joe types **write smith** and waits.

Smith types **write joe** and waits.

Joc now types his message (as many lines as he likes). When he's ready for a reply, he signals it by typing **(o)**, which stands for "over".

Now Smith types a reply, also terminated by **(o)**.

This cycle repeats until someone gets tired; he then signals his intent to quit with **(oo)**, for "over and out".

To terminate the conversation, each side must type a "control-d" character alone on a line. ("Delete" also works.) When the other person types his "control-d", you will get the message **EOF** on your terminal.

If you write to someone who isn't logged in, or who doesn't want to be disturbed, you'll be told. If the target is logged in but doesn't answer after a decent interval, simply type "control-d".

On-line Manual

The *UNIX User's Manual* is typically kept on-line. If you get stuck on something, and can't find an expert to assist you, you can print on your terminal some manual section that might help. This is also useful for getting the most up-to-date information on a command. To print a manual section, type "man command-name". Thus to read up on the **who** command, type

```
man who
```

and, of course,

```
man man
```

tells all about the **man** command.

Computer Aided Instruction

Your UNIX system may have available a program called **learn**, which provides computer aided instruction on the file system and basic commands, the editor, document preparation, and even C programming. Try typing the command

```
learn
```

If **learn** exists on your system, it will tell you what to do from there.

II. DAY-TO-DAY USE

Creating Files — The Editor

If you have to type a paper or a letter or a program, how do you get the information stored in the machine? Most of these tasks are done with the UNIX "text editor" **ed**. Since **ed** is thoroughly documented in **ed(1)** and explained in *A Tutorial Introduction to the UNIX Text Editor*, we won't spend any time here describing how to use it. All we want it for right now is to make some *files*. (A file is just a collection of information stored in the machine, a simplistic but adequate definition.)

To create a file called **junk** with some text in it, do the following:

```
ed junk    (invokes the text editor)
a          (command to "ed" to add text)
now type in
whatever text you want ...
.         (signals the end of adding text)
```

The "." that signals the end of adding text must be at the beginning of a line by itself. Don't forget it, for until it is typed, no other **ed** commands will be recognized — everything you type will be treated as text to be added.

At this point you can do various editing operations on the text you typed in, such as correcting spelling mistakes, rearranging paragraphs and the like. Finally, you must write the information you have typed into a file with the editor command **w**:

```
w
```

ed will respond with the number of characters it wrote into the file **junk**.

Until the **w** command, nothing is stored permanently, so if you hang up and go home the information is lost.* But after **w** the information is there permanently; you can re-access it any time by typing

```
ed junk
```

Type a **q** command to quit the editor. (If you try to quit without writing, **ed** will print a ? to remind you. A second **q** gets you out regardless.)

Now create a second file called **temp** in the same manner. You should now have two files, **junk** and **temp**.

* This is not strictly true — if you hang up while editing, the data you were working on is saved in a file called **ed.hup**, which you can continue with at your next session.

What files are out there?

The `ls` (for “list”) command lists the names (not contents) of any of the files that UNIX knows about. If you type

```
ls
```

the response will be

```
junk
temp
```

which are indeed the two files just created. The names are sorted into alphabetical order automatically, but other variations are possible. For example, the command

```
ls -t
```

causes the files to be listed in the order in which they were last changed, most recent first. The `-l` option gives a “long” listing:

```
ls -l
```

will produce something like

```
-rw-rw-rw- 1 bwk 41 Jul 22 2:56 junk
-rw-rw-rw- 1 bwk 78 Jul 22 2:57 temp
```

The date and time are of the last change to the file. The 41 and 78 are the number of characters (which should agree with the numbers you got from `ed`). `bwk` is the owner of the file, that is, the person who created it. The `-rw-rw-rw-` tells who has permission to read and write the file, in this case everyone.

Options can be combined: `ls -lt` gives the same thing as `ls -l`, but sorted into time order. You can also name the files you’re interested in, and `ls` will list the information about them only. More details can be found in `ls(1)`.

The use of optional arguments that begin with a minus sign, like `-t` and `-lt`, is a common convention for UNIX programs. In general, if a program accepts such optional arguments, they precede any file name arguments. It is also vital that you separate the various arguments with spaces: `ls-l` is not the same as `ls -l`.

Printing Files

Now that you’ve got a file of text, how do you print it so people can look at it? There are a host of programs that do that, probably more than are needed.

One simple thing is to use the editor, since printing is often done just before making changes anyway. You can say

```
ed junk
1,Sp
```

`ed` will reply with the count of the characters in `junk` and then print all the lines in the file.

After you learn how to use the editor, you can be selective about the parts you print.

There are times when it’s not feasible to use the editor for printing. For example, there is a limit on how big a file `ed` can handle (several thousand lines). Secondly, it will only print one file at a time, and sometimes you want to print several, one after another. So here are a couple of alternatives.

First is `cat`, the simplest of all the printing programs. `cat` simply prints on the terminal the contents of all the files named in a list. Thus

```
cat junk
```

prints one file, and

```
cat junk temp
```

prints two. The files are simply concatenated (hence the name “`cat`”) onto the terminal.

`pr` produces formatted printouts of files. As with `cat`, `pr` prints all the files named in a list. The difference is that it produces headings with date, time, page number and file name at the top of each page, and extra lines to skip over the fold in the paper. Thus,

```
pr junk temp
```

will print `junk` neatly, then skip to the top of a new page and print `temp` neatly.

`pr` can also produce multi-column output:

```
pr -3 junk
```

prints `junk` in 3-column format. You can use any reasonable number in place of “3” and `pr` will do its best. `pr` has other capabilities as well; see `pr(1)`.

It should be noted that `pr` is *not* a formatting program in the sense of shuffling lines around and justifying margins. The true formatters are `nroff` and `troff`, which we will get to in the section on document preparation.

There are also programs that print files on a high-speed printer. Look in your manual under `opr` and `lpr`. Which to use depends on what equipment is attached to your machine.

Shuffling Files About

Now that you have some files in the file system and some experience in printing them, you can try bigger things. For example, you can move a file from one place to another (which amounts to giving it a new name), like this:

```
mv junk precious
```

This means that what used to be “junk” is now “precious”. If you do an `ls` command now, you will get

precious
temp

Beware that if you move a file to another one that already exists, the already existing contents are lost forever.

If you want to make a *copy* of a file (that is, to have two versions of something), you can use the **cp** command:

cp precious temp1

makes a duplicate copy of **precious** in **temp1**.

Finally, when you get tired of creating and moving files, there is a command to remove files from the file system, called **rm**.

rm temp temp1

will remove both of the files named.

You will get a warning message if one of the named files wasn't there, but otherwise **rm**, like most UNIX commands, does its work silently. There is no prompting or chatter, and error messages are occasionally curt. This terseness is sometimes disconcerting to newcomers, but experienced users find it desirable.

What's in a File Name

So far we have used file names without ever saying what's a legal name, so it's time for a couple of rules. First, file names are limited to 14 characters, which is enough to be descriptive. Second, although you can use almost any character in a file name, common sense says you should stick to ones that are visible, and that you should probably avoid characters that might be used with other meanings. We have already seen, for example, that in the **ls** command, **ls -t** means to list in time order. So if you had a file whose name was **-t**, you would have a tough time listing it by name. Besides the minus sign, there are other characters which have special meaning. To avoid pitfalls, you would do well to use only letters, numbers and the period until you're familiar with the situation.

On to some more positive suggestions. Suppose you're typing a large document like a book. Logically this divides into many small pieces, like chapters and perhaps sections. Physically it must be divided too, for **ed** will not handle really big files. Thus you should type the document as a number of files. You might have a separate file for each chapter, called

chap1
chap2
etc. ...

Or, if each chapter were broken into several files, you might have

chap1.1
chap1.2
chap1.3
...
chap2.1
chap2.2
...

You can now tell at a glance where a particular file fits into the whole.

There are advantages to a systematic naming convention which are not obvious to the novice UNIX user. What if you wanted to print the whole book? You could say

pr chap1.1 chap1.2 chap1.3 ...

but you would get tired pretty fast, and would probably even make mistakes. Fortunately, there is a shortcut. You can say

pr chap*

The ***** means "anything at all," so this translates into "print all files whose names begin with **chap**", listed in alphabetical order.

This shorthand notation is not a property of the **pr** command, by the way. It is system-wide, a service of the program that interprets commands (the "shell," **sh(1)**). Using that fact, you can see how to list the names of the files in the book:

ls chap*

produces

chap1.1
chap1.2
chap1.3
...

The ***** is not limited to the last position in a file name — it can be anywhere and can occur several times. Thus

rm *junk* *temp*

removes all files that contain **junk** or **temp** as any part of their name. As a special case, ***** by itself matches every file name, so

pr *

prints all your files (alphabetical order), and

rm *

removes *all files*. (You had better be *very* sure that's what you wanted to say!)

The ***** is not the only pattern-matching feature available. Suppose you want to print only chapters 1 through 4 and 9. Then you can say

```
pr chap[12349]*
```

The [...] means to match any of the characters inside the brackets. A range of consecutive letters or digits can be abbreviated, so you can also do this with

```
pr chap[1-49]*
```

Letters can also be used within brackets: [a-z] matches any character in the range a through z.

The ? pattern matches any single character, so

```
ls ?
```

lists all files which have single-character names, and

```
ls -l chap?.1
```

lists information about the first file of each chapter (**chap1.1**, **chap2.1**, etc.).

Of these niceties, * is certainly the most useful, and you should get used to it. The others are frills, but worth knowing.

If you should ever have to turn off the special meaning of *, ?, etc., enclose the entire argument in single quotes, as in

```
ls '?'
```

We'll see some more examples of this shortly.

What's in a File Name, Continued

When you first made that file called **junk**, how did the system know that there wasn't another **junk** somewhere else, especially since the person in the next office is also reading this tutorial? The answer is that generally each user has a private *directory*, which contains only the files that belong to him. When you log in, you are "in" your directory. Unless you take special action, when you create a new file, it is made in the directory that you are currently in; this is most often your own directory, and thus the file is unrelated to any other file of the same name that might exist in someone else's directory.

The set of all files is organized into a (usually big) tree, with your files located several branches into the tree. It is possible for you to "walk" around this tree, and to find any file in the system, by starting at the root of the tree and walking along the proper set of branches. Conversely, you can start where you are and walk toward the root.

Let's try the latter first. The basic tools is the command **pwd** ("print working directory"), which prints the name of the directory you are currently in.

Although the details will vary according to the system you are on, if you give the command **pwd**, it will print something like

```
/usr/your_name
```

This says that you are currently in the directory **your_name**, which is in turn in the directory **/usr**, which is in turn in the root directory called by convention just **/**. (Even if it's not called **/usr** on your system, you will get something analogous. Make the corresponding changes and read on.)

If you now type

```
ls /usr/your_name
```

you should get exactly the same list of file names as you get from a plain **ls**: with no arguments, **ls** lists the contents of the current directory; given the name of a directory, it lists the contents of that directory.

Next, try

```
ls /usr
```

This should print a long series of names, among which is your own login name **your_name**. On many systems, **usr** is a directory that contains the directories of all the normal users of the system, like you.

The next step is to try

```
ls /
```

You should get a response something like this (although again the details may be different):

```
bin
dev
etc
lib
tmp
usr
```

This is a collection of the basic directories of files that the system knows about; we are at the root of the tree.

Now try

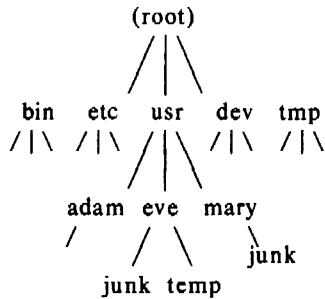
```
cat /usr/your_name/junk
```

(if **junk** is still around in your directory). The name

```
/usr/your_name/junk
```

is called the **path-name** of the file that you normally think of as "junk". "Path-name" has an obvious meaning: it represents the full name of the path you have to follow from the root through the tree of directories to get to a particular file. It is a universal rule in the UNIX system that anywhere you can use an ordinary file name, you can use a path-name.

Here is a picture that may make this clearer:



Notice that Mary's **junk** is unrelated to Eve's.

This isn't too exciting if all the files of interest are in your own directory, but if you work with someone else or on several projects concurrently, it becomes handy indeed. For example, your friends can print your book by saying

```
pr /usr/your_name/chap*
```

Similarly, you can find out what files your neighbor has by saying

```
ls /usr/neighbor_name
```

or make your own copy of one of his files by

```
cp /usr/your_neighbor/his_file your_file
```

If your neighbor doesn't want you poking around in his files, or vice versa, privacy can be arranged. Each file and directory has read-write-execute permissions for the owner, a group, and everyone else, which can be set to control access. See **ls(1)** and **chmod(1)** for details. As a matter of observed fact, most users most of the time find openness of more benefit than privacy.

As a final experiment with path-names, try

```
ls /bin /usr/bin
```

Do some of the names look familiar? When you run a program, by typing its name after the prompt character, the system simply looks for a file of that name. It normally looks first in your directory (where it typically doesn't find it), then in **/bin** and finally in **/usr/bin**. There is nothing magic about commands like **cat** or **ls**, except that they have been collected into a couple of places to be easy to find and administer.

What if you work regularly with someone else on common information in his directory? You could just log in as your friend each time you want to, but you can also say "I want to work on his files instead of my own". This is done by changing the directory that you are currently in:

```
cd /usr/your_friend
```

Now when you use a file name in something like **cat** or **pr**, it refers to the file in your friend's directory. Changing directories doesn't affect any permissions associated with a file — if you couldn't access a file from your own directory, changing to another directory won't alter that fact. Of course, if you forget what directory you're in, type

```
pwd
```

to find out.

It is usually convenient to arrange your own files so that all the files related to one thing are in a directory separate from other projects. For example, when you write your book, you might want to keep all the text in a directory called **book**. So make one with

```
mkdir book
```

then go to it with

```
cd book
```

then start typing chapters. The book is now found in (presumably)

```
/usr/your_name/book
```

To remove the directory **book**, type

```
rm book/*
rmdir book
```

The first command removes all files from the directory; the second removes the empty directory.

You can go up one level in the tree of files by saying

```
cd ..
```

".." is the name of the parent of whatever directory you are currently in. For completeness, "." is an alternate name for the directory you are in.

Using Files instead of the Terminal

Most of the commands we have seen so far produce output on the terminal; some, like the editor, also take their input from the terminal. It is universal in UNIX systems that the terminal can be replaced by a file for either or both of input and output. As one example,

```
ls
```

makes a list of files on your terminal. But if you say

```
ls >file_list
```

a list of your files will be placed in the file **file_list** (which will be created if it doesn't already exist, or overwritten if it does). The symbol **>** means "put the output on the

following file, rather than on the terminal.” Nothing is produced on the terminal. As another example, you could combine several files into one by capturing the output of `cat` in a file:

```
cat f1 f2 f3 >temp
```

The symbol `>>` operates very much like `>` does, except that it means “add to the end of.” That is,

```
cat f1 f2 f3 >>temp
```

means to concatenate `f1`, `f2` and `f3` to the end of whatever is already in `temp`, instead of overwriting the existing contents. As with `>`, if `temp` doesn’t exist, it will be created for you.

In a similar way, the symbol `<` means to take the input for a program from the following file, instead of from the terminal. Thus, you could make up a script of commonly used editing commands and put them into a file called `script`. Then you can run the script on a file by saying

```
ed file <script
```

As another example, you can use `ed` to prepare a letter in file `let`, then send it to several people with

```
mail adam eve mary joe <let
```

Pipes

One of the novel contributions of the UNIX system is the idea of a *pipe*. A pipe is simply a way to connect the output of one program to the input of another program, so the two run as a sequence of processes — a pipeline.

For example,

```
pr f g h
```

will print the files `f`, `g`, and `h`, beginning each on a new page. Suppose you want them run together instead. You could say

```
cat f g h >temp
pr <temp
rm temp
```

but this is more work than necessary. Clearly what we want is to take the output of `cat` and connect it to the input of `pr`. So let us use a pipe:

```
cat f g h | pr
```

The vertical bar `|` means to take the output from `cat`, which would normally have gone to the terminal, and put it into `pr` to be neatly formatted.

There are many other examples of pipes. For example,

```
ls | pr -3
```

prints a list of your files in three columns. The program `wc` counts the number of lines, words and characters in its input, and as we saw earlier, `who` prints a list of currently-logged on people, one per line. Thus

```
who | wc
```

tells how many people are logged on. And of course

```
ls | wc
```

counts your files.

Any program that reads from the terminal can read from a pipe instead; any program that writes on the terminal can drive a pipe. You can have as many elements in a pipeline as you wish.

Many UNIX programs are written so that they will take their input from one or more files if file arguments are given; if no arguments are given they will read from the terminal, and thus can be used in pipelines. `pr` is one example:

```
pr -3 a b c
```

prints files `a`, `b` and `c` in order in three columns. But in

```
cat a b c | pr -3
```

`pr` prints the information coming down the pipeline, still in three columns.

The Shell

We have already mentioned once or twice the mysterious “shell,” which is in fact `sh(1)`. The shell is the program that interprets what you type as commands and arguments. It also looks after translating `*`, etc., into lists of file names, and `<`, `>`, and `|` into changes of input and output streams.

The shell has other capabilities too. For example, you can run two programs with one command line by separating the commands with a semicolon; the shell recognizes the semicolon and breaks the line into two commands. Thus

```
date; who
```

does both commands before returning with a prompt character.

You can also have more than one program running *simultaneously* if you wish. For example, if you are doing something time-consuming, like the editor script of an earlier section, and you don’t want to wait around for the results before starting something else, you can say

```
ed file <script &
```

The ampersand at the end of a command line says "start this command running, then take further commands from the terminal immediately," that is, don't wait for it to complete. Thus the script will begin, but you can do something else at the same time. Of course, to keep the output from interfering with what you're doing on the terminal, it would be better to say

```
ed file <script >script.out &
```

which saves the output lines in a file called **script.out**.

When you initiate a command with **&**, the system replies with a number called the process number, which identifies the command in case you later want to stop it. If you do, you can say

```
kill process_number
```

If you forget the process number, the command **ps** will tell you about everything you have running. (If you are desperate, **kill 0** will kill all your processes.) And if you're curious about other people, **ps a** will tell you about *all* programs that are currently running.

You can say

```
(command_1; command_2; command_3) &
```

to start three commands in the background, or you can start a background pipeline with

```
command_1 | command_2 &
```

Just as you can tell the editor or some similar program to take its input from a file instead of from the terminal, you can tell the shell to read a file to get commands. (Why not? The shell, after all, is just a program, albeit a clever one.) For instance, suppose you want to set tabs on your terminal, and find out the date and who's on the system every time you log in. Then you can put the three necessary commands (**tabs**, **date**, **who**) into a file, let's call it **startup**, and then run it with

```
sh startup
```

This says to run the shell with the file **startup** as input. The effect is as if you had typed the contents of **startup** on the terminal.

If this is to be a regular thing, you can eliminate the need to type **sh**: simply type, once only, the command

```
chmod +x startup
```

and thereafter you need only say

```
startup
```

to run the sequence of commands. The **chmod(1)** command marks the file executable; the shell recognizes this and runs it as a sequence of commands.

If you want **startup** to run automatically every time you log in, create a file in your login directory called **.profile**, and place in it the line **startup**. When the shell first gains control when you log in, it looks for the **.profile** file and does whatever commands it finds in it. We'll get back to the shell in the section on programming.

III. DOCUMENT PREPARATION

UNIX systems are used extensively for document preparation. There are two major formatting programs, that is, programs that produce a text with justified right margins, automatic page numbering and titling, automatic hyphenation, and the like. **nroff** is designed to produce output on terminals and line-printers. **troff** (pronounced "tee-roff") instead drives a phototypesetter, which produces very high quality output on photographic paper. This paper was formatted with **troff**.

Formatting Packages

The basic idea of **nroff** and **troff** is that the text to be formatted contains within it "formatting commands" that indicate in detail how the formatted text is to look. For example, there might be commands that specify how long lines are, whether to use single or double spacing, and what running titles to use on each page.

Because **nroff** and **troff** are relatively hard to learn to use effectively, several "packages" of canned formatting requests are available to let you specify paragraphs, running titles, footnotes, multi-column output, and so on, with little effort and without having to learn **nroff** and **troff**. These packages take a modest effort to learn, but the rewards for using them are so great that it is time well spent.

In this section, we will provide a hasty look at the "manuscript" package known as **-ms**. Formatting requests typically consist of a period and two upper-case letters, such as **.TL**, which is used to introduce a title, or **.PP** to begin a new paragraph.

A document is typed so it looks something like this:

.TL
title of document
.AU
author name
.SH
section heading
.PP
paragraph ...
.PP
another paragraph ...
.SH
another section heading
.PP
etc.

The lines that begin with a period are the formatting requests. For example, **.PP** calls for starting a new paragraph. The precise meaning of **.PP** depends on what output device is being used (typesetter or terminal, for instance), and on what publication the document will appear in. For example, **-ms** normally assumes that a paragraph is preceded by a space (one line in **nroff**, ½ line in **troff**), and the first word is indented. These rules can be changed if you like, but they are changed by changing the interpretation of **.PP**, not by re-typing the document.

To actually produce a document in standard format using **-ms**, use the command

```
troff -ms files ...
```

for the typesetter, and

```
nroff -ms files ...
```

for a terminal. The **-ms** argument tells **troff** and **nroff** to use the manuscript package of formatting requests.

There are several similar packages; check with a local expert to determine which ones are in common use on your machine.

Supporting Tools

In addition to the basic formatters, there is a host of supporting programs that help with document preparation. The list in the next few paragraphs is far from complete, so browse through the manual and check with people around you for other possibilities.

eqn and **neqn** let you integrate mathematics into the text of a document, in an easy-to-learn language that closely resembles the way you would speak it aloud. For example, the **eqn** input

```
sum from i=0 to n x sub i ~ ~ pi over 2
```

produces the output

$$\sum_{i=0}^n x_i = \frac{\pi}{2}$$

The program **tbl** provides an analogous service for preparing tabular material; it does all the computations necessary to align complicated columns with elements of varying widths.

spell and **typo** detect possible spelling mistakes in a document. **spell** works by comparing the words in your document to a dictionary, printing those that are not in the dictionary. It knows enough about English spelling to detect plurals and the like, so it does a very good job. **typo** looks for words which are "unusual", and prints those. Spelling mistakes tend to be more unusual, and thus show up early when the most unusual words are printed first.

grep looks through a set of files for lines that contain a particular text pattern (rather like the editor's context search does, but on a bunch of files). For example,

```
grep 'ing$' chap*
```

will find all lines that end with the letters **ing** in the files **chap***. (It is almost always a good practice to put single quotes around the pattern you're searching for, in case it contains characters like ***** or **\$** that have a special meaning to the shell.) **grep** is often useful for finding out in which of a set of files the misspelled words detected by **spell** are actually located.

diff prints a list of the differences between two files, so you can compare two versions of something automatically (which certainly beats proofreading by hand).

wc counts the words, lines and characters in a set of files. **tr** translates characters into other characters; for example it will convert upper to lower case and vice versa. This translates upper into lower:

```
tr A-Z a-z <input >output
```

sort sorts files in a variety of ways; **cref** makes cross-references; **ptx** makes a permuted index (keyword-in-context listing). **sed** provides many of the editing facilities of **ed**, but can apply them to arbitrarily long inputs. **awk** provides the ability to do both pattern matching and numeric computations, and to conveniently process fields within lines. These programs are for more advanced users, and they are not limited to document preparation. Put them on your list of things to learn about.

Most of these programs are either independently documented (like **eqn** and **tbl**), or are sufficiently simple that the description in the *UNIX User's Manual* is adequate explanation.

Hints for Preparing Documents

Most documents go through several versions (always more than you expected) before they are finally finished. Accordingly, you should do whatever possible to make the job of changing them easy.

First, when you do the purely mechanical operations of typing, type so that subsequent editing will be easy. Start each sentence on a new line. Make lines short, and break lines at natural places, such as after commas and semicolons, rather than randomly. Since most people change documents by rewriting phrases and adding, deleting and rearranging sentences, these precautions simplify any editing you have to do later.

Keep the individual files of a document down to modest size, perhaps ten to fifteen thousand characters. Larger files edit more slowly, and of course if you make a dumb mistake it's better to have clobbered a small file than a big one. Split into files at natural boundaries in the document, for the same reasons that you start each sentence on a new line.

The second aspect of making change easy is to not commit yourself to formatting details too early. One of the advantages of formatting packages like `ms` is that they permit you to delay decisions to the last possible moment. Indeed, until a document is printed, it is not even decided whether it will be typeset or put on a line printer.

As a rule of thumb, for all but the most trivial jobs, you should type a document in terms of a set of requests like `.PP`, and then define them appropriately, either by using one of the canned packages (the better way) or by defining your own `nroff` and `troff` commands. As long as you have entered the text in some systematic way, it can always be cleaned up and reformatted by a judicious combination of editing commands and request definitions.

IV. PROGRAMMING

There will be no attempt made to teach any of the programming languages available but a few words of advice are in order. One of the reasons why the UNIX system is a productive programming environment is that there is already a rich set of tools available, and facilities like pipes, I/O redirection, and the capabilities of the shell often make it possible to do a job by pasting together programs that already exist instead of writing from scratch.

The Shell

The pipe mechanism lets you fabricate quite complicated operations out of spare parts that already exist. For example, the first draft of the `spell` program was (roughly)

```
cat ...    collect the files
| tr ...   put each word on a new line
| tr ...   delete punctuation, etc.
| sort     into dictionary order
| uniq     discard duplicates
| comm     print words in text
           but not in dictionary
```

More pieces have been added subsequently, but this goes a long way for such a small effort.

The editor can be made to do things that would normally require special programs on other systems. For example, to list the first and last lines of each of a set of files, such as a book, you could laboriously type

```
ed
e chap1.1
lp
Sp
e chap1.2
lp
Sp
etc.
```

But you can do the job much more easily. One way is to type

```
ls chap* >temp
```

to get the list of file names into a file. Then edit this file to make the necessary series of editing commands (using the global commands of `ed`), and write it into `script`. Now the command

```
ed <script
```

will produce the same output as the laborious hand typing. Alternately (and more easily), you can use the fact that the shell will perform loops, repeating a set of commands over and over again for a set of arguments:

```
for i in chap*
do
    ed $i <script
done
```

This sets the shell variable `i` to each file name in turn, then does the command. You can type this command at the terminal, or put it in a file for later execution.

Programming the Shell

An option often overlooked by newcomers is that the shell is itself a programming language, with variables, control flow (`if-else`, `while`, `for`,

case), subroutines, and interrupt handling. Since there are many building-block programs, you can sometimes avoid writing a new program merely by piecing together some of the building blocks with shell command files.

We will not go into any details here; examples and rules can be found in the *UNIX Shell Tutorial*, by G. A. Snyder and J. R. Mashey.

Programming in C

If you are undertaking anything substantial, C is the only reasonable choice of programming language: everything in the UNIX system is tuned to it. The system itself is written in C, as are most of the programs that run on it. It is an easy language to use once you get started. C is introduced and fully described in *The C Programming Language* by B. W. Kernighan and D. M. Ritchie (Prentice-Hall, 1978); several sections describe the system interfaces, that is, how you do I/O and similar functions. Read *UNIX Programming* for more complicated things.

Most input and output in C is best handled with the standard I/O library, which provides a set of I/O functions that exist in compatible form on most machines that have C compilers. In general, it's wisest to confine the system interactions in a program to the facilities provided by this library.

C programs that don't depend too much on special features of UNIX (such as pipes) can be moved to other computers that have C compilers. The list of such machines grows daily; in addition to the original PDP-11, it currently includes at least Honeywell 6000, IBM 370, Interdata 8/32, Data General Nova and Eclipse, HP 2100, Harris /7, VAX 11/780, SEL 86, and Zilog Z80. Calls to the standard I/O library will work on all of these machines.

There are a number of supporting programs that go with C. `lint` checks C programs for potential portability problems, and detects errors such as mismatched argument types and uninitialized variables.

For larger programs (anything whose source is on more than one file) `make` allows you to specify the dependencies among the source files and the processing steps needed to make a new version; it then checks the times that the pieces were last changed and does the minimal amount of recompiling to create a consistent updated version.

The debugger `adb` is useful for digging through the dead bodies of C programs, but is rather hard to learn to use effectively. The most effective debugging tool is still careful thought, coupled with judiciously placed print statements.

The C compiler provides a limited instrumentation service, so you can find out where programs spend their time and what parts are worth optimizing. Compile the routines with the `-p` option; after the test run, use `prof` to print an execution profile. The command `time` will give you the gross run-time statistics of a program, but they are not super accurate or reproducible.

Other Languages

If you *have* to use Fortran, there are two possibilities. You might consider Ratfor, which gives you the decent control structures and free-form input that characterize C, yet lets you write code that is still portable to other environments. Bear in mind that UNIX Fortran tends to produce large and relatively slow-running programs. Furthermore, supporting software like `adb`, `prof`, etc., are all virtually useless with Fortran programs. There may also be a Fortran 77 compiler on your system. If so, this is a viable alternative to Ratfor, and has the non-trivial advantage that it is compatible with C and related programs. (The Ratfor processor and C tools can be used with Fortran 77 too.)

If your application requires you to translate a language into a set of actions or another language, you are in effect building a compiler, though probably a small one. In that case, you should be using the `yacc` compiler-compiler, which helps you develop a compiler quickly. The `lex` lexical analyzer generator does the same job for the simpler languages that can be expressed as regular expressions. It can be used by itself, or as a front end to recognize inputs for a `yacc`-based program. Both `yacc` and `lex` require some sophistication to use, but the initial effort of learning them can be repaid many times over in programs that are easy to change later on.

Most UNIX systems also make available other languages, such as Algol 68, APL, Basic, Lisp, Pascal, and Snobol. Whether these are useful depends largely on the local environment: if someone cares about the language and has worked on it, it may be in good shape. If not, the odds are strong that it will be more trouble than it's worth.

V. ADDITIONAL READING

See the *UNIX Documentation Road Map* by G. A. Snyder and J. R. Mashey for additional reading suggestions.

January 1981

UNIX Shell Tutorial

G. A. Snyder
J. R. Mashey

Bell Laboratories
Murray Hill, New Jersey 07974

1. INTRODUCTION

In any programming project, some effort is used to build the end product. The remainder is consumed in building the supporting tools and procedures used to manage and maintain that end product. The second effort can far exceed the first, especially in larger projects. A good command language can be an invaluable tool in such situations. If it is a flexible programming language, it can be used to solve many internal support problems without requiring compilable programs to be written, debugged, and maintained; its most important advantage is the ability to get the job done *now*. For a perspective on the motivations for using a command language in this way, see [1,4,5,6].

When users log into a UNIX[†] system, they communicate with an instance of the shell that reads commands typed at the terminal and arranges for their execution. Thus, the shell's most important function is to provide a good interface for human beings. In addition, a sequence of commands may be preserved for repeated use by saving it in a file, called a *shell procedure*, a *command file*, or a *runcom*, according to local preference.

Some UNIX users need little knowledge of the shell to do their work; others make heavy use of its programming features. This tutorial may be read in several different ways, depending on the reader's interests. A brief discussion of the UNIX environment is found in §2. The discussion in §3 covers aspects of the shell that are important for everyone, while all of §4 and most of §5 are mainly of interest to those who write shell procedures. A group of annotated shell procedure examples is given in §6. Finally, a brief discussion of efficiency is offered in §7; this is found in its proper place (at the end), and is intended for those who write especially time-consuming shell procedures.

Complete beginners should *not* be reading this tutorial, but should work their way through other available tutorials first. See [14] for an appropriate plan of study. All the *commands* mentioned below are described in Section 1 of the *UNIX User's Manual* [7], while *system calls* are described in Section 2 and *subroutines* in Section 3 thereof; references of the form *name(N)* point to entry *name* in Section *N* of that manual.

2. OVERVIEW OF THE UNIX ENVIRONMENT

Full understanding of what follows depends on familiarity with UNIX; [13] is useful for that, and it would be helpful to read [8] and at least one of [9,10]. For completeness, a short overview of the most relevant concepts is given below.

2.1 File System

The UNIX file system's overall structure is that of a rooted tree composed of *directories* and other files. A simple *file name* is a sequence of characters other than a slash (/). A *path name* is a sequence of directory names followed by a simple file name, each separated from the previous one by a /. If a path name begins with a /, the search for the file begins at the *root* of the entire tree; otherwise, it begins at the user's *current directory* (also known as the *working directory*). The first kind of name is often called a *full* (or *absolute*) *path name* because it is invariant

[†] UNIX is a trademark of Bell Laboratories.

with regard to the user's current directory. The latter is often called a *relative path name*, because it specifies a path relative to the current directory. The user may change the current directory at any time by using the `cd` command. In most cases, a file name and its corresponding path name may be used interchangeably. Some sample names are:

<code>/</code>	absolute path name of the root directory of the entire file structure.
<code>/bin</code>	directory containing most of the frequently used public commands.
<code>/a1/tf/jtb/bin</code>	a full path name typical of multi-person programming projects. This one happens to be a private directory of commands belonging to person <code>jtb</code> in project <code>tf</code> ; <code>a1</code> is the name of a <i>file system</i> .
<code>bin/x</code>	a relative path name; it names file <code>x</code> in subdirectory <code>bin</code> of the current directory. If the current directory is <code>/</code> , it names <code>/bin/x</code> . If, on the other hand, the current directory is <code>/a1/tf/jtb</code> , it names <code>/a1/tf/jtb/bin/x</code> .
<code>memox</code>	name of a file in the current directory.

The UNIX file system provides special shorthand notations for the current directory and the *parent* directory of the current directory:

- `.` is the generic name of the current directory; `./memox` names the same file as `memox` if such a file exists in the current directory.
- `..` is the generic name of the parent directory of the current directory; if you type:

```
cd ..
```

then the parent directory of your current working directory will become your new current directory.

2.2 UNIX Processes

Beginners should skip this section on first reading.

An *image* is a computer execution environment, including contents of memory, register values, name of the current directory, status of open files, information recorded at login time, and various other items. A *process* is the execution of an image; most UNIX commands execute as separate processes. One process may spawn another using the `fork` system call, which duplicates the image of the original (*parent*) process. The new (*child*) process continues to execute the same program as the parent. The two images are identical, except that each program can determine whether it is executing as parent or child. Each program may continue execution of the image or may abandon it by issuing an `exec` system call, thus initiating execution of another program. In any case, each process is free to proceed in parallel with the other, although the parent most commonly issues a `wait` system call to suspend execution until a child terminates (`exits`).

Figure 1 illustrates these ideas. *Program A* is executing (as *process 1*) and wishes to run *program B*. It `forks` and spawns a child (*process 2*) that continues to run *program A*. The child abandons *A* by `execing B`, while the parent goes to sleep until the child `exits`.

A child inherits its parent's *open files*. This mechanism permits processes to share common input streams in various ways. In particular, an open file possesses a *pointer* that indicates a position in the file and is modified by various operations on the file; `read` and `write` system calls copy a requested number of bytes from and to a file, beginning at the position given by the current value of the pointer. As a side effect, the pointer is incremented by the number of bytes transferred, yielding the effect of sequential I/O; `lseek` can be used to obtain random-access I/O; it sets the pointer to an absolute position within the file, or to a position offset either from the end of the file or from the current pointer position.

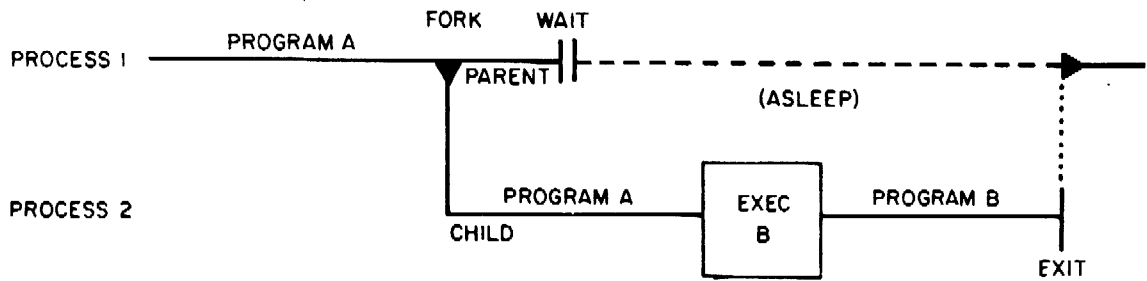


Figure 1

When a process terminates, it can set an eight-bit *exit status* (see `$?` in §3.4.4) that is available to its parent. This code is *usually* used to indicate success (zero) or failure (non-zero).

Signals indicate the occurrence of events that may have some impact on a process. A signal may be sent to a process by another process, from the terminal, or by UNIX itself. A child process inherits its parent's signals. For most signals, a process can arrange to be terminated on receipt of a signal, to ignore it completely, or to *catch* it and take appropriate action, as described in §4.4.11. For example, an `INTERRUPT` signal may be sent by depressing an appropriate key (*del*, *break*, or *rubout*). The action taken depends on the requirements of the specific program being executed:

- The shell invokes most commands in such a way that they immediately die when an interrupt is received. For example, the `pr` (print) command normally dies, allowing the user to terminate unwanted output.
- The shell *itself* ignores interrupts when reading from the terminal, because it should continue execution even when the user terminates a command like `pr`.
- The editor `ed` chooses to *catch* interrupts so that it can halt its current action (especially printing) without allowing itself to be terminated.

3. SHELL BASICS

The shell (i.e., the `sh` command) implements the command language visible to most UNIX users. It reads input from a terminal or a file and arranges for the execution of the requested commands. It is a program written in the C language [11]; it is *not* part of the operating system, but is an ordinary user program. The discussion below is adapted from [2,3,7,12].

3.1 Commands

A *simple command* is a sequence of non-blank arguments separated by blanks or tabs. The first argument (numbered *zero*) usually specifies the name of the command to be executed; any remaining arguments, with a few exceptions, are passed as arguments to that command. A command may be as simple as:

```
who
```

which prints the login names of users who are currently logged into the system. The following line requests the `pr` command to print files `a`, `b`, and `c`:

```
pr a b c
```

If the first argument of a command names a file that is *executable*¹ and is actually a compiled program, the shell (as parent) spawns a new (child) process that immediately executes that program. If the file is marked as being executable, but is not a compiled program, it is assumed to be a shell procedure, i.e., a file of ordinary text containing shell command lines, as well as possibly lines meant to be read by other programs. In this case, the shell spawns another instance of itself (a *sub-shell*) to read the file and execute the commands included in it. The shell *forks* to do this, but no *exec* call is made. The following command requests that the on-line *UNIX User's Manual* [7] entries that describe the *who* and *pr* commands be printed on the terminal:

```
man who pr
```

(Incidentally, the *man* command itself is actually implemented as a shell procedure.) From the user's viewpoint, compiled programs and shell procedures are invoked in exactly the same way. The shell determines which implementation has been used, rather than requiring the user to do so. This preserves the uniformity of invocation and the ease of changing the choice of implementation for a given command. The actions of the shell in executing any of these commands are illustrated in Figure 1 above.

3.2 How the Shell Finds Commands

The shell normally searches for commands in a way that permits them to be found in three distinct locations in the file structure. The shell first attempts to find the command (as given on the command line) in the current directory; if this fails, it prepends the string */bin* to the name, and, finally, */usr/bin*. The effect is to search, in order, the current directory, then the directory */bin*, and finally, */usr/bin*. For example, the *pr* and *man* commands are actually the files */bin/pr* and */usr/bin/man*, respectively. A more complex path name may be given, either to locate a file relative to the user's current directory, or to access a command via an absolute path name. If a command name *as given* begins with a */*, *./*, or *../* (e.g., */bin/sort* or *../cmd*), the prepending is *not* performed. Instead, a single attempt is made to execute the command as given.

This mechanism gives the user a convenient way to execute public commands and commands in or *near* the current directory, as well as the ability to execute *any* accessible command regardless of its location in the file structure. Because the current directory is usually searched first, anyone can possess a private version of a public command without affecting other users. Similarly, the creation of a new public command will not affect a user who already has a private command with the same name. The particular sequence of directories searched may be changed by resetting the *PATH* variable, as described in §3.4.2.

3.3 Generation of Argument Lists

Command arguments are very often file names. A list of file names can be automatically generated as arguments on a command line, by specifying a pattern that the shell matches against the file names in a directory.

Most characters in such a pattern match themselves, but there are also special *meta-characters* that may be included in a pattern. These special characters are: ***, which matches any string *including* the null string; *?*, which matches *any one* character; any sequence of characters enclosed within square brackets² (*[...]*), which matches *any one* of the enclosed characters; and any sequence of characters preceded by a *!* and enclosed within *[...]*, which matches

1. As indicated by an appropriate set of permission bits associated with that file.

2. Be warned that square brackets are also used below for another purpose: in descriptions of commands, they indicate that the enclosed argument is optional. See also §5.1 below.

any one character *other* than one of the enclosed characters. Inside square brackets, a pair of characters separated by a - includes in the set all characters lexically within the inclusive range of that pair, so that `[a-de]` is equivalent to `[abcde]`.

For example, `*` matches all file names in the current directory, `*temp*` matches all file names containing `temp`, `[a-f]*` matches all file names that begin with `a` through `f`, `[!0-9]` matches all single-character names other than the digits, and `*.c` matches all file names ending in `.c`, while `/a1/tf/bin/?` matches all single-character file names found in `/a1/tf/bin`. This capability saves much typing and, more importantly, makes it possible to organize information in large collections of small files that are named in disciplined ways.

Pattern-matching has some restrictions. If the first character of a file name is a period (`.`), it can be matched only by an argument that literally begins with a period. If a pattern does not match any file names, then the pattern itself is returned as the result of the match, for example:

```
echo *.c
```

will print:

```
*.c
```

if the current directory contains no files ending in `.c`.

Directory names should not contain the characters `*`, `?`, `[`, or `]`, because this may cause infinite recursion during pattern matching attempts.³

3.4 Shell Variables

The shell has several mechanisms for creating variables. A variable is a name representing a string value. Certain variables are usually referred to as *parameters*; these are the variables which are normally set only on a command line; there are *positional parameters* (§3.4.1) and *keyword parameters* (§4.1). Other variables are simply names to which the user or the shell itself may assign string values.

3.4.1 Positional Parameters. When a shell procedure is invoked, the shell implicitly creates *positional parameters*: the argument in position zero on the command line (the name of the shell procedure itself) is called `$0`, the first argument is called `$1`, and so on. The `shift` command (§4.3) may be used to access arguments in positions numbered higher than nine.

One can explicitly force values into these positional parameters by using the `set` command:

```
set abc def ghi
```

assigns the string `abc` to the first positional parameter (`$1`), `def` to the second (`$2`), and `ghi` to the third (`$3`); it also *unsets* `$4`, `$5`, etc., even if they were previously set. `$0` may not be assigned a value in this way—it always refers to the name of the shell procedure, or, in the login shell, to the name of the shell.

3.4.2 User-defined Variables. The shell also recognizes alphanumeric variables to which string values may be assigned. Positional parameters may not appear on the left-hand side of an assignment statement; they can only be set as described above. A simple assignment is of the form:

```
name=string
```

3. This is a bug that may be fixed in the future.

Thereafter, `$name` will yield the value *string*. A *name* is a sequence of letters, digits, and underscores that begins with a letter or an underscore. Note that no spaces surround the = in an assignment statement.

More than one assignment may appear in an assignment statement, but beware: *the shell performs the assignments from right to left*; the following command line results in the variable `a` acquiring the value `abc`:

```
a=$b b=abc
```

The following are examples of simple assignments. *Double quotes around the right-hand side* allow blanks, tabs, semi-colons, and new-lines to be included in *string*, while also allowing *variable substitution* (also known as *parameter substitution*) to occur; that is, references to positional parameters and other variable names that are prefaced by `$` are replaced by the corresponding values, if any; *single quotes inhibit variable substitution*:

```
MAIL=/usr/mail/gas
var="echo $1 $2 $3 $4"
stars=*****
asterisks='$stars'
```

The variable `var` has as its value the string consisting of the values of the first four positional parameters, separated by blanks. No quotes are needed around the string of asterisks being assigned to `stars` because pattern matching (expansion of `*`, `?`, `[...]`) does *not* apply in this context. Note that the value of `$asterisks` is the literal string `$stars`, *not* the string `*****`, because the single quotes inhibit substitution.

In assignments, blanks are not reinterpreted after variable substitution, so that the following example results in `$first` and `$second` having the same value:

```
first='a string with embedded blanks'
second=$first
```

In accessing the value of a variable, one may enclose the variable's name (or the digit designating the positional parameter) in braces `{ }` to delimit the variable name from any following string.⁴ In particular, if the character immediately following the name is a letter, digit, or underscore (digit only for positional parameters), then the braces are *required*:

```
a='This is a string'
echo "${a}ent test"
```

The following variables are used by the shell. Some of them are set by the shell, and all of them can be set and reset by the user:

- HOME** is initialized by the `login` program to the name of the user's *login directory*, i.e., the directory that becomes the current directory upon completion of a login; `cd` without arguments uses `$HOME` as the directory to switch to. Using this variable helps one to keep full path names out of shell procedures. This is a big help when the path name of your login directory is changed (e.g., to balance disk loads).
- MAIL** is the path name of a file where your mail is deposited. If `MAIL` is set, then the shell checks to see if anything has been added to the file it names and announces the arrival of new mail every time you return to command level (e.g., by leaving the editor). `MAIL` must be set by the user. (The presence of mail in the standard mail file is also announced at login, regardless of whether `MAIL` is set.)

4. See §4.4.7 and § 5.7 for other meanings of braces in the shell.

PATH is the variable that specifies where the shell is to look when it is searching for commands. Its value is an ordered list of directory path names separated by colons. A null character anywhere in that list represents the current directory. The shell initializes **PATH** to the list `:/bin:/usr/bin` where, by convention, a null character appears in front of the first colon. Thus if you wish to search your current directory last, rather than first, you would type:

```
PATH=/bin:/usr/bin::
```

where the two colons together represent a colon followed by a null followed by a colon, thus naming the current directory. A user often has a personal directory of commands (say, `$HOME/bin`) and causes it to be searched *before* the `/bin` and `/usr/bin` directories by using:

```
PATH=$HOME/bin:/bin:/usr/bin
```

The setting of **PATH** to other than the default value is normally done in a user's `.profile` file (§3.9.2).

CDPATH is the variable that specifies where the shell is to look when searching for the argument of the `cd` command whenever that argument is not null and does not begin with `/`, `./`, or `../` (see `cd(1)`, §2.1, and §4.5). The value of **CDPATH** is an ordered list of directory path names separated by colons. A null character anywhere in that list represents the current directory. By convention, if the list begins with a colon, a null character is assumed to precede that colon. Initially, **CDPATH** is *unset*, resulting in only the current directory being searched. Thus if you wish the `cd` command to first search your current directory and then your home directory, you would type:

```
CDPATH=$HOME
```

The setting of **CDPATH** to other than the default value is normally done in a user's `.profile` file (§3.9.2).

Note that if the `cd` command changes to a directory that is *not* a descendent of the current directory, it writes the full name of the new directory on the diagnostic output (§3.6.1, §3.6.2).

PS1 is the variable that specifies what string is to be used as the primary *prompt* string. If the shell is interactive, it prompts with the value of **PS1** when it expects input. The default value of **PS1** is `"$ "` (a `$` followed by a blank).

PS2 is the variable that specifies the secondary prompt string. If the shell expects more input when it encounters a new-line in its input, it will prompt with the value of **PS2**. The default value of **PS2** is `"> "` (a `>` followed by a blank).

IFS is the variable that specifies which characters are *internal field separators*. These are the characters the shell uses during blank interpretation. (If you want to parse some delimiter-separated data easily, you can set **IFS** to include that delimiter.) The shell initially sets **IFS** to include the blank, tab, and new-line characters.

3.4.3 Command Substitution. Any command line can be placed within grave accents (``...``) to capture the output of the command. This concept is known as *command substitution*. The command or commands enclosed between grave accents are first executed by the shell and then their output replaces the whole expression, grave accents and all. This feature is often combined with shell variables:

```
today=`date`
```

assigns the string representing the current date to the variable `today` (e.g., `Tue Nov 27 16:01:09 EST 1979`).

```
users=`who | wc -l`
```

saves the number of logged-in users in the variable `users`. Any command that writes to the standard output can be enclosed in grave accents. Grave accents (§3.5) may be nested; the inside sets must be escaped with `\`. For example:

```
logmsg=`echo Your login directory is `pwd``
```

Shell variables can also be given values indirectly by using the `read` command. The `read` command takes a line from the standard input (usually your terminal) and assigns consecutive words on that line to any variables named:

```
read first init last
```

will take an input line of the form:

```
G. A. Snyder
```

and have the same effect as if you had typed:

```
first=G.  init=A.  last=Snyder
```

The `read` command assigns any excess “words” to the last variable.

3.4.4 Predefined Special Variables.

Beginners should skip this section on first reading.

Several variables have special meanings; the following are set *only* by the shell:

\$# records the number of *positional* arguments passed to the shell, not counting the name of the shell procedure itself; **\$#** thus yields the number of the highest-numbered positional parameter that is set. Thus, `sh x a b c` sets **\$#** to 3. One of its primary uses is in checking for the presence of the required number of arguments:

```
if test $# -lt 2
then
    echo 'two or more args required'; exit
fi
```

\$? is the exit status (also referred to as *return code*, *exit code*, or *value*) of the last command executed. Its value is a decimal string. Most UNIX commands return 0 to indicate successful completion. The shell itself returns the current value of **\$?** as its exit status.

\$\$ is the process number of the current process; because process numbers are unique among all existing processes, this string of up to five digits is often used to generate unique names for temporary files. UNIX provides no mechanism for the automatic creation and deletion of temporary files: a file exists until it is explicitly removed. Temporary files are generally undesirable objects: the UNIX pipe mechanism is far superior for many applications. However, the need for uniquely-named temporary files does occasionally occur. The following example also illustrates the recommended practice of creating temporary files in a directory used only for that purpose:

```
temp=$HOME/temp/$$      # use current process number
ls > $temp              # to form unique temp file
    commands, some of which use $temp, go here
rm $temp                # clean up at end
```

\$! is the process number of the last process run in the background (using `&`—see §4.4). Again, this is a string of up to five digits.

\$- is a string consisting of names of execution flags (§3.9.3, §4.7) currently turned on in the shell; **\$-** might have the value `xv` if you are tracing your output.

3.5 Quoting Mechanisms

Many characters have a special meaning to the shell which is sometimes necessary to conceal. Single quotes (' ') and double quotes (" ") surrounding a string, or backslash (\) before a single character, provide this function in somewhat different ways. (Grave accents (` `) are sometimes called *back quotes*, but are used only for command substitution (§3.4.3) in the shell and do not hide special meanings of any characters.)

Within single quotes, all characters (except ' itself) are taken literally, with any special meaning removed. Thus:

```
stuff='echo $? $*; ls * | wc'
```

results only in the string `echo $? $*; ls * | wc` being assigned to the variable `stuff`, but *not* in any other commands being executed.

Within double quotes, the special meaning of certain characters does persist, while all other characters are taken literally. The characters that retain their special meaning are \$, `, and " itself. Thus, within double quotes, variables are expanded and command substitution takes place; however, any commands in a command substitution are not affected by double quotes outside of the grave accents, so that characters such as * retain their special meaning.

To hide the special meaning of \$, `, and " within double quotes, you can precede these characters with a backslash (\). Outside of double quotes, preceding a character with \ is equivalent to placing single quotes around that character. A \ followed by a new-line causes that new-line to be ignored, thus allowing continuation of long command lines.

3.6 Redirection of Input and Output

In general, most commands neither know nor care whether their input (output) is coming from (going to) a terminal or a file. Thus, a command can be used conveniently either at a terminal or in a pipeline (see §3.7). A few commands vary their actions depending on the nature of their input or output, either for efficiency's sake, or to avoid useless actions (such as attempting random-access I/O on a terminal).

3.6.1 Standard Input and Standard Output. When a command begins execution, it usually expects that three files are already open: a *standard input*, a *standard output*, and a *diagnostic (error) output*. A number called a *file descriptor* is associated with each of these files; by convention, file descriptor 0 is associated with standard input, file descriptor 1 with standard output, and file descriptor 2 with diagnostic output. A child process normally inherits these files from its parent; all three files are initially connected to the terminal (0 to the keyboard, 1 and 2 to the printer or screen). The shell permits them to be redirected elsewhere before control is passed to an invoked command. An argument to the shell of the form `< file` or `> file` opens the specified file as the standard input or output, respectively (in the case of output, destroying the previous contents of *file*, if any). An argument of the form `>> file` directs the standard output to the end of *file*, thus providing a way to *append* data to it without destroying its existing contents. In either of the two output cases, the shell creates *file* if it does not already exist (thus `> output` alone on a line creates a zero-length file). The following appends to file `log` the list of users who are currently logged on:

```
who >> log
```

Such redirection arguments are only subject to variable and command substitution; neither blank interpretation nor pattern matching of file names occurs after these substitutions. Thus:

```
echo 'this is a test' > *.ggg
```

and:

```
cat < ?
```

will produce, respectively, a one-line file named `*.ggg` (a rather disastrous name for a file) and an error message (unless you have a file named `?`, which is also *not* a wise choice for a file name—see end of §3.3).

3.6.2 Diagnostic and Other Outputs. Diagnostic output from UNIX commands is traditionally directed to the file associated with file descriptor 2. (There is often a need for an error output file that is different from standard output so that error messages do not get lost down pipelines—see §3.7.) One can redirect this error output to a file by immediately prepending the number of the file descriptor (i.e., 2 in this case) to either output redirection symbol (`>` or `>>`). The following line will append error messages from the `cc` command to file `ERRORS`:

```
cc testfile.c 2>> ERRORS
```

Note that the file descriptor number must be prepended to the redirection symbol *without* any intervening blanks or tabs; otherwise, the number will be passed as an argument to the command.

This method may be generalized to allow one to redirect output associated with any of the first ten file descriptors (numbered 0–9) so that, for instance, if `cmd` puts output on file descriptor 9, the following line will capture that output in file `savedata`:

```
cmd 9> savedata
```

A command often generates standard output and error output, and might even have some other output, perhaps a data file. In this case, one can redirect independently all the different outputs. Suppose that `cmd` directs its standard output to file descriptor 1, its error output to file descriptor 2, and builds a data file on file descriptor 9. The following would direct each of these three outputs to a different file:

```
cmd > standard 2> error 9> data
```

Other forms of input/output redirection are described in §4.4.8, §4.4.9, and §5.6.

3.7 Command Lines and Pipelines

A sequence of one or more commands separated by `|` (or `^`) make up a *pipeline*. In a pipeline consisting of more than one command, each command is run as a separate process connected to its neighbor(s) by *pipes*, i.e., the *output* of each command (except the last one) becomes the *input* of the next command in line. A *filter* is a command that reads its standard input, transforms it in some way, then writes it as its standard output. A pipeline normally consists of a series of filters. Although the processes in a pipeline are permitted to execute in parallel, they are synchronized to the extent that each program needs to read the output of its predecessor. Many commands operate on individual lines of text, reading a line, processing it, writing it out, and looping back for more input. Some must read larger amounts of data before producing output; `sort` is an example of the extreme case that requires all input to be read before any output is produced.

The following is an example of a typical pipeline: `nroff` is a text formatter whose output may contain reverse line motions; `col` converts these motions to a form that can be printed on a terminal lacking reverse-motion capability; `greek` is used to adapt the output to a specific terminal, here specified by `-Thp`. The flag `-cm` indicates one of the commonly used formatting options, and `text` is the name of the file to be formatted:

```
nroff -cm text | col | greek -Thp
```

3.8 Examples

The following examples illustrate the variety of effects that can be obtained by combining a few commands in the ways described above. It may be helpful to try these examples at a terminal:

- `who`
Print (on the terminal) the list of logged-in users.
- `who >> log`
Append the list of logged-in users to the end of file `log`.
- `who | wc -l`
Print the number of logged-in users. (The argument to `wc` is *minus ell.*)
- `who | pr`
Print a paginated list of logged-in users.
- `who | sort`
Print an alphabetized list of logged-in users.
- `who | grep pw`
Print the list of logged-in users whose login names contain the string `pw`.
- `who | grep pw | sort | pr`
Print an alphabetized, paginated list of logged-in users whose login names contain the string `pw`.
- `{ date; who | wc -l; } >> log`
Append (to file `log`) the current date followed by the count of logged-in users (see §4.4.7 for the meaning of `{ . . . }` in this context).
- `who | sed 's/ .*//' | sort | uniq -d`
Print only the login names of all users who are logged in more than once.

The `who` command does not *by itself* provide options to yield all these results—they are obtained by combining `who` with other commands. Note that `who` just serves as the data source in these examples. As an exercise, replace `who |` by `< /etc/passwd` in the above examples to see how a file can be used as a data source in the same way. Notice that redirection arguments may appear anywhere on the command line.

3.9 Changing the State of the Shell and the `.profile` File

The state of a given instance of the shell includes the values of positional parameters (§3.4.1), user-defined variables (§3.4.2), environment variables (§4.1), modes of execution (§4.7), and the current working directory.

The state of a shell may be altered in various ways. These include the `cd` command, several flags that can be set by the user, and a file in one's login directory called `.profile` that is treated specially by the shell.

3.9.1 Cd. The `cd` command changes the current directory to the one specified as its argument. This can (and should) be used to change to a convenient place in the directory structure; `cd` is often combined with `()` to cause a sub-shell to change to a different directory and execute a group of commands without affecting the original shell. The first sequence below extracts the component files of the archive file `/a1/tf/q.a` and places them in whatever directory is the current one; the second places them in directory `/a1/tf`:

```
ar x /a1/tf/q.a
(cd /a1/tf; ar x q.a)
```

3.9.2 The `.profile` File. When you log in, the shell is invoked to read your commands. First, however, the shell checks to see if a file named `/etc/profile` exists on your UNIX system, and if it does, commands are read from it; `/etc/profile` is used by system administrators to set up variables needed by *all* users. Type:

```
cat /etc/profile
```

to see what your system administrator has already done for you. After this, the shell proceeds to see if you have a file named `.profile` in your login directory. If so, commands are read and executed from it. For a sample `.profile`, see *profile(5)*. Finally, the shell is ready to read commands from your standard input—usually the terminal.

3.9.3 Execution Flags: `set`. The `set` command provides the capability of altering several aspects of the behavior of the shell by setting certain *shell flags*. In particular, the `x` and `v` flags may be useful from the terminal. Flags may be `set` by typing, for example:

```
set -xv
```

(to turn on flags `x` and `v`). The same flags may be turned *off* by typing:

```
set +xv
```

These two flags have the following meaning:

- v Input lines are printed as they are read by the shell. This flag is particularly useful for isolating syntax errors. The commands on each input line are executed after that input line is printed.
- x Commands and their arguments are printed as they are executed. (Shell control commands, such as `for`, `while`, etc., are not printed, however.) Note that `-x` causes a trace of *only* those commands that are actually executed, whereas `-v` prints each line of input until a syntax error is detected.

The `set` command is also used to set these and other flags within shell procedures (see §4.7).

4. USING THE SHELL AS A COMMAND: SHELL PROCEDURES

4.1 A Command's Environment

All the variables (with their associated values) that are known to a command at the beginning of execution of that command constitute its *environment*. This environment includes variables that the command inherits from its parent process and variables specified as *keyword parameters* on the command line that invokes the command.

The variables that a shell passes to its child processes are those that have been named as arguments to the `export` command. The `export` command places the named variables in the environments of both the shell *and* all its future child processes.

Keyword parameters are variable-value pairs that appear in the form of assignments, normally *before* the procedure name on a command line (but see also `-k` flag in §4.7). Such variables are placed in the environment of the procedure being invoked. For example:

```
# key_command
echo $a $b
```

is a simple procedure that echoes the values of two variables; if it is invoked as:

```
a=key1 b=key2 key_command
```

then the output is:

```
key1 key2
```

A procedure's keyword parameters are *not* included in the argument count `$#` (§3.4.4).

A procedure may access the value of any variable in its environment; however, if changes are made to the value of a variable, these changes are *not* reflected in the environment—they are local to the procedure in question. In order for these changes to be placed in the environment that the procedure passes to *its* child processes, the variable must be named as an argument to the `export` command within that procedure (but see §4.2 below). To obtain a list of variables that have been made `exportable` from the current shell, type:

```
export
```

(You will also get a list of variables that have been made `readonly`—see §4.5 below.) To get a list of name-value pairs in the current environment, type:

```
env
```

4.2 Invoking the Shell

The shell is an ordinary command and may be invoked in the same way as other commands:

```
sh proc [ arg ... ]   A new instance of the shell is explicitly invoked to read proc.
                        Arguments, if any, can be manipulated as described in §4.3.

sh -v proc [ arg ... ] This is equivalent to putting set -v at the beginning of proc.
                        Similarly for the x, e, u, and n flags (§3.9.3, §4.7).

proc [ arg ... ]      If proc is marked executable, and is not a compiled, executable program,
                        the effect is similar to that of sh proc [ args ... ]. An
                        advantage of this form is that proc may be found by the search procedure
                        described in §3.2 and §3.4.2. Also, variables that have been
                        exported in the shell will still be exported from proc when this
                        form is used (because the shell only forks to read commands
                        from proc). Thus any changes made within proc to the values of
                        exported variables will be passed on to subsequent commands
                        invoked from within proc.
```

There are several shell invocation flags that are sometimes useful for more advanced shell programming. They are described in §5.8.

4.3 Passing Arguments to the Shell; `shift`

When a command line is scanned, any character sequence of the form `$n` is replaced by the n th argument to the shell, counting the name of the shell procedure itself as `$0`. This notation permits direct reference to the procedure name and to as many as nine positional parameters (§3.4.1). Additional arguments can be processed using the `shift` command or by using a `for` loop (§4.4.4).

The `shift` command shifts arguments to the left; i.e., the value of `$1` is thrown away, `$2` replaces `$1`, `$3` replaces `$2`, etc.; the highest-numbered positional parameter becomes *unset*. (`$0` is *never* shifted.) The command `shift n` is a shorthand notation for n consecutive shifts; `shift 0` does nothing. For example, consider the shell procedure `ripple` below: `echo` writes its arguments to the standard output; `while` is discussed in §4.4.3 (it is a looping command); lines that begin with `#` are comments.

```
#       ripple command
while test $# != 0
do
    echo $1 $2 $3 $4 $5 $6 $7 $8 $9
    shift
done
```

If the procedure were invoked by:

```
ripple a b c
```

it would print:

```
a b c
b c
c
```


The notation `$$` causes substitution of *all* positional parameters except `$0`. Thus, the `echo` line in the `ripple` example above could be written more compactly as:

```
echo $$
```

These two `echo` commands are *not* equivalent: the first prints at most nine positional parameters; the second prints *all* of the current positional parameters. The `$$` notation is more concise and less error-prone. One obvious application is in passing an arbitrary number of arguments to a command such as the `nroff` text formatter:

```
nroff -h -rW120 -T450 -cm $$
```

It is important to understand the sequence of actions used by the shell in scanning command lines and substituting arguments. The shell first reads input up to a new-line or semicolon, and then parses that much of the input. Variables are replaced by their values and then command substitution (via *grave accents*) is attempted. I/O redirection arguments are detected, acted upon, and deleted from the command line. Next the shell scans the resulting command line for *internal field separators*, that is, for any characters specified by `IFS` to break the command line into distinct arguments; *explicit* null arguments (specified by `"` or `'`) are retained, while *implicit* null arguments resulting from evaluation of variables that are null or not set are removed. Then file name generation occurs, with all meta-characters being expanded. The resulting command line is executed by the shell.

Sometimes, one builds command lines inside a shell procedure. In this case, one might want to have the shell rescan the command line after all the initial substitutions and expansions are done. The special command `eval` is available for this purpose; `eval` takes a command line as its argument and simply rescans the line, performing any variable or command substitutions that are specified. Consider the following (simplified) situation:

```
command=who
output=' | wc -l'
eval $command $output
```

This segment of code results in the pipeline `who | wc -l` being executed.

The output of `eval` cannot be redirected; uses of `eval` can, however, be nested.

4.4 Control Commands

The shell provides several flow-of-control commands that are useful in creating shell procedures. To explain them, we first need a few definitions.

A *simple command* is as defined in §3.1. I/O redirection arguments can appear in a simple command line and are passed to the shell, *not* to the command.

A *command* is a simple command or any of the shell control commands described below. A *pipeline* is a sequence of one or more commands separated by `|`. (For historical reasons, `^` is a synonym for `|` in this context.) The standard output of each command but the last in a pipeline is connected (by a *pipe*(2)) to the standard input of the next command. Each command in a pipeline is run separately; the shell waits for the last command to finish. The exit status of a pipeline is non-zero if the exit status of either the first or last process in the pipeline is non-zero. (This is a bit weird, and may be changed in the future.)

A *command list* is a sequence of one or more pipelines separated by `;`, `&`, `&&`, or `||`, and optionally terminated by `;` or `&`. A semicolon (`;`) causes sequential execution of the previous pipeline (i.e., the shell waits for the pipeline to finish before reading the next pipeline), while `&` causes asynchronous execution of the preceding pipeline; both sequential and asynchronous execution are thus allowed. An asynchronous pipeline continues execution until it terminates voluntarily, or until its processes are killed. In the first example below, the shell executes `who`, waits for it to terminate, then executes `date` and waits for it to terminate; in the second example, the shell invokes both commands in order, but does not wait for either one to finish.

Figure 2 shows the actions of the shell involved in executing these two command lists:

```
who >log; date
who >log& date&
```

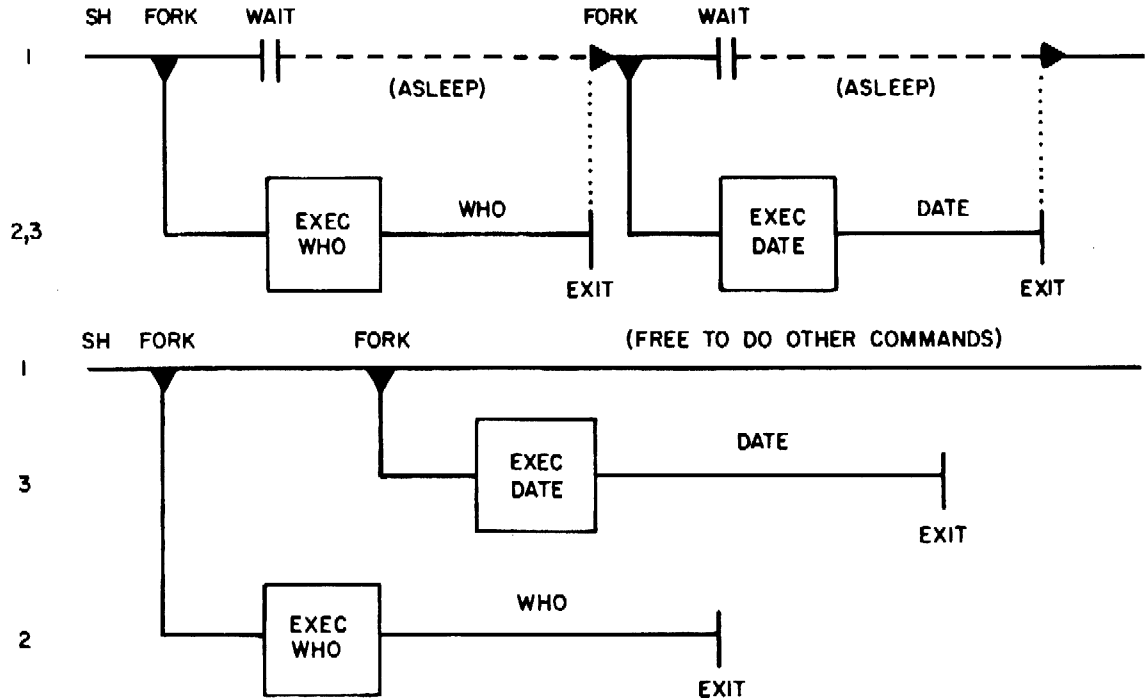


Figure 2

More typical uses of & include off-line printing, background compilation, and generation of jobs to be sent to other computers. For example, if you type:

```
nohup cc prog.c&
```

you may continue working while the C compiler runs in the background. A command line ending with & is immune to interrupts and quits, but it is wise to make it immune to hang-ups as well. The nohup command is used for this purpose. Without nohup, if you hang up while cc in the above example is still executing, cc will be killed and your output will disappear.

The & operator should be used with restraint, especially on heavily-loaded systems. Other users will not consider you a good citizen if you start up a large number of simultaneous, asynchronous processes without a compelling reason for doing so.

The && and !! operators, which are of equal precedence (but lower than & and !), cause conditional execution of pipelines. In cmd1 !! cmd2, cmd1 is executed and its exit status examined. Only if cmd1 fails (i.e., has a non-zero exit status) is cmd2 executed. This is thus a more terse notation for:

```
if      cmd1
      test $? != 0
then
      cmd2
fi
```

See writemail in §6 for an example of use of !!.

The `&&` operator yields the complementary test: in `cmd1 && cmd2`, the second command is executed only if the first succeeds (has a zero exit status). In the sequence below, each command is executed in order until one fails:

```
cmd1 && cmd2 && cmd3 && ... && cmdn
```

A simple command in a pipeline may be replaced by a command list enclosed in either parentheses or braces. The output of all the commands so enclosed is combined into one stream that becomes the input to the next command in the pipeline. The following line prints *two* separate documents in a way similar to that shown in a previous example (§3.7):

```
{ nroff -cm text1; nroff -cm text2; } | col | greek -Thp
```

See §4.4.7 for further details on command grouping.

All of the following commands are formally described in *sh(1)*.

4.4.1 Structured Conditional: `if`. The shell provides an `if` command. The simplest form of the `if` command is:

```
if command list
then command list
fi
```

The *command list* following `if` is executed and if the last command in the list has a *zero* exit status, then the *command list* that follows `then` is executed; `fi` indicates the end of the `if` command.

In order to cause an alternative set of commands to be executed in the case where the *command list* following `if` has a *non-zero* exit status, one may add an `else`-clause to the form given above. This results in the following structure:

```
if command list
then command list
else command list
fi
```

Multiple tests can be achieved in an `if` command by using the `elif` clause. For example:

```
if      test -f "$1" # is $1 a file?
then    pr $1
elif    test -d "$1" # else, is $1 a directory?
then    (cd $1; pr *)
else    echo $1 is neither a file nor a directory
fi
```

The above example is executed as follows: if the value of the first positional parameter is a file name, then print that file; if not, then check to see if it is the name of a directory. If so, change to that directory and print all the files there. Otherwise, `echo` the error message.

The `if` command may be nested (but be sure to end each one with a `fi`). The new-lines in the above examples of `if` may be replaced by semicolons.

The exit status of the `if` command is the exit status of the last command executed in any `then` clause or `else` clause. If no such command was executed, `if` returns a zero exit status.

4.4.2 Multi-way Branch: `case`. A multiple way branch is provided by the `case` command. The basic format of `case` is:

```

case string in
pattern) command list;;
:
pattern) command list;;
esac

```

The shell tries to match *string* against each pattern in turn, using the same pattern-matching conventions as in file-name generation (§3.3). If a match is found, the *command list* following the matched pattern is executed; the `;;` serves as a break out of the `case` and is required after each command list except the last. Note that only one pattern is ever matched, and that matches are attempted in order, so that if `*` is the first pattern in a `case`, no other patterns will ever be looked at.

More than one pattern may be associated with a given command list by specifying alternate patterns separated by `|`. For example:

```

case $i in
*.c)      cc $i
          ;;
*.h|*.sh)      # do nothing
          ;;
*)          echo "$i of unknown type"
          ;;
esac

```

In the above example, no action is taken for the second set of patterns because the *null* command is specified; `*` is used as a default pattern, because it matches any word.

The exit status of `case` is the exit status of the last command executed in the `case` command. If no commands were executed, then `case` has a zero exit status.

4.4.3 Conditional Looping: `while` and `until`. A `while` command has the general form:

```

while command list
do
    command list
done

```

The commands in the first *command list* are executed, and if the exit status of the last command in that list is zero, then the commands in the second list are executed. This sequence is repeated as long as the exit status of the first *command list* is zero. A loop can be executed as long as the first *command list* returns a non-zero exit status by replacing `while` with `until`.

Any new-line in the above example may be replaced by a semicolon. The exit status of a `while` (`until`) command is the exit status of the last command executed in the *second* command list. If no such command is executed, `while` (`until`) has exit status zero.

4.4.4 Looping over a List: `for`. Often, one wishes to perform some set of operations for each in a set of files, or execute some command once for each of several arguments. The `for` command can be used to accomplish this. The `for` command has the format:

```

for variable in word list
do
    command list
done

```

where *word list* is a list of strings separated by blanks. The commands in the *command list* are executed once for each word in *word list*. *Variable* takes on as its value each word from *word list*, in turn; *word list* is fixed after it is evaluated the first time. For example, the following `for` loop will cause each of the C source files `xec.c`, `cmd.c`, and `word.c` in the current directory to be `diff`d with a file of the same name in the directory `/usr/src/cmd/sh`:

```
for cfile in xec cmd word
do      diff $cfile.c /usr/src/cmd/sh/$cfile.c
done
```

One can omit the “in *word list*” part of a `for` command; this will cause the current set of positional parameters to be used in place of *word list*. This is very convenient when one wishes to write a command that performs the same set of commands for each of an unknown number of arguments. See `null` in §6 for an example of this feature.

4.4.5 Loop Control: `break` and `continue`. The `break` command can be used to terminate execution of a `while`, `until`, or a `for` loop; `continue` requests the execution of the next iteration of the loop. These commands are effective only when they appear between `do` and `done`.

The `break` command terminates execution of the smallest (i.e., innermost) enclosing loop, causing execution to resume after the nearest following unmatched `done`. Exit from *n* levels is obtained by `break n`.

The `continue` command causes execution to resume at the nearest enclosing `while`, `until`, or `for`, i.e., the one that begins the innermost loop containing the `continue`; one can also specify an argument *n* to `continue` and execution will resume at the *n*th enclosing loop:

```
# This procedure is interactive; 'break' and 'continue'
# commands are used to allow the user to control data entry.
while true
do      echo "Please enter data"
        read response
        case "$response" in
            "done") break      # no more data
                ;;
            "")      continue
                ;;
            *)
                    process the data here
                ;;
        esac
done
```

4.4.6 End-of-file and `exit`. When the shell reaches the end-of-file, it terminates execution, returning to its parent the exit status of the last command executed prior to the end-of-file. The `exit` command simply reads to the end-of-file and returns, setting the exit status to the value of its argument, if any. Thus, a procedure can be terminated “normally” by using `exit 0`.

4.4.7 Command Grouping: Parentheses and Braces. There are two methods for grouping commands in the shell. As mentioned in §3.9.1, parentheses `()` cause the shell to spawn a *sub-shell* that reads the enclosed commands. Both the right and left parentheses are recognized *wherever* they appear in a command line—they can appear as literal parentheses *only* by being quoted. For example, if you type `garble(stuff)` the shell interprets this as four separate words: `garble`, `(`, `stuff`, and `)`.

This sub-shell capability is useful if one wishes to perform some operations without affecting the values of variables in the current shell, or to temporarily change directory and execute some commands in the new directory without having to explicitly return to the current directory. The current environment is passed to the sub-shell and variables that are exported in the current shell are also exported in the sub-shell. Thus:

```
current=`pwd`; cd /usr/docs/sh_tut;
nohup mm -Tlp sc_? | lpr& cd $current
```

and:

```
(cd /usr/docs/sh_tut; nohup mm -Tlp sc_? | lpr&)
```

accomplish the same result: a copy of this tutorial is printed on the line printer; however, the second example automatically puts you back in your original working directory. In the second example above, blanks or new-lines surrounding the parentheses are allowed but not necessary. The shell will prompt with `$PS2` if a `)` is expected. See also the example in §3.9.1.

Braces `{ }` may also be used to group commands together.⁵ Both the left and the right brace are recognized *only* if they appear as the first (unquoted) word of a command. The opening brace `{` may be followed by a new-line (in which case the shell will prompt for more input). Unlike in the case of parentheses, no sub-shell is spawned for braces; the enclosed commands are simply read by the shell. The braces are convenient when you wish to use the (sequential) output of several commands as input to one command; see the last example in §4.4 above.

The exit status of a set of commands grouped by either parentheses or braces is the exit status of the last enclosed executed command.

4.4.8 Input/Output Redirection and Control Commands. The shell normally does not *fork* when it recognizes the *control* commands (other than parentheses) described above. However, each command in a pipeline is run as a separate process in order to direct input (output) to (from) each command. Also, when redirection of input/output is specified explicitly for a control command, a separate process is spawned to execute that command. Thus, when *if*, *while*, *until*, *case*, or *for* is used in a pipeline consisting of more than one command, the shell *forks* and a sub-shell runs the control command. This has certain implications; the most noticeable one is that *any changes made to variables within the control command are not effective once that control command finishes* (similar to the effect of using parentheses to group commands). The control commands run slightly slower when redirection is specified.

✎ *Beginners should skip to Section 4.5 on first reading.*

4.4.9 In-line Input Documents. Upon seeing a command line of the form:

```
command << eofstring
```

where *eofstring* is any arbitrary string, the shell will take the subsequent lines as the standard input of *command* until a line is read consisting only of *eofstring* (possibly preceded by one or more tab characters). By appending a minus (`-`) to `<<`, leading tab characters are deleted from each line of the input document before the shell passes the line to *command*.

The shell creates a temporary file containing the input document and performs variable and command substitution (§3.4.3) on its contents before passing it to the command. Pattern matching on file names is performed on the arguments of command lines in command substitutions. In order to prohibit all substitutions, one may quote any character of *eofstring*:⁶

```
command << \eofstring
```

The in-line input document feature is especially useful for small amounts of input data (e.g., an editor “script”), where it is more convenient to place the data in the shell procedure than to keep it in a separate file. For instance, one could type:

5. See §3.4.2 and §5.7 for other meanings of braces in the shell.

6. Typically, *eofstring* consists of a single character; `!` is often used for this purpose.

```

cat <<- xyz
    This message will be printed on the
    terminal with leading tabs removed.
xyz

```

This in-line input document feature is most useful in shell procedures. See `edfind`, `edlast`, and `mmt` in §6. Note that in-line input documents may *not* appear within grave accents.⁷

4.4.10 Transfer to Another File and Back: the Dot (.) Command. A command line of the form:

```
. proc
```

causes the shell to read commands from *proc* without spawning a new process. Changes made to variables in *proc* are in effect after the *dot* command finishes. This is thus a good way to gather a number of shell variable initializations into one file. Note that an `exit` command in a file executed in this manner will cause an exit from your current shell; if you are at login level, you will be logged out.

4.4.11 Interrupt Handling: trap. As noted in §2.2, a program may choose to *catch* an interrupt from the terminal, *ignore* it completely, or be terminated by it. Shell procedures can use the `trap` command to obtain the same effects.

```
trap arg signal-list
```

is the form of the `trap` command, where *arg* is a string to be interpreted as a command list and *signal-list* consists of one or more signal numbers (as described in *signal(2)*). The commands in *arg* are scanned at least once, when the shell first encounters the `trap` command. Because of this, it is usually wise to use single rather than double quotes to surround these commands. The former inhibit immediate command and variable substitution; this becomes important, for instance, when one wishes to remove temporary files and the names of those files have not yet been determined when the `trap` command is first read by the shell. The following procedure will print the name of the current directory on the file `errdirect` when it is interrupted, thus giving the user information as to how much of the job was done:

```

trap 'echo `pwd` >errdirect' 2 3 15
for i in /bin /usr/bin /usr/gas/bin
do
    cd $i
    commands to be executed in directory $i here
done

```

while the same procedure with double (rather than single) quotes (`trap "echo `pwd` >errdirect" 2 3 15`) will, instead, print the name of the directory from which the procedure was executed.

Signal 11 (SEGMENTATION VIOLATION) may never be trapped, because the shell itself needs to catch it to deal with memory allocation. Zero is not a UNIX signal, but is effectively interpreted by the `trap` command as a signal generated by exiting from a shell (either via an `exit` command, or by “falling through” the end of a procedure). If *arg* is not specified, then the action taken upon receipt of any of the signals in *signal-list* is reset to the default system action. If *arg* is an explicit null string (‘ ’ or “”), then the signals in *signal-list* are *ignored* by the shell.

7. This is a implementation bug that should (and may) be fixed eventually.

The most frequent use of `trap` is to assure removal of temporary files upon termination of a procedure. The second example of §3.4.4 would be written more typically as follows:

```
temp=$HOME/temp/$$
trap 'rm $temp; trap 0; exit' 0 1 2 3 15
ls > $temp
    commands, some of which use $temp, go here
```

In this example, whenever signals 1 (HANGUP), 2 (INTERRUPT), 3 (QUIT), or 15 (SOFTWARE TERMINATION) are received by the shell procedure, or whenever the shell procedure is about to exit, the commands enclosed between the single quotes will be executed. The `exit` command must be included, or else the shell continues reading commands where it left off when the signal was received. The `trap 0` turns off the original trap on exits from the shell, so that the `exit` command does not reactivate the execution of the trap commands.

Sometimes it is useful to take advantage of the fact that the shell continues reading commands after executing the trap commands. The following procedure takes each directory in the current directory, changes to it, prompts with its name, and executes commands typed at the terminal until an end-of-file (*control-d*) or an interrupt is received. An end-of-file causes the `read` command to return a non-zero exit status, thus terminating the `while` loop and restarting the cycle for the next directory; the entire procedure is terminated if interrupted when waiting for input, but during the execution of a command, an interrupt terminates *only* that command:

```
dir=`pwd`
for i in *
do
    if test -d $dir/$i
    then
        cd $dir/$i
        while echo "$i:"
            trap exit 2
            read x
        do
            trap : 2          # ignore interrupts
            eval $x
        done
    fi
done
```

Several traps may be in effect at the same time; if multiple signals are received simultaneously, they are serviced in ascending order. To check what traps are currently set, type:

```
trap
```

It is important to understand some things about the way in which the shell implements the `trap` command in order not to be surprised. When a signal (other than 11) is received by the shell, it is passed on to whatever child processes are currently executing. When those (synchronous) processes terminate, normally or abnormally, the shell *then* polls any traps that happen to be set and executes the appropriate `trap` commands. This process is straightforward, except in the case of traps set at the command (outermost, or login) level; in this case, it is possible that no child process is running, so the shell waits for the termination of the first process spawned *after* the signal is received before it polls the traps.

For internal commands, the shell normally polls traps on completion of the command; an exception to this rule is made for the `read` command, for which traps are serviced immediately, so that `read` can be interrupted while waiting for input.

4.5 Special Shell Commands

There are several special commands that are *internal* to the shell (some of which have already been mentioned). These commands should be used in preference to other UNIX commands whenever possible, because they are, in general, faster and more efficient. The shell does not fork to execute these commands, so no additional processes are spawned; the trade-off for this efficiency is that redirection of input/output is not allowed for most of these special commands.

Several of the special commands have already been described in §4.4 because they affect the flow of control. They are `break`, `continue`, `exit`, `dot` (`.`), and `trap`. The `set` command described in §3.4.1 and §3.9.3 is also a special command. Descriptions of the remaining special commands are given here:

<code>:</code>	The <i>null</i> command; this command does nothing; the exit status is zero (<i>true</i>). <i>Beware</i> : any arguments to the null command are parsed for syntactic correctness; when in doubt, quote such arguments. Parameter substitution takes place, just as in other commands.
<code>cd arg</code>	Make <i>arg</i> the current directory. If <i>arg</i> does not begin with <code>/</code> , <code>./</code> , or <code>../</code> , <code>cd</code> uses the <code>CDPATH</code> shell variable (§3.4.2) to locate a parent directory that contains the directory <i>arg</i> . If <i>arg</i> is not a directory, or the user is not authorized to access it, a non-zero exit status is returned. Specifying <code>cd</code> with no <i>arg</i> is equivalent to typing <code>cd \$HOME</code> .
<code>exec arg ...</code>	If <i>arg</i> is a command, then the shell executes it without forking. No new process is created. Input/output redirection arguments <i>are</i> allowed on the command line. If <i>only</i> input/output redirection arguments appear, then the input/output of the shell itself is modified accordingly. See <code>merge</code> in §6 for an example of this use of <code>exec</code> .
<code>newgrp arg ...</code>	The <code>newgrp(1)</code> command is executed, replacing the shell; <code>newgrp</code> in turn spawns a new shell; see <code>newgrp(1)</code> . <i>Beware</i> : Only variables in the environment will be known in the shell that is spawned by the <code>newgrp</code> command. Any variables that were <code>exported</code> will no longer be marked as such.
<code>read var ...</code>	One line (up to a new-line) is read from standard input and the first word is assigned to the first variable, the second word to the second variable, and so on. All left-over words are assigned to the <i>last</i> variable. The exit status of <code>read</code> is zero unless an end-of-file is read.
<code>readonly var ...</code>	The specified variables are made <code>readonly</code> so that no subsequent assignments may be made to them. If no arguments are given, a list of all <code>readonly</code> and of all <code>exported</code> variables is given.
<code>test</code>	A conditional expression is evaluated. More details are given in §5.1 below.
<code>times</code>	The accumulated user and system times for processes run from the current shell are printed.
<code>umask nnn</code>	The user file creation mask is set to <i>nnn</i> ; see <code>umask(2)</code> for details. If <i>nnn</i> is omitted, then the current value of the mask is printed.
<code>ulimit n</code>	This command imposes a limit of <i>n</i> blocks on the size of files written by the shell and its child processes (files of any size may be read). If <i>n</i> is omitted, the current value of this limit is printed. The default value for <i>n</i> varies from on installation to another.
<code>wait n</code>	The shell waits for the child process whose process number is <i>n</i> to terminate; the exit status of the <code>wait</code> command is that of the process waited on. If <i>n</i> is omitted or is not a child of the current shell, then <i>all</i> currently active processes are waited for and the return code of the <code>wait</code> command is zero.

4.6 Creation and Organization of Shell Procedures

A shell procedure can be created in two simple steps: first, one builds an ordinary text file; then one changes its *mode* to make it *executable*, thus permitting it to be invoked by *proc args*, rather than by *sh proc args*. The second step may be omitted for a procedure to be used once or twice and then discarded, but is recommended for longer-lived ones. Here is the entire input needed to set up a simple procedure (the executable part of `draft` in §6):

```
ed
a
nroff -rC3 -T450-12 -cm **
.
w draft
q
chmod +x draft
```

It may then be invoked as `draft file1 file2`. Note that shell procedures must always be at least readable, so that the shell itself can read commands from the file.

If `draft` were thus created in a directory whose name appears in the user's `PATH` variable, the user could change working directories and still invoke the `draft` command.

Shell procedures may be created dynamically. A procedure may generate a file of commands, invoke another instance of the shell to execute that file, and then remove it. An alternate approach is that of using the *dot* command (`.`) to make the current shell read commands from the new file, allowing use of existing shell variables and avoiding the spawning of an additional process for another shell.

Many users prefer to write shell procedures instead of C programs. First, it is easy to create and maintain a shell procedure because it is only a file of ordinary text. Second, it has no corresponding object program that must be generated and maintained. Third, it is easy to create a procedure on the fly, use it a few times, and then remove it. Finally, because shell procedures are usually short in length, written in a high-level programming language, and kept only in their source-language form, they are generally easy to find, understand, and modify.

By convention, directories that contain only commands and/or shell procedures are usually named *bin*. Most groups of users sharing common interests have one or more *bin* directories set up to hold common procedures. Some users have their `PATH` variable list several such directories. Although you can have a number of such directories, it is unwise to go overboard—it may become difficult to keep track of your environment, and efficiency may suffer (§7.3).

4.7 More about Execution Flags

There are several execution flags available in the shell that can be useful in shell procedures:

- e The shell will exit immediately if any command that it executes exits with a non-zero exit status.
- u When this flag is set, the shell treats the use of an unset variable as an error. This flag can be used to perform a global check on variables.
- t The shell exits after reading and executing the commands on the remainder of the current input line.
- n This is a *don't execute* flag. On occasion, one may want to check a procedure for syntax errors, but not to execute the commands in the procedure. Writing `set -nv` at the beginning of the file will accomplish this.
- k All arguments of the form *variable=value* are treated as keyword parameters. When this flag is *not* set, only such arguments that appear *before* the command name are treated as keyword parameters.

5. MISCELLANEOUS SUPPORTING COMMANDS AND FEATURES

Shell procedures can make use of any UNIX command. The commands described in this section are either used especially frequently in shell procedures, or are explicitly designed for such use. More detailed descriptions of each of these commands can be found in Section 1 of the *UNIX User's Manual* [7].

5.1 Conditional Evaluation: `test`

The `test` command evaluates the expression specified by its arguments and, if the expression is true, returns a zero exit status; otherwise, a non-zero (false) exit status is returned; `test` also returns a non-zero exit status if it has no arguments. Often it is convenient to use the `test` command as the first command in the *command list* following an `if` or a `while`. Shell variables used in `test` expressions should be enclosed in double quotes if there is any chance of their being null or not set.

On some UNIX systems, the square brackets (`[]`) may be used as an alias for `test`; e.g., `[expression]` has the same effect as `test expression`.

The following is a partial list of the primaries that can be used to construct a conditional expression:

- `-r file` *true* if the named file exists and is readable by the user.
- `-w file` *true* if the named file exists and is writable by the user.
- `-x file` *true* if the named file exists and is executable by the user.
- `-s file` *true* if the named file exists and has a size greater than zero.
- `-d file` *true* if the named file exists and is a directory.
- `-f file` *true* if the named file exists and is an ordinary file.
- `-p file` *true* if the named file exists and is a named pipe (*fifo*).
- `-z s1` *true* if the length of string *s1* is zero.
- `-n s1` *true* if the length of the string *s1* is non-zero.
- `-t fildes` *true* if the open file whose file descriptor number is *fildes* is associated with a terminal device. If *fildes* is not specified, file descriptor 1 is used by default.
- `s1 = s2` *true* if strings *s1* and *s2* are identical.
- `s1 != s2` *true* if strings *s1* and *s2* are *not* identical.
- `s1` *true* if *s1* is *not* the null string.
- `n1 -eq n2` *true* if the integers *n1* and *n2* are algebraically equal; other algebraic comparisons are indicated by `-ne`, `-gt`, `-ge`, `-lt`, and `-le`.

These primaries may be combined with the following operators:

- `!` unary negation operator.
- `-a` binary logical *and* operator.
- `-o` binary logical *or* operator; it has lower precedence than `-a`.
- `(expr)` parentheses for grouping; they must be escaped to remove their significance to the shell; in the absence of parentheses, evaluation proceeds from left to right.

Note that all primaries, operators, file names, etc., are separate arguments to `test`.

5.2 Reading a Line: `line`

The `line` command takes one line from standard input and prints it on standard output. This is useful when you need to read a line from a file, or capture the line in a variable. The functions of `line` and of the `read` command that is internal to the shell differ in that input/output redirection is possible only with `line`. If the user does not require input/output redirection, `read` is faster and more efficient. An example of a usage of `line` for which `read` would not suffice is:

```
firstline=`line < somefile`
```

5.3 Simple Output: `echo`

The `echo` command, invoked as `echo [arg ...]` copies its arguments to the standard output, each followed by a single space, except for the last argument, which is normally followed by a new-line; often, it is used to prompt the user for input, to issue diagnostics in shell procedures, or to add a few lines to an output stream in the middle of a pipeline. Another use is to verify the argument list generation process before issuing a command that does something drastic. The command `ls` is often replaced by `echo *`, because the latter is faster and prints fewer lines of output.

The `echo` command recognizes several escape sequences. A `\n` yields a new-line character; a `\c` removes the new-line from the end of the echoed line. The following prompts the user, allowing one to type on the same line as the prompt:

```
echo 'enter name:\c'
read name
```

The `echo` command also recognizes octal escape sequences for *all* characters, whether printable or not: `echo "\007"` typed at a terminal will cause the bell on that terminal to ring.

5.4 Expression Evaluation: `expr`

The `expr` command provides arithmetic and logical operations on integers and some pattern matching facilities on its arguments. It evaluates a single expression and writes the result on the standard output; `expr` can be used inside grave accents to set a variable. Typical examples are:

```
#      increment $a
a=`expr $a + 1`
#      put third through last characters of
#      $1 into substring
substring=`expr "$1" : '.*\(.*\)`
#      obtain length of $1
c=`expr "$1" : '.*'`
```

The most common uses of `expr` are in counting iterations of a loop and in using its pattern matching capability to pick apart strings; see `expr(1)` for more details.

5.5 `true` and `false`

The `true` and `false` commands perform the obvious functions of exiting with zero and non-zero exit status, respectively. The `true` command is often used to implement an unconditional loop.

5.6 Input/Output Redirection Using File Descriptors.

✗ Beginners should skip this section on first reading.

Above (§3.6.2), we mentioned that a command occasionally directs output to some file associated with a file descriptor other than 1 or 2. In languages such as C, one can associate output with *any* file descriptor by using the `write(2)` system call. The shell provides its own mechanism for creating an output file associated with a particular file descriptor. By typing:

```
fd1 >&fd2
```

where *fd1* and *fd2* are valid file descriptors, one can direct output that would normally be associated with file descriptor *fd1* onto the file associated with *fd2*. The default value for *fd1* and *fd2* is 1. If, at execution time, no file is associated with *fd2*, then the redirection is void. The most common use of this mechanism is that of directing standard error output to the same file as standard output. This is accomplished by typing:

```
command 2>&1
```

If one wanted to redirect both standard output and standard error output to the same file, one would type:

```
command 1> file 2>&1
```

The order here is significant: first, file descriptor 1 is associated with *file*; then file descriptor 2 is associated with the same file as is *currently* associated with file descriptor 1. If the order of the redirections were reversed, standard error output would go to the terminal, and standard output would go to *file*, because at the time of the error output redirection, file descriptor 1 still would have been associated with the terminal.

This mechanism can also be generalized to the redirection of standard *input*. One could type:

```
fda<&fdb
```

to cause both file descriptors *fda* and *fdb* to be associated with the same input file; if *fda* or *fdb* is not specified, file descriptor 0 is assumed. Such input redirection is useful for commands that use two or more input sources. Another use of this notation is for sequential reading and processing of a file; see *merge* in §6 for an example of use of this feature.

5.7 Conditional Substitution

Normally, the shell replaces occurrences of *\$variable* by the string value assigned to *variable*, if any. However, there exists a special notation to allow conditional substitution, dependent upon whether the variable is set and/or not null. By definition, a variable is *set* if it has *ever* been assigned a value. The value of a variable can be the null string, which may be assigned to a variable in any one of the following ways:

```
A=
bcd=""
Ef_g=' '
set ' ' ""
```

The first three of these examples assign the null string to each of the corresponding *shell variables*. The last example sets the first and second *positional parameters* to the null string, and *unsets* all other positional parameters.

The following conditional expressions depend upon whether a variable is *set and not null* (note that, in these expressions, *variable* refers to either a digit or a variable name and the meaning of braces differs from that described in §3.4.2 and §4.4.7):

- `${variable}:-string`** If *variable* is set and is non-null, then substitute the value *\$variable* in place of this expression. Otherwise, replace the expression with *string*. Note that the value of *variable* is *not* changed by the evaluation of this expression.
- `${variable}:=string`** If *variable* is set and is non-null, then substitute the value *\$variable* in place of this expression; otherwise, set *variable* to *string*, and then substitute the value *\$variable* in place of this expression. Positional parameters may not be assigned values in this fashion.

`${variable:?string}` If *variable* is set and is non-null, then substitute the value of *variable* for the expression; otherwise, print a message of the form:

variable: *string*

and exit from the current shell. (If the shell is the login shell, it is not exited.) If *string* is omitted in this form, then the message:

variable: parameter null or not set

is printed instead.

`${variable:+string}` If *variable* is set and is non-null, then substitute *string* for this expression, otherwise, substitute the null string. Note that the value of *variable* is not altered by the evaluation of this expression.

These expressions may also be used without the colon (:), in which case the shell does *not* check whether *variable* is null or not; it only checks whether *variable* has *ever* been set.

The two examples below illustrate the use of this facility:

1. If `PATH` has ever been set and is not null, then keep its current value; otherwise, set it to the string `:/bin:/usr/bin`. Note that one needs an explicit assignment to set `PATH` in this form:

```
PATH=${PATH:-'/bin:/usr/bin'}
```

2. If `HOME` is set and is not null, then change directory to it, otherwise set it to the given value and change directory to it; note that `HOME` is automatically assigned a value in this case:

```
cd ${HOME:='/usr/gas'}
```

5.8 Invocation Flags

There are four flags that may be specified on the command line invoking the shell; these flags may *not* be turned on via the `set` command:

- i If this flag is specified, or if the shell's input and output are both attached to a terminal, the shell is *interactive*. In such a shell, `INTERRUPT` (signal 2) is caught and ignored, while `QUIT` (signal 3) and `SOFTWARE TERMINATION` (signal 15) are ignored.
- s If this flag is specified or if no input/output redirection arguments are given, the shell reads commands from standard input. Shell output is written to file descriptor 2. The shell you get upon logging into the system effectively has the `-s` flag turned on.
- c When this flag is turned on, the shell reads commands from the first string following the flag. Remaining arguments are ignored. Double quotes should be used to enclose a multi-word string, in order to allow for variable substitution.
- r When this flag is specified on invocation, then the *restricted shell* is invoked. This is a version of the shell in which certain actions are disallowed. In particular, the `cd` command produces an error message, and the user cannot set `PATH`. See `sh(1)` for a more detailed description.

6. EXAMPLES OF SHELL PROCEDURES

Some examples in this section are quite difficult for beginners. For ease of reference, the examples are arranged alphabetically by name, rather than by degree of difficulty.

coppairs:

```
#      usage: coppairs file1 file2 ...
#      copy file1 to file2, file3 to file4, ...
while test "$2" != ""
do
    cp $1 $2
    shift; shift
done
if test "$1" != ""
then echo "$0: odd number of arguments"
fi
```

Note: This procedure illustrates the use of a while loop to process a list of positional parameters that are somehow related to one another. Here a while loop is much better than a for loop, because you can adjust the positional parameters via `shift` to handle related arguments.

copyto:

```
#      usage: copyto dir file ...
#      copy argument files to 'dir', making sure that at least
#      two arguments exist and that 'dir' is a directory
if test $# -lt 2
then echo "$0: usage: copyto directory file ..."
elif test ! -d $1
then echo "$0: $1 is not a directory";
else dir=$1; shift
    for eachfile
    do
        cp $eachfile $dir
    done
fi
```

Note: This procedure uses an `if` command with two tests in order to screen out improper usage. The `for` loop at the end of the procedure loops over all of the arguments to `copyto` but the first; the original `$1` is shifted off.

distinct:

```
#      usage: distinct
#      reads standard input and reports list of alphanumeric strings
#      that differ only in case, giving lower-case form of each
tr -cs '[A-Z][a-z][0-9]' '\012*' | sort -u |
tr '[A-Z]' '[a-z]' | sort | uniq -d
```

Note: This procedure is an example of the kind of process that is created by the left-to-right construction of a long pipeline. It may not be immediately obvious how this works. (You may wish to consult `tr(1)`, `sort(1)`, and `uniq(1)` if you are completely unfamiliar with these commands.) The `tr` translates all characters except letters and digits into new-line characters, and then squeezes out repeated new-line characters. This leaves each string (in this case, any contiguous sequence of letters and digits) on a separate line. The `sort` command sorts the lines and emits only one line from any sequence of one or more repeated lines. The next `tr` converts everything to lower case, so that identifiers differing only in case become identical. The output is sorted again to bring such duplicates together. The `uniq -d` prints (once) only those lines that occur more than once, yielding the desired list.

The process of building such a pipeline uses the fact that pipes and files can usually be interchanged; the two lines below are equivalent, assuming that sufficient disk space is available:

```
cmd1 | cmd2 | cmd3
cmd1 > temp1; < temp1 cmd2 > temp2; < temp2 cmd3; rm temp[12]
```

Starting with a file of test data on the standard input and working from left to right, each command is executed taking its input from the previous file and putting its output in the next file. The final output is then examined to make sure that it contains the expected result. The goal is to create a series of transformations that will convert the input to the desired output. As an exercise, try to mimic `distinct` with such a step-by-step process, using a file of test data containing:

```
ABC:DEF/DEF
ABC1 ABC
Abc abc
```

Although pipelines can give a concise notation for complex processes, exercise some restraint, lest you succumb to the "one-line syndrome" sometimes found among users of especially concise languages. This syndrome often yields incomprehensible code.

draft:

```
#      usage: draft file(s)
#      prints the draft (-rC3) of a document on a DASI 450
#      terminal in 12-pitch using memorandum macros (MM).
nroff -rC3 -T450-12 -cm $*
```

Note: Users often write this kind of procedure for convenience in dealing with commands that require the use of many distinct flags that cannot be given default values that are reasonable for all (or even most) users.

edfind:

```
#      usage: edfind file arg
#      find the last occurrence in 'file' of a line whose
#      beginning matches 'arg', then print 3 lines (the one
#      before, the line itself, and the one after)
ed - $1 <<|
H
?^$2?;-,+p
|
```

Note: This procedure illustrates the practice of using editor (`ed`) in-line input scripts into which the shell can substitute the values of variables. It is a good idea to turn on the `H` option of `ed` when embedding an `ed` script in a shell procedure (see `ed(1)`).

edlast:

```
#      usage: edlast file
#      prints the last line of file, then deletes that line
ed - $1 <<-\eof      # no variable substitutions in "ed" script
H
$P
$d
w
q
eof
echo Done.
```


Note: This procedure contains an in-line input document or script (see §4.4.9); it also illustrates the effect of inhibiting substitution by escaping a character in the *eofstring* (here, *eof*) of the input redirection. If this had not been done, *\$p* and *\$d* would have been treated as shell variables.

fsplit:

```
#      usage: fsplit file1 file2
#      read standard input and divide it into three parts:
#      append any line containing at least one letter
#      to file1, any line containing at least one digit
#      but no letters to file2, and throw the rest away
total=0 lost=0
while read next
do
    total="`expr $total + 1`"
    case "$next" in
    *[A-Za-z]*)
        echo "$next" >> $1 ;;
    *[0-9]*)
        echo "$next" >> $2 ;;
    *)
        lost="`expr $lost + 1`"
    esac
done
echo "$total lines read, $lost thrown away"
```

Note: In this procedure, each iteration of the while loop reads a line from the input and analyzes it. The loop terminates only when read encounters an end-of-file.

☞ *Don't use the shell to read a line at a time unless you must—it can be grotesquely slow (§7.2.1).*

initvars:

```
#      usage: . initvars
#      use carriage return to indicate "no change"
echo "initializations? \c"
read response
if test "$response" = y
then  echo "PS1=\c"; read temp
      PS1=${temp:-$PS1}
      echo "PS2=\c"; read temp
      PS2=${temp:-$PS2}
      echo "PATH=\c"; read temp
      PATH=${temp:-$PATH}
      echo "TERM=\c"; read temp
      TERM=${temp:-$TERM}
fi
```

Note: This procedure would be invoked by a user at the terminal, or as part of a *.profile* file. The assignments are effective even when the procedure is finished, because the *dot* command is used to invoke it. To better understand the *dot* command, invoke *initvars* as indicated above and check the values of *PS1*, *PS2*, *PATH*, and *TERM*; then make *initvars* executable, type *initvars*, assigning different values to the three variables, and check again the values of these three shell variables after *initvars* terminates. It is assumed that *PS1*, *PS2*, *PATH*, and *TERM* have been exported, presumably by your *.profile* (§3.9.2, §4.1).

merge:

```

#      usage:  merge src1 src2 [ dest ]
#      merge two files, every other line.
#      the first argument starts off the merge,
#      excess lines of the longer file are appended to
#      the end of the resultant file
exec 4<$1 5<$2
dest=${3-$1.m}          # default destination file is named $1.m
while true
do
    # alternate reading from the files;
    # 'more' represents the file descriptor
    # of the longer file
    line <&4 >>$dest || { more=5; break ;}
    line <&5 >>$dest || { more=4; break ;}
done
    # delete the last line of destination
    # file, because it is blank.

ed - $dest <<\eof
    H
    $d
    w
    q
eof
while line <&$more >> $dest
do ;; done          # read the remainder of the longer
                    # file - the body of the 'while' loop
                    # does nothing; the work of the loop
                    # is done in the command list following
                    # 'while'

```

Note: This procedure illustrates a technique for reading sequential lines from a file or files without creating any sub-shells to do so. When the file descriptor is used to access a file, the effect is that of opening the file and moving a file pointer along until the end of the file is read. If the input redirections used *src1* and *src2* explicitly rather than the associated file descriptors, this procedure would never terminate, because the *first* line of each file would be read over and over again.

mkfiles:

```

#      usage:  mkfiles pref [ quantity ]
#      makes 'quantity' (default = 5) files, named pref1, pref2, ...
quantity=${2-5}
i=1
while test "$i" -le "$quantity"
do
    > $1$i
    i=`expr $i + 1`
done

```

Note: This procedure uses input/output redirection to create zero-length files. The `expr` command is used for counting iterations of the `while` loop. Compare this procedure with procedure `null` below.

mmt:

```

if test "$#" = 0; then cat <<\|
Usage: "mmt [ options ] files" where "options" are:
-a      => output to terminal
-e      => preprocess input with eqn
-t      => preprocess input with tbl
-Tst    => output to STARE
-T4014  => output to Tektronix 4014
-Tvp    => output to Versatec printer
-       => use instead of "files" when mmt used inside a pipeline.
Other options as required by TROFF and the MM macros.
|
    exit 1
fi
PATH='/bin:/usr/bin'; O='-g'; o='|gcat -ph';
#           Assumes typesetter is accessed via gcat(1)
#           If typesetter is on-line, use O=''; o=''
while test -n "$1" -a ! -r "$1"
do case "$1" in
-a)          O='-a';          o=''; ;;
-Tst)       O='-g';          o='|gcat -st';;
#           Above line for STARE only
-T4014)     O='-t';          o='|tc';;
-Tvp)       O='-t';          o='|vpr -t';;
-e)         e='eqn';;
-t)         f='tbl';;
-)         break;;
*)         a="$a $1";;
    esac
    shift
done
if test -z "$1";      then echo 'mmt: no input file'; exit 1; fi
if test "$O" = '-g'; then x="-f$1"; fi
d="$*"
if test "$d" = '-';  then shift;    x='';    d='';    fi
if test -n "$f";    then f="tbl $*|";  d='';    fi
if test -n "$e"
    then          if test -n "$f"
                  then e='eqn|'
                  else e="eqn $*|";          d='';
    fi
fi
eval "$f $e troff $O -cm $a $d $o $x"; exit 0

```

Note: This is a slightly simplified version of an actual UNIX command (although this is *not* the version included in UNIX Release 4.0). It uses many of the features available in the shell; if you can follow through it without getting lost, you have a good understanding of shell programming. Pay particular attention to the process of building a command line from shell variables and then using `eval` to execute it.

null:

```

#       usage: null file
#       create each of the named files as an empty file
for eachfile
do
    > $eachfile
done

```

Note: This procedure uses the fact that output redirection creates the (empty) output file if that file does not already exist. Compare this procedure with procedure `mkfiles` above.

phone:

```

#      usage: phone initials
#      prints the phone number(s) of person with given initials
echo 'inits      ext      home'
grep "^$1" <<\!
abc      1234      999-2345
def      2234      583-2245
ghi      3342      988-1010
xyz      4567      555-1234
|

```

Note: This procedure is an example of using an in-line input document or *script* to maintain a *small* data base.

writemail:

```

#      usage: writemail message user
#      if user is logged in, write message on terminal;
#      otherwise, mail it to user
echo "$1" | { write "$2" || mail "$2" ;}

```

Note: This procedure illustrates command grouping. The message specified by \$1 is piped to the write command and, if write fails, to the mail command.

7. EFFECTIVE AND EFFICIENT SHELL PROGRAMMING**7.1 Overall Approach**

This section outlines strategies for writing *efficient* shell procedures, i.e., ones that do not waste resources unreasonably in accomplishing their purposes. In the authors' opinion, the primary reason for choosing the shell procedure as the implementation method is to achieve a desired result at a minimum *human* cost. Emphasis should *always* be placed on simplicity, clarity, and readability, but efficiency can also be gained through awareness of a few design strategies. In many cases, an effective redesign of an existing procedure improves its efficiency by reducing its size, and often increases its comprehensibility. In any case, one should not worry about optimizing shell procedures unless they are intolerably slow or are known to consume a lot of resources.

The same kind of iteration cycle should be applied to shell procedures as to other programs: write code, measure it, and optimize only the *few* important parts. The user should become familiar with the `time` command, which can be used to measure both entire procedures and parts thereof. Its use is strongly recommended; human intuition is notoriously unreliable when used to estimate timings of programs, even when the style of programming is a familiar one. Each timing test should be run several times, because the results are easily disturbed by, for instance, variations in system load.

7.2 Approximate Measures of Resource Consumption

7.2.1 Number of Processes Generated. When large numbers of short commands are executed, the actual execution time of the commands may well be dominated by the overhead of creating processes. The procedures that incur significant amounts of such overhead are those that perform much looping and those that generate command sequences to be interpreted by another shell.

If you are worried about efficiency, it is important to know which commands are currently built into the shell, and which are not. Here is the alphabetical list of those that are built-in:

break	case	cd	continue	eval	exec
exit	export	for	if	newgrp	read
readonly	set	shift	test	times	trap
ulimit	umask	until	wait	while	.
:	{...}				

(...) executes as a child process, i.e., the shell does a *fork*, but no *exec*. Any command *not* in the above list requires both *fork* and *exec*.

The user should always have at least a vague idea of the number of processes generated by a shell procedure. In the bulk of observed procedures, the number of processes spawned (not necessarily simultaneously) can be described by:

$$\text{processes} = k * n + c$$

where *k* and *c* are constants, and *n* is the number of procedure arguments, the number of lines in some input file, the number of entries in some directory, or some other obvious quantity. Efficiency improvements are most commonly gained by reducing the value of *k*, sometimes to zero. Any procedure whose complexity measure includes n^2 terms or higher powers of *n* is likely to be intolerably expensive.

As an example, here is an analysis of procedure `fsplit` of §6. For each iteration of the loop, there is one `expr` plus either an `echo` or another `expr`. One additional `echo` is executed at the end. If *n* is the number of lines of input, the number of processes is $2 * n + 1$. On the other hand, the number of processes in the following (equivalent) procedure is 12, regardless of the number of lines of input:

```
#      faster fsplit
trap 'rm temp$$; trap 0; exit' 0 1 2 3 15
start1=0 start2=0
b='[A-Za-z]'
```

```
cat > temp$$          # read standard input into temp file
                    # save original lengths of $1, $2
if test -s "$1"; then start1=`wc -l < $1`; fi
if test -s "$2"; then start2=`wc -l < $2`; fi
grep "$b" temp$$ >> $1 # lines with letters onto $1
grep -v "$b" temp$$ | grep '[0-9]' >> $2
                    # lines with only numbers onto $2
total=`wc -l < temp$$`
end1=`wc -l < $1`
end2=`wc -l < $2`
lost=`expr $total - \( $end1 - $start1 \) - \( $end2 - $start2 \)`
echo "$total lines read, $lost thrown away"
```

This version is often ten times faster than `fsplit`, and it is even faster for larger input files.

Some types of procedures should *not* be written using the shell. For example, if one or more processes are generated for each character in some file, it is a good indication that the procedure should be rewritten in C.

Shell procedures should not be used to scan or build files a character at a time.

7.2.2 Number of Data Bytes Accessed. It is worthwhile considering any action that reduces the number of bytes read or written. This may be important for those procedures whose time is spent passing data around among a few processes, rather than in creating large numbers of short processes. Some filters shrink their output, others usually increase it. It always pays to put the *shrinkers* first when the order is irrelevant. Which of the following is likely to be faster?

```
sort file | grep pattern
grep pattern file | sort
```

7.2.3 Directory Searches. Directory searching can consume a great deal of time, especially in those applications that utilize deep directory structures and long path names. Judicious use of `cd` can help shorten long path names and thus reduce the number of directory searches needed. As an exercise, try the following commands (on a fairly quiet system):⁸

```
time sh -c 'ls -l /usr/bin/* >/dev/null'
time sh -c 'cd /usr/bin; ls -l * >/dev/null'
```

7.3 Efficient Organization

7.3.1 Directory-Search Order and the PATH Variable. The `PATH` variable is a convenient mechanism for allowing organization and sharing of procedures. However, it must be used in a sensible fashion, or the result may be a great increase in system overhead that occurs in a subtle, but avoidable, way.

The process of finding a command involves reading every directory included in every path name that precedes the needed path name in the current `PATH` variable. As an example, consider the effect of invoking `nroff` (i.e., `/usr/bin/nroff`) when `$PATH` is `:/bin:/usr/bin`. The sequence of directories read is: `.`, `/`, `/bin`, `/`, `/usr`, and `/usr/bin`, i.e., a total of six directories. A long path list assigned to `PATH` can increase this number significantly.

The vast majority of command executions are of commands found in `/bin` and, to a somewhat lesser extent, in `/usr/bin`. Careless `PATH` setup may lead to a great deal of unnecessary searching. The following four examples are ordered from worst to best (but *only* with respect to the efficiency of command searches):

```
:/a1/tf/jtb/bin:/usr/lbin:/bin:/usr/bin
:/bin:/a1/tf/jtb/bin:/usr/lbin:/usr/bin
:/bin:/usr/bin:/a1/tf/jtb/bin:/usr/lbin
/bin::/usr/bin:/a1/tf/jtb/bin:/usr/lbin
```

The first one above should be avoided. The others are acceptable, the choice among them is dictated by the rate of change in the set of commands kept in `/bin` and `/usr/bin`.

A procedure that is expensive because it invokes many short-lived commands may often be speeded up by setting the `PATH` variable inside the procedure such that the fewest possible directories are searched in an optimum order; the `mmt` example in §6 does this.

7.3.2 Good Ways to Set up Directories. It is wise to avoid directories that are larger than necessary. You should be aware of several *magic* sizes. A directory that contains entries for up to 30 files (plus the required `.` and `..`) fits in a single disk block and can be searched very efficiently. One that has up to 286 entries is still a *small* file; anything larger is usually a disaster when used as a working directory. It is especially important to keep login directories small, preferably one block at most. Note that, as a rule, directories never shrink.

ACKNOWLEDGEMENTS

The UNIX shell was initially written by S. R. Bourne [2,3]. Its design is based, in part, on the original UNIX shell [15] and on the PWB/UNIX shell [12], some features having been taken from both. Similarities also exist with the command interpreters of the Cambridge Multiple Access System and of the MIT Compatible Time-Sharing System. T. E. Fritz and several other colleagues provided helpful comments during the writing of this tutorial; T. A. Dolotta, in addition, provided a great deal of editorial assistance.

8. You may have to do some reading in the *UNIX User's Manual* [7] to understand exactly what is going on in these examples.

REFERENCES

- [1] Bianchi, M. H., and Wood, J. L. A User's Viewpoint on the Programmer's Workbench. *Proc. Second Int. Conf. on Software Engineering*, pp. 193-99 (Oct. 13-15, 1976).
- [2] Bourne, S. R. The UNIX Shell. *The Bell System Technical Journal*, Vol. 57, No. 6, Part 2, pp. 1971-90 (July-Aug. 1978).
- [3] Bourne, S. R. *An Introduction to the UNIX Shell*. Bell Laboratories (1979).
- [4] Dolotta, T. A., Haight, R. C., and Mashey, J. R. The Programmer's Workbench. *The Bell System Technical Journal*, Vol. 57, No. 6, Part 2, pp. 2177-200 (July-Aug. 1978).
- [5] Dolotta, T. A., and Mashey, J. R. An Introduction to the Programmer's Workbench. *Proc. Second Int. Conf. on Software Engineering*, pp. 164-68 (Oct. 13-15, 1976).
- [6] Dolotta, T. A., and Mashey, J. R. Using a Command Language as the Primary Programming Tool. In: Beech, D. (ed.), *Command Language Directions* (Proc. of the Second IFIP Working Conf. on Command Languages), pp. 35-55. Amsterdam: North Holland (1980).
- [7] Dolotta, T. A., Olsson, S. B., and Petrucci, A. G., eds. *UNIX User's Manual—Release 3.0*. Bell Laboratories (June 1980).
- [8] Kernighan, B. W., and Mashey, J. R. The UNIX Programming Environment. *COMPUTER*, Vol. 14, No. 4, pp. 12-24 (April 1981); an earlier version of this paper was published in *Software—Practice & Experience*, Vol. 9, No. 1, pp. 1-15 (Jan. 1979).
- [9] Kernighan, B. W., and Plauger, P. J. Software Tools. *Proc. First Nat. Conf. on Software Engineering*, pp. 8-13 (Sept. 11-12, 1975).
- [10] Kernighan, B. W., and Plauger, P. J. *Software Tools*. Reading, MA: Addison-Wesley (1976).
- [11] Kernighan, B. W., and Ritchie, D. M. *The C Programming Language*. Englewood Cliffs, NJ: Prentice-Hall (1978).
- [12] Mashey, J. R. *PWB/UNIX Shell Tutorial*. Bell Laboratories (1977).
- [13] Ritchie, D. M., and Thompson, K. The UNIX Time-Sharing System. *The Bell System Technical Journal*, Vol. 57, No. 6, Part 2, pp. 1905-29 (July-Aug. 1978).
- [14] Snyder, G. A., and Mashey, J. R. UNIX Documentation Road Map. Bell Laboratories (January 1981).
- [15] Thompson, K. The UNIX Command Language. In: *Structured Programming—Infotech State of the Art Report*, pp. 375-84. Infotech International Limited, Nicholson House, Maidenhead, Berkshire, England (1976).

January 1981

CONTENTS

1. INTRODUCTION	1
2. OVERVIEW OF THE UNIX ENVIRONMENT	1
2.1 File System 1	
2.2 UNIX Processes 2	
3. SHELL BASICS	3
3.1 Commands 3	
3.2 How the Shell Finds Commands 4	
3.3 Generation of Argument Lists 4	
3.4 Shell Variables 5	
3.4.1 Positional Parameters. 5	
3.4.2 User-defined Variables. 5	
3.4.3 Command Substitution. 7	
3.4.4 Predefined Special Variables. 8	
3.5 Quoting Mechanisms 9	
3.6 Redirection of Input and Output 9	
3.6.1 Standard Input and Standard Output. 9	
3.6.2 Diagnostic and Other Outputs. 10	
3.7 Command Lines and Pipelines 10	
3.8 Examples 10	
3.9 Changing the State of the Shell and the .profile File 11	
3.9.1 Cd. 11	
3.9.2 The .profile File. 11	
3.9.3 Execution Flags: set. 12	
4. USING THE SHELL AS A COMMAND: SHELL PROCEDURES	12
4.1 A Command's Environment 12	
4.2 Invoking the Shell 13	
4.3 Passing Arguments to the Shell; shift 13	
4.4 Control Commands 14	
4.4.1 Structured Conditional: if. 16	
4.4.2 Multi-way Branch: case. 16	
4.4.3 Conditional Looping: while and until. 17	
4.4.4 Looping over a List: for. 17	
4.4.5 Loop Control: break and continue. 18	
4.4.6 End-of-file and exit. 18	
4.4.7 Command Grouping: Parentheses and Braces. 18	
4.4.8 Input/Output Redirection and Control Commands. 19	
4.4.9 In-line Input Documents. 19	
4.4.10 Transfer to Another File and Back: the Dot (.) Command. 20	
4.4.11 Interrupt Handling: trap. 20	
4.5 Special Shell Commands 21	
4.6 Creation and Organization of Shell Procedures 23	
4.7 More about Execution Flags 23	
5. MISCELLANEOUS SUPPORTING COMMANDS AND FEATURES	24
5.1 Conditional Evaluation: test 24	
5.2 Reading a Line: line 25	
5.3 Simple Output: echo 25	
5.4 Expression Evaluation: expr 25	
5.5 true and false 25	
5.6 Input/Output Redirection Using File Descriptors. 25	

5.7 Conditional Substitution	26
5.8 Invocation Flags	27
6. EXAMPLES OF SHELL PROCEDURES	28
coppairs:	28
copyto:	28
distinct:	28
draft:	29
edfind:	29
edlast:	29
fsplit:	30
initvars:	30
merge:	31
mkfiles:	31
mmt:	32
null:	32
phone:	33
writemail:	33
7. EFFECTIVE AND EFFICIENT SHELL PROGRAMMING	33
7.1 Overall Approach	33
7.2 Approximate Measures of Resource Consumption	33
7.2.1 Number of Processes Generated.	33
7.2.2 Number of Data Bytes Accessed.	34
7.2.3 Directory Searches.	35
7.3 Efficient Organization	35
7.3.1 Directory-Search Order and the PATH Variable.	35
7.3.2 Good Ways to Set up Directories.	35
ACKNOWLEDGEMENTS	35
REFERENCES	36

An Introduction to the UNIX Shell

S. R. Bourne

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

The *shell* is a command programming language that provides an interface to the UNIX† operating system. Its features include control-flow primitives, parameter passing, variables and string substitution. Constructs such as *while*, *if then else*, *case* and *for* are available. Two-way communication is possible between the *shell* and commands. String-valued parameters, typically file names or flags, may be passed to a command. A return code is set by commands that may be used to determine control-flow, and the standard output from a command may be used as shell input.

The *shell* can modify the environment in which commands run. Input and output can be redirected to files, and processes that communicate through ‘pipes’ can be invoked. Commands are found by searching directories in the file system in a sequence that can be defined by the user. Commands can be read either from the terminal or from a file, which allows command procedures to be stored for later use.

1.0 INTRODUCTION

The shell is both a command language and a programming language that provides an interface to the UNIX operating system. This memorandum describes, with examples, the UNIX shell. The first section covers most of the everyday requirements of terminal users. Some familiarity with UNIX is an advantage when reading this section; see, for example, *UNIX for Beginners*.¹ Section 2 describes those features of the shell primarily intended for use within shell procedures. These include the control-flow primitives and string-valued variables provided by the shell. A knowledge of a programming language would be a help when reading this section. The last section describes the more advanced features of the shell. References of the form “see *pipe* (2)” are to a section of the *UNIX User’s Manual*.²

1.1 Simple Commands

Simple commands consist of one or more words separated by blanks. The first word is the name of the command to be executed; any remaining words are passed as arguments to the command. For example,

who

is a command that prints the names of users logged in. The command

ls -l

prints a list of files in the current directory. The argument *-l* tells *ls* to print status information, size and the creation date for each file.

† UNIX is a trademark of Bell Laboratories.

1.2 Background Commands

To execute a command the shell normally creates a new *process* and waits for it to finish. A command may be run without waiting for it to finish. For example,

```
cc pgm.c &
```

calls the C compiler to compile the file *pgm.c*. The trailing **&** is an operator that instructs the shell not to wait for the command to finish. To help keep track of such a process the shell reports its process number following its creation. A list of currently active processes may be obtained using the *ps* command.

1.3 Input/Output Redirection

Most commands produce output on the standard output that is initially connected to the terminal. This output may be sent to a file by writing, for example,

```
ls -l >file
```

The notation *>file* is interpreted by the shell and is not passed as an argument to *ls*. If *file* does not exist then the shell creates it; otherwise the original contents of *file* are replaced with the output from *ls*. Output may be appended to a file using the notation

```
ls -l >>file
```

In this case *file* is also created if it does not already exist.

The standard input of a command may be taken from a file instead of the terminal by writing, for example,

```
wc <file
```

The command *wc* reads its standard input (in this case redirected from *file*) and prints the number of characters, words and lines found. If only the number of lines is required then

```
wc -l <file
```

could be used.

1.4 Pipelines and Filters

The standard output of one command may be connected to the standard input of another by writing the 'pipe' operator, indicated by **|**, as in,

```
ls -l | wc
```

Two commands connected in this way constitute a *pipeline* and the overall effect is the same as

```
ls -l >file; wc <file
```

except that no *file* is used. Instead the two processes are connected by a pipe (see *pipe* (2)) and are run in parallel. Pipes are unidirectional and synchronization is achieved by halting *wc* when there is nothing to read and halting *ls* when the pipe is full.

A *filter* is a command that reads its standard input, transforms it in some way, and prints the result as output. One such filter, *grep*, selects from its input those lines that contain some specified string. For example,

```
ls | grep old
```

prints those lines, if any, of the output from *ls* that contain the string *old*. Another useful filter is *sort*. For example,

```
who | sort
```

will print an alphabetically sorted list of logged in users.

A pipeline may consist of more than two commands, for example,

```
ls | grep old | wc -l
```

prints the number of file names in the current directory containing the string *old*.

1.5 File Name Generation

Many commands accept arguments which are file names. For example,

```
ls -l main.c
```

prints information relating to the file *main.c*.

The shell provides a mechanism for generating a list of file names that match a pattern. For example,

```
ls -l *.c
```

generates, as arguments to *ls*, all file names in the current directory that end in *.c*. The character *** is a pattern that will match any string including the null string. In general *patterns* are specified as follows.

- * Matches any string of characters including the null string.
- ? Matches any single character.
- [...] Matches any one of the characters enclosed. A pair of characters separated by a minus will match any character lexically between the pair.

For example,

```
[a-z]*
```

matches all names in the current directory beginning with one of the letters *a* through *z*.

```
/usr/fred/test/?
```

matches all names in the directory */usr/fred/test* that consist of a single character. If no file name is found that matches the pattern then the pattern is passed, unchanged, as an argument.

This mechanism is useful both to save typing and to select names according to some pattern. It may also be used to find files. For example,

```
echo /usr/fred/*/core
```

finds and prints the names of all *core* files in sub-directories of */usr/fred*. (*echo* is a standard UNIX command that prints its arguments, separated by blanks.) This last feature can be expensive, requiring a scan of all sub-directories of */usr/fred*.

There is one exception to the general rules given for patterns. The character *'* at the start of a file name must be explicitly matched.

```
echo *
```

will therefore echo all file names in the current directory not beginning with *'*.

```
echo .*
```

will echo all those file names that begin with *'*. This avoids inadvertent matching of the names *'* and *'.*, which mean 'the current directory' and 'the parent directory', respectively. (Notice that *ls* suppresses information for the files *'* and *'.*.)

1.6 Quoting

Characters that have a special meaning to the shell, such as *<* *>* *** *?* *|* *&*, are called metacharacters. A complete list of metacharacters is given in appendix B. Any character preceded by a ** is *quoted* and loses its special meaning, if any. The ** is elided so that:

```
echo \?
```

will echo a single `?`, and

```
echo \\
```

will echo a single `\`. To allow long strings to be continued over more than one line the sequence `\new-line` is ignored. The `\` is convenient for quoting single characters. When more than one character needs quoting the above mechanism is clumsy and error prone. A string of characters may be quoted by enclosing the string between single quotes. For example,

```
echo xx`****`xx
```

will echo

```
xx****xx
```

The quoted string may not contain a single quote but may contain new-lines, which are preserved. This quoting mechanism is the most simple and is recommended for casual use. A third quoting mechanism using double quotes is also available that prevents interpretation of some but not all metacharacters. Discussion of the details is deferred to section 3.4.

1.7 Prompting

When the shell is used from a terminal it will issue a prompt before reading a command. By default this prompt is `'$'`. It may be changed by saying, for example,

```
PS1=yesdear
```

that sets the prompt to be the string *yesdear*. If a new-line is typed and further input is needed then the shell will issue the prompt `'>'`. Sometimes this can be caused by mistyping a quote mark. If it is unexpected then an interrupt (DEL) will return the shell to read another command. This prompt may be changed by saying, for example,

```
PS2=more
```

1.8 The Shell and login

Following *login* (1) the shell is called to read and execute commands typed at the terminal. If the user's login directory contains the file `.profile` then it is assumed to contain commands and is read by the shell before reading any commands from the terminal.

1.9 Summary

- `ls`
Print the names of files in the current directory.
- `ls >file`
Put the output from *ls* into *file*.
- `ls | wc -l`
Print the number of files in the current directory.
- `ls | grep old`
Print those file names containing the string *old*.
- `ls | grep old | wc -l`
Print the number of files whose name contains the string *old*.
- `cc pgm.c &`
Run *cc* in the background.

2.0 SHELL PROCEDURES

The shell may be used to read and execute commands contained in a file. For example,

```
sh file [ args ... ]
```

calls the shell to read commands from *file*. Such a file is called a *command procedure* or *shell procedure*. Arguments may be supplied with the call and are referred to in *file* using the positional parameters **\$1**, **\$2**, For example, if the file *wg* contains

```
who | grep $1
```

then

```
sh wg fred
```

is equivalent to

```
who | grep fred
```

UNIX files have three independent attributes, *read*, *write* and *execute*. The UNIX command *chmod* (1) may be used to make a file executable. For example,

```
chmod +x wg
```

will ensure that the file *wg* has execute status. Following this, the command

```
wg fred
```

is equivalent to

```
sh wg fred
```

This allows shell procedures and programs to be used interchangeably. In either case a new process is created to run the command.

As well as providing names for the positional parameters, the number of positional parameters in the call is available as **\$#**. The name of the file being executed is available as **\$0**.

A special shell parameter **\$*** is used to substitute for all positional parameters except **\$0**. A typical use of this is to provide some default arguments, as in

```
nroff -T450 -cm $*
```

which simply prepends some arguments to those already given.

2.1 Control Flow—*for*

A frequent use of shell procedures is to loop through the arguments (**\$1**, **\$2**, ...) executing commands once for each argument. An example of such a procedure is *tel* that searches the file */usr/lib/telnos* that contains lines of the form

```
...
fred mh0123
bert mh0789
...
```

The text of *tel* is

```
for i
do grep $i /usr/lib/telno; done
```

The command

```
tel fred
```

prints those lines in */usr/lib/telno* that contain the string *fred*.

```
tel fred bert
```

prints those lines containing *fred* followed by those for *bert*.

The **for** loop notation is recognized by the shell and has the general form

```
for name in w1 w2 ...
do command-list
done
```

A *command-list* is a sequence of one or more simple commands separated or terminated by a new-line or semicolon. Furthermore, reserved words like **do** and **done** are only recognized following a new-line or semicolon. *name* is a shell variable that is set to the words *w1 w2 ...* in turn each time the *command-list* following **do** is executed. If **in** *w1 w2 ...* is omitted then the loop is executed once for each positional parameter; that is, **in** *\$** is assumed.

Another example of the use of the **for** loop is the *create* command whose text is

```
for i do >$i; done
```

The command

```
create alpha beta
```

ensures that two empty files *alpha* and *beta* exist and are empty. The notation *>file* may be used on its own to create or clear the contents of a file. Notice also that a semicolon (or new-line) is required before **done**.

2.2 Control Flow—case

A multiple way branch is provided for by the **case** notation. For example,

```
case $# in
  1) cat >>$1 ;;
  2) cat >>$2 <$1 ;;
  *) echo `usage: append [ from ] to` ;;
esac
```

is an *append* command. When called with one argument as

```
append file
```

\$# is the string *1* and the standard input is copied onto the end of *file* using the *cat* command.

```
append file1 file2
```

appends the contents of *file1* onto *file2*. If the number of arguments supplied to *append* is other than 1 or 2 then a message is printed indicating proper usage.

The general form of the **case** command is

```
case word in
  pattern ) command-list ;;
  ...
esac
```

The shell attempts to match *word* with each *pattern*, in the order in which the patterns appear. If a match is found the associated *command-list* is executed and execution of the **case** is complete. Since *** is the pattern that matches any string it can be used for the default case.

A word of caution: no check is made to ensure that only one pattern matches the case argument. The first match found defines the set of commands to be executed. In the example below the commands following the second *** will never be executed.

```

case $# in
  *) ... ;;
  *) ... ;;
esac

```

Another example of the use of the **case** construction is to distinguish between different forms of an argument. The following example is a fragment of a *cc* command.

```

for i
do case $i in
  -[ocs]) ... ;;
  -*) echo "unknown flag $i" ;;
  *.c) /lib/c0 $i ... ;;
  *) echo "unexpected argument $i" ;;
esac
done

```

To allow the same commands to be associated with more than one pattern the **case** command provides for alternative patterns separated by a **|**. For example,

```

case $i in
  -x|-y) ...
esac

```

is equivalent to

```

case $i in
  -[xy]) ...
esac

```

The usual quoting conventions apply so that

```

case $i in
  \?) ...

```

will match the character **?**.

2.3 Here Documents

The shell procedure *tel* in section 2.1 uses the file */usr/lib/telnet* to supply the data for *grep*. An alternative is to include this data within the shell procedure as a *here* document, as in

```

for i
do grep $i <<!
  ...
  fred mh0123
  bert mh0789
  ...
!
done

```

In this example the shell takes the lines between **<<!** and **!** as the standard input for *grep*. The string **!** is arbitrary, the document being terminated by a line that consists of the string following **<<**.

Parameters are substituted in the document before it is made available to *grep* as illustrated by the following procedure called *edg*.


```
ed $3 <<%
g/$1/s//$2/g
w
%
```

The call

```
edg string1 string2 file
```

is then equivalent to the command

```
ed file <<%
g/string1/s//string2/g
w
%
```

and changes all occurrences of *string1* in *file* to *string2*. Substitution can be prevented using \ to quote the special character \$ as in

```
ed $3 <<+
l,\$s/$1/$2/g
w
+
```

(This version of *edg* is equivalent to the first except that *ed* will print a ? if there are no occurrences of the string \$1.) Substitution within a *here* document may be prevented entirely by quoting the terminating string, for example,

```
grep $i <<\#
...
#
```

The document is presented without modification to *grep*. If parameter substitution is not required in a *here* document this latter form is more efficient.

2.4 Shell Variables

The shell provides string-valued variables. Variable names begin with a letter and consist of letters, digits and underscores. Variables may be given values by writing, for example,

```
user=fred box=m000 acct=mh0000
```

which assigns values to the variables *user*, *box* and *acct*. A variable may be set to the null string by saying, for example,

```
null=
```

The value of a variable is substituted by preceding its name with \$; for example,

```
echo $user
```

will echo *fred*.

Variables may be used interactively to provide abbreviations for frequently used strings. For example,

```
b=/usr/fred/bin
mv pgm $b
```

will move the file *pgm* from the current directory to the directory */usr/fred/bin*. A more general notation is available for parameter (or variable) substitution, as in

```
echo ${user}
```

which is equivalent to:

```
echo $user
```

and is used when the parameter name is followed by a letter or digit. For example,

```
tmp=/tmp/ps
ps a >${tmp}a
```

will direct the output of *ps* to the file */tmp/psa*, whereas,

```
ps a >$tmpa
```

would cause the value of the variable *tmpa* to be substituted.

Except for **\$?** the following are set initially by the shell. **\$?** is set after executing each command.

\$? The exit status (return code) of the last command executed as a decimal string. Most commands return a zero exit status if they complete successfully, otherwise a non-zero exit status is returned. Testing the value of return codes is dealt with later under *if* and *while* commands.

\$# The number of positional parameters (in decimal). Used, for example, in the *append* command to check the number of parameters.

\$\$ The process number of this shell (in decimal). Since process numbers are unique among all existing processes, this string is frequently used to generate unique temporary file names. For example,

```
ps a >/tmp/ps$$
...
rm /tmp/ps$$
```

\$! The process number of the last process run in the background (in decimal).

\$- The current shell flags, such as *-x* and *-v*.

Some variables have a special meaning to the shell and should be avoided for general use.

\$MAIL When used interactively the shell looks at the file specified by this variable before it issues a prompt. If the specified file has been modified since it was last looked at the shell prints the message *you have mail* before prompting for the next command. This variable is typically set in the file *.profile*, in the user's login directory. For example,

```
MAIL=/usr/mail/fred
```

\$HOME The default argument for the *cd* command. The current directory is used to resolve file name references that do not begin with a */*, and is changed using the *cd* command. For example,

```
cd /usr/fred/bin
```

makes the current directory */usr/fred/bin*.

```
cat wn
```

will print on the terminal the file *wn* in this directory. The command *cd* with no argument is equivalent to

```
cd $HOME
```

This variable is also typically set in the the user's login profile.

\$PATH A list of directories that contain commands (the *search path*). Each time a command is executed by the shell a list of directories is searched for an executable

file. If `$PATH` is not set then the current directory, `/bin`, and `/usr/bin` are searched by default. Otherwise `$PATH` consists of directory names separated by `:`. For example,

```
PATH=:/usr/fred/bin:/bin:/usr/bin
```

specifies that the current directory (the null string before the first `:`), `/usr/fred/bin`, `/bin` and `/usr/bin` are to be searched in that order. In this way individual users can have their own 'private' commands that are accessible independently of the current directory. If the command name contains a `/` then this directory search is not used; a single attempt is made to execute the command.

- \$PS1** The primary shell prompt string, by default, `'$ '`.
- \$PS2** The shell prompt when further input is needed, by default, `'> '`.
- \$IFS** The set of characters used by *blank interpretation* (see section 3.4).

2.5 The test Command

The `test` command, although not part of the shell, is intended for use by shell programs. For example,

```
test -f file
```

returns zero exit status if `file` exists and non-zero exit status otherwise. In general `test` evaluates a predicate and returns the result as its exit status. Some of the more frequently used `test` arguments are given here, see `test (1)` for a complete specification.

```
test s           true if the argument s is not the null string
test -f file     true if file exists
test -r file     true if file is readable
test -w file     true if file is writable
test -d file     true if file is a directory
```

2.6 Control Flow — while

The actions of the `for` loop and the `case` branch are determined by data available to the shell. A `while` or `until` loop and an `if then else` branch are also provided whose actions are determined by the exit status returned by commands. A `while` loop has the general form

```
while command-list,
do command-list,
done
```

The value tested by the `while` command is the exit status of the last simple command following `while`. Each time round the loop `command-list1` is executed; if a zero exit status is returned then `command-list2` is executed; otherwise, the loop terminates. For example,

```
while test $1
do ...
  shift
done
```

is equivalent to

```
for i
do ...
done
```

`shift` is a shell command that renames the positional parameters `$2`, `$3`, ... as `$1`, `$2`, ... and loses `$1`.

Another kind of use for the **while/until** loop is to wait until some external event occurs and then run some commands. In an **until** loop the termination condition is reversed. For example,

```
until test -f file
do sleep 300; done
commands
```

will loop until *file* exists. Each time round the loop it waits for 5 minutes before trying again. (Presumably another process will eventually create the file.)

2.7 Control Flow—**if**

Also available is a general conditional branch of the form,

```
if command-list
then  command-list
else  command-list
fi
```

that tests the value returned by the last simple command following **if**.

The **if** command may be used in conjunction with the *test* command to test for the existence of a file as in

```
if test -f file
then  process file
else  do something else
fi
```

An example of the use of **if**, **case** and **for** constructions is given in section 2.10.

A multiple test **if** command of the form

```
if ...
then ...
else if ...
      then ...
      else if ...
            ...
            fi
      fi
fi
```

may be written using an extension of the **if** notation as,

```
if ...
then ...
elif ...
then ...
elif ...
...
fi
```

The following example is the *touch* command which changes the 'last modified' time for a list of files. The command may be used in conjunction with *make* (1) to force recompilation of a list of files.

```

flag=
for i
do case $i in
  -c) flag=N ;;
  *) if test -f $i
     then ln $i junk$$; rm junk$$
     elif test $flag
     then echo file \"$i\" does not exist
     else >$i
     fi
  esac
done

```

The `-c` flag is used in this command to force subsequent files to be created if they do not already exist. Otherwise, if the file does not exist, an error message is printed. The shell variable `flag` is set to some non-null string if the `-c` argument is encountered. The commands

```
ln ...; rm ...
```

make a link to the file and then remove it thus causing the last modified date to be updated.

The sequence

```

if command1
then command2
fi

```

may be written

```
command1 && command2
```

Conversely,

```
command1 || command2
```

executes `command2` only if `command1` fails. In each case the value returned is that of the last simple command executed.

2.8 Command Grouping

Commands may be grouped in two ways,

```
{ command-list ; }
```

and

```
( command-list )
```

In the first `command-list` is simply executed. The second form executes `command-list` as a separate process. For example,

```
(cd x; rm junk )
```

executes `rm junk` in the directory `x` without changing the current directory of the invoking shell.

The commands

```
cd x; rm junk
```

have the same effect but leave the invoking shell in the directory `x`.

2.9 Debugging Shell Procedures

The shell provides two tracing mechanisms to help when debugging shell procedures. The first is invoked within the procedure as

```
set -v
```

(*v* for verbose) and causes lines of the procedure to be printed as they are read. It is useful to help isolate syntax errors. It may be invoked without modifying the procedure by saying

```
sh -v proc ...
```

where *proc* is the name of the shell procedure. This flag may be used in conjunction with the *-n* flag which prevents execution of subsequent commands. (Note that saying *set -n* at a terminal will render the terminal useless until an end-of-file is typed.)

The command

```
set -x
```

will produce an execution trace. Following parameter substitution each command is printed as it is executed. (Try these at the terminal to see what effect they have.) Both flags may be turned off by saying

```
set -
```

and the current setting of the shell flags is available as *\$-*.

2.10 The *man* Command

The following is the *man* command which is used to print entries from the UNIX manual. It is called, for example, as:

```
man sh
man -t ed
man 2 fork
```

The first prints the manual entry for *sh*; because no section of the manual is specified, all sections of the manual are searched and the entry is found in Section 1. The second example typesets (*-t* option) the manual entry for *ed*. The last prints the *fork* manual entry from Section 2.

```

cd /usr/man
: `colon is the comment command`
: `default is nroff ($N), section 1 ($s)`
N=n s=1
for i
do case $i in
  [1-9]*)    s=$i ;;
  -t) N=t ;;
  -n) N=n ;;
  -*) echo unknown flag `"$i"` ;;
  *) if test -f man$s/$i.$s
     then  ${N}roff man0/${N}aa man$s/$i.$s
        else : `look through all manual sections`
           found=no
           for j in 1 2 3 4 5 6 7 8 9
           do if test -f man$j/$i.$j
              then man $j $i
                 found=yes
           fi
           done
           case $found in
             no) echo `"$i": manual page not found`
           esac
        fi
      esac
done

```

Figure 1. A version of the man command

3.0 KEYWORD PARAMETERS

Shell variables may be given values by assignment or when a shell procedure is invoked. An argument to a shell procedure of the form *name=value* that precedes the command name causes *value* to be assigned to *name* before execution of the procedure begins. The value of *name* in the invoking shell is not affected. For example,

```
user=fred command
```

will execute *command* with *user* set to *fred*. The *-k* flag causes arguments of the form *name=value* to be interpreted in this way anywhere in the argument list. Such *names* are sometimes called keyword parameters. If any arguments remain they are available as positional parameters *\$1*, *\$2*,

The *set* command may also be used to set positional parameters from within a procedure. For example,

```
set - *
```

will set *\$1* to the first file name in the current directory, *\$2* to the next, and so on. Note that the first argument, *-*, ensures correct treatment when the first file name begins with a *-*.

3.1 Parameter Transmission

When a shell procedure is invoked both positional and keyword parameters may be supplied with the call. Keyword parameters are also made available implicitly to a shell procedure by specifying in advance that such parameters are to be exported. For example,

```
export user box
```

marks the variables **user** and **box** for export. When a shell procedure is invoked copies are made of all exportable variables for use within the invoked procedure. Modification of such variables within the procedure does not affect the values in the invoking shell. It is generally true of a shell procedure that it may not modify the state of its caller without explicit request on the part of the caller. (Shared file descriptors are an exception to this rule.)

Names whose value is intended to remain constant may be declared *readonly*. The form of this command is the same as that of the *export* command,

```
readonly name ...
```

Subsequent attempts to set readonly variables are illegal.

3.2 Parameter Substitution

If a shell parameter is not set then the null string is substituted for it. For example, if the variable **d** is not set

```
echo $d
```

or

```
echo ${d}
```

will echo nothing. A default string may be given as in

```
echo ${d-.}
```

which will echo the value of the variable **d** if it is set and **.** otherwise. The default string is evaluated using the usual quoting conventions so that

```
echo ${d-`*`}
```

will echo ***** if the variable **d** is not set. Similarly

```
echo ${d-$1}
```

will echo the value of **d** if it is set and the value (if any) of **\$1** otherwise. A variable may be assigned a default value using the notation

```
echo ${d=.
```

which substitutes the same string as

```
echo ${d-.}
```

and if **d** were not previously set then it will be set to the string **.**. (The notation **\${... = ...}** is not available for positional parameters.)

If there is no sensible default then the notation

```
echo ${d?message}
```

will echo the value of the variable **d** if it has one, otherwise *message* is printed by the shell and execution of the shell procedure is abandoned. If *message* is absent then a standard message is printed. A shell procedure that requires some parameters to be set might start as follows.

```
: ${user?} ${acct?} ${bin?}
...
```

Colon (:) is a command that is built in to the shell and does nothing once its arguments have been evaluated. If any of the variables **user**, **acct** or **bin** are not set then the shell will abandon execution of the procedure.

3.3 Command Substitution

The standard output from a command can be substituted in a similar way to parameters. The command *pwd* prints on its standard output the name of the current directory. For example, if the current directory is */usr/fred/bin* then the command

```
d=`pwd`
```

is equivalent to

```
d=/usr/fred/bin
```

The entire string between grave accents (``...``) is taken as the command to be executed and is replaced with the output from the command. The command is written using the usual quoting conventions except that a ``` must be escaped using a `\`. For example,

```
ls `echo "$1"`
```

is equivalent to

```
ls $1
```

Command substitution occurs in all contexts where parameter substitution occurs (including *here* documents) and the treatment of the resulting text is the same in both cases. This mechanism allows string processing commands to be used within shell procedures. An example of such a command is *basename* which removes a specified suffix from a string. For example,

```
basename main.c .c
```

will print the string *main*. Its use is illustrated by the following fragment from a *cc* command.

```
case $A in
  ...
  *.c)      B=`basename $A .c`
  ...
esac
```

that sets **B** to the part of **\$A** with the suffix *.c* stripped.

Here are some composite examples.

- `for i in `ls -t`; do ...`
The variable *i* is set to the names of files in time order, most recent first.
- `set `date`; echo $6 $2 $3, $4`
will print, e.g., *1977 Nov 1, 23:59:59*

3.4 Evaluation and Quoting

The shell is a macro processor that provides parameter substitution, command substitution and file name generation for the arguments to commands. This section discusses the order in which these evaluations occur and the effects of the various quoting mechanisms.

Commands are parsed initially according to the grammar given in appendix A. Before a command is executed the following substitutions occur.

- parameter substitution, e.g. `$user`
- command substitution, e.g. ``pwd``

Only one evaluation occurs so that if, for example, the value of the variable **X** is the string *\$y* then

```
echo $X
```

will echo *\$y*.

- blank interpretation

Following the above substitutions the resulting characters are broken into non-blank words (*blank interpretation*). For this purpose 'blanks' are the characters of the string `$IFS`. By default, this string consists of blank, tab and new-line. The null string is not regarded as a word unless it is quoted. For example,

```
echo ""
```

will pass on the null string as the first argument to *echo*, whereas

```
echo $null
```

will call *echo* with no arguments if the variable `null` is not set or set to the null string.

- file name generation

Each word is then scanned for the file pattern characters `*`, `?` and `[...]` and an alphabetical list of file names is generated to replace the word. Each such file name is a separate argument.

The evaluations just described also occur in the list of words associated with a **for** loop. Only substitution occurs in the *word* used for a **case** branch.

As well as the quoting mechanisms described earlier using `\` and `"..."` a third quoting mechanism is provided using double quotes. Within double quotes parameter and command substitution occurs but file name generation and the interpretation of blanks does not. The following characters have a special meaning within double quotes and may be quoted using `\`.

<code>\$</code>	parameter substitution
<code>`</code>	command substitution
<code>*</code>	ends the quoted string
<code>\</code>	quotes the special characters <code>\$ ` * \</code>

For example,

```
echo "$x"
```

will pass the value of the variable `x` as a single argument to *echo*. Similarly,

```
echo "$@"
```

will pass the positional parameters as a single argument and is equivalent to

```
echo "$1 $2 ..."
```

The notation `$@` is the same as `$*` except when it is quoted.

```
echo "$@"
```

will pass the positional parameters, unevaluated, to *echo* and is equivalent to

```
echo "$1" "$2" ...
```

The following table gives, for each quoting mechanism, the shell metacharacters that are evaluated.

	<i>metacharacter</i>					
	\	\$	*	~	"	'
\	n	n	n	n	n	t
\$	y	n	n	t	n	n
*	y	y	n	y	t	n

t terminator
 y interpreted
 n not interpreted

Figure 2. Quoting mechanisms

In cases where more than one evaluation of a string is required the built-in command *eval* may be used. For example, if the variable *X* has the value *\$y*, and if *y* has the value *pqr* then

```
eval echo $X
```

will echo the string *pqr*.

In general the *eval* command evaluates its arguments (as do all commands) and treats the result as input to the shell. The input is read and the resulting command(s) executed. For example,

```
wg=`eval who |grep`
$wg fred
```

is equivalent to

```
who |grep fred
```

In this example, *eval* is required since there is no interpretation of metacharacters, such as *|*, following substitution.

3.5 Error Handling

The treatment of errors detected by the shell depends on the type of error and on whether the shell is being used interactively. An interactive shell is one whose input and output are connected to a terminal (as determined by *gty* (2)). A shell invoked with the *-i* flag is also interactive.

Execution of a command (see also 3.7) may fail for any of the following reasons.

- Input/output redirection may fail. For example, if a file does not exist or cannot be created.
- The command itself does not exist or cannot be executed.
- The command terminates abnormally, for example, with a "bus error" or "memory fault". See Figure 2 below for a complete list of UNIX signals.
- The command terminates normally but returns a non-zero exit status.

In all of these cases the shell will go on to execute the next command. Except for the last case an error message will be printed by the shell. All remaining errors cause the shell to exit from a command procedure. An interactive shell will return to read another command from the terminal. Such errors include the following.

- Syntax errors. e.g., *if ... then ... done*
- A signal such as interrupt. The shell waits for the current command, if any, to finish execution and then either exits or returns to the terminal.
- Failure of any of the built-in commands such as *cd*.

The shell flag *-e* causes the shell to terminate if any error is detected.

1	hangup
2	interrupt
3*	quit
4*	illegal instruction
5*	trace trap
6*	IOT instruction
7*	EMT instruction
8*	floating point exception
9	kill (cannot be caught or ignored)
10*	bus error
11*	segmentation violation
12*	bad argument to system call
13	write on a pipe with no one to read it
14	alarm clock
15	software termination (from <i>kill</i> (1))

Figure 3. UNIX signals

Those signals marked with an asterisk produce a core dump if not caught. However, the shell itself ignores quit which is the only external signal that can cause a dump. The signals in this list of potential interest to shell programs are 1, 2, 3, 14 and 15.

3.6 Fault Handling

Shell procedures normally terminate when an interrupt is received from the terminal. The *trap* command is used if some cleaning up is required, such as removing temporary files. For example,

```
trap `rm /tmp/ps$$; exit` 2
```

sets a trap for signal 2 (terminal interrupt), and if this signal is received will execute the commands

```
rm /tmp/ps$$; exit
```

exit is another built-in command that terminates execution of a shell procedure. The *exit* is required; otherwise, after the trap has been taken, the shell will resume executing the procedure at the place where it was interrupted.

UNIX signals can be handled in one of three ways. They can be ignored, in which case the signal is never sent to the process. They can be caught, in which case the process must decide what action to take when the signal is received. Lastly, they can be left to cause termination of the process without it having to take any further action. If a signal is being ignored on entry to the shell procedure, for example, by invoking it in the background (see 3.7) then *trap* commands (and the signal) are ignored.

The use of *trap* is illustrated by this modified version of the *touch* command (Figure 4). The cleanup action is to remove the file *junk\$\$*.

```

flag=
trap `rm -f junk$$; exit` 1 2 3 15
for i
do case $i in
  -c) flag=N ;;
  *) if test -f $i
     then ln $i junk$$; rm junk$$
     elif test $flag
     then echo file \"$i\" does not exist
     else >$i
     fi
  esac
done

```

Figure 4. The touch command

The *trap* command appears before the creation of the temporary file; otherwise it would be possible for the process to die without removing the file.

Since there is no signal 0 in UNIX it is used by the shell to indicate the commands to be executed on exit from the shell procedure.

A procedure may, itself, elect to ignore signals by specifying the null string as the argument to trap. The following fragment is taken from the *nohup* command.

```
trap "" 1 2 3 15
```

which causes *hangup*, *interrupt*, *quit* and *kill* to be ignored both by the procedure and by invoked commands.

Traps may be reset by saying

```
trap 2 3
```

which resets the traps for signals 2 and 3 to their default values. A list of the current values of traps may be obtained by writing

```
trap
```

The procedure *scan* (Figure 5) is an example of the use of *trap* where there is no exit in the trap command. *scan* takes each directory in the current directory, prompts with its name, and then executes commands typed at the terminal until an end of file or an interrupt is received. Interrupts are ignored while executing the requested commands but cause termination when *scan* is waiting for input.

```

d=`pwd`
for i in *
do if test -d $d/$i
  then cd $d/$i
    while echo "$i:"
      trap exit 2
      read x
    do trap : 2; eval $x; done
  fi
done

```

Figure 5. The scan command

read x is a built-in command that reads one line from the standard input and places the result in the variable *x*. It returns a non-zero exit status if either an end-of-file is read or an interrupt is received.

3.7 Command Execution

To run a command (other than a built-in) the shell first creates a new process using the system call *fork*. The execution environment for the command includes input, output and the states of signals, and is established in the child process before the command is executed. The built-in command *exec* is used in the rare cases when no fork is required and simply replaces the shell with a new command. For example, a simple version of the *nohup* command looks like

```
trap "" 1 2 3 15
exec $*
```

The *trap* turns off the signals specified so that they are ignored by subsequently created commands and *exec* replaces the shell by the command specified.

Most forms of input/output redirection have already been described. In the following *word* is only subject to parameter and command substitution. No file name generation or blank interpretation takes place so that, for example,

```
echo ... >*.c
```

will write its output into a file whose name is **.c*. Input/output specifications are evaluated left to right as they appear in the command.

- > *word* The standard output (file descriptor 1) is sent to the file *word* which is created if it does not already exist.
- >> *word* The standard output is sent to file *word*. If the file exists then output is appended (by seeking to the end); otherwise the file is created.
- < *word* The standard input (file descriptor 0) is taken from the file *word*.
- << *word* The standard input is taken from the lines of shell input that follow up to but not including a line consisting only of *word*. If *word* is quoted then no interpretation of the document occurs. If *word* is not quoted then parameter and command substitution occur and \ is used to quote the characters \ \$ ` and the first character of *word*. In the latter case \new-line is ignored (c.f. quoted strings).
- >& *digit* The file descriptor *digit* is duplicated using the system call *dup* (2) and the result is used as the standard output.
- <& *digit* The standard input is duplicated from file descriptor *digit*.
- <&- The standard input is closed.
- >&- The standard output is closed.

Any of the above may be preceded by a digit in which case the file descriptor created is that specified by the digit instead of the default 0 or 1. For example,

```
... 2>file
```

runs a command with message output (file descriptor 2) directed to *file*.

```
... 2>&1
```

runs a command with its standard output and message output merged. (Strictly speaking file descriptor 2 is created by duplicating file descriptor 1 but the effect is usually to merge the two streams.)

The environment for a command run in the background such as

```
list *.c | lpr &
```

is modified in two ways. Firstly, the default standard input for such a command is the empty file */dev/null*. This prevents two processes (the shell and the command), which are running in parallel, from trying to read the same input. Chaos would ensue if this were not the case. For example,

ed file &

would allow both the editor and the shell to read from the same input at the same time.

The other modification to the environment of a background command is to turn off the QUIT and INTERRUPT signals so that they are ignored by the command. This allows these signals to be used at the terminal without causing background commands to terminate. For this reason the UNIX convention for a signal is that if it is set to 1 (ignored) then it is never changed even for a short time. Note that the shell command *trap* has no effect for an ignored signal.

3.8 Invoking the Shell

The following flags are interpreted by the shell when it is invoked. If the first character of argument zero is a minus, then commands are read from the file *.profile*.

- c *string* If the —c flag is present then commands are read from *string*.
- s If the —s flag is present or if no arguments remain then commands are read from the standard input. Shell output is written to file descriptor 2.
- i If the —i flag is present or if the shell input and output are attached to a terminal (as told by *gty*) then this shell is *interactive*. In this case TERMINATE is ignored (so that *kill 0* does not kill an interactive shell) and INTERRUPT is caught and ignored (so that *wait* is interruptable). In all cases QUIT is ignored by the shell.

Acknowledgements

The design of the shell is based in part on the original UNIX shell³ and the PWB/UNIX shell,⁴ some features having been taken from both. Similarities also exist with the command interpreters of the Cambridge Multiple Access System⁵ and of CTSS.⁶

I would like to thank Dennis Ritchie and John Mashey for many discussions during the design of the shell. I am also grateful to the members of the Computing Science Research Center and to Joe Maranzano for their comments on drafts of this document.

References

- [1] B. W. Kernighan, *UNIX for Beginners*, Bell Laboratories (1978).
- [2] T. A. Dolotta, S. B. Olsson, and A. G. Petruccelli (eds.), *UNIX User's Manual—Release 3.0*, Bell Laboratories (June 1980).
- [3] K. Thompson, "The UNIX Command Language," *Structured Programming—Infotech State of the Art Report*, pp. 375-384, Infotech International Ltd., Nicholson House, Maidenhead, Berkshire, England (March 1975).
- [4] J. R. Mashey, *PWB/UNIX Shell Tutorial*, Bell Laboratories (September 1977).
- [5] D. F. Hartley (ed.), *The Cambridge Multiple Access System—Users Reference Manual*, University Mathematical Laboratory, Cambridge, England (1968).
- [6] P. A. Crisman (ed.), *The Compatible Time-Sharing System*, M.I.T. Press, Cambridge, Mass. (1965).

Appendix A—Grammar

<i>item:</i>	<i>word</i> <i>input-output</i> <i>name = value</i>
<i>simple-command:</i>	<i>item</i> <i>simple-command item</i>
<i>command:</i>	<i>simple-command</i> (<i>command-list</i>) { <i>command-list</i> } for name do <i>command-list</i> done for name in word ... do <i>command-list</i> done while <i>command-list</i> do <i>command-list</i> done until <i>command-list</i> do <i>command-list</i> done case word in case-part ... esac if <i>command-list</i> then <i>command-list</i> else-part fi
<i>pipeline:</i>	<i>command</i> <i>pipeline command</i>
<i>andor:</i>	<i>pipeline</i> <i>andor && pipeline</i> <i>andor pipeline</i>
<i>command-list:</i>	<i>andor</i> <i>command-list ;</i> <i>command-list &</i> <i>command-list ; andor</i> <i>command-list & andor</i>
<i>input-output:</i>	> <i>file</i> < <i>file</i> >> <i>word</i> << <i>word</i>
<i>file:</i>	<i>word</i> & <i>digit</i> & -
<i>case-part:</i>	<i>pattern</i>) <i>command-list</i> ;;
<i>pattern:</i>	<i>word</i> <i>pattern</i> <i>word</i>
<i>else-part:</i>	elif <i>command-list</i> then <i>command-list</i> else-part else <i>command-list</i> <i>empty</i>
<i>empty:</i>	
<i>word:</i>	a sequence of non-blank characters
<i>name:</i>	a sequence of letters, digits or underscores starting with a letter
<i>digit:</i>	0 1 2 3 4 5 6 7 8 9

Appendix B—Meta-characters and Reserved Words

a) syntactic

| pipe symbol
 && 'andf' symbol
 || 'orf' symbol
 ; command separator
 ;; case delimiter
 & background commands
 () command grouping
 < input redirection
 << input from a here document
 > output creation
 >> output append

b) patterns

* match any character(s) including none
 ? match any single character
 [...] match any of the enclosed characters

c) substitution

\${...} substitute shell variable
 `...` substitute command output

d) quoting

\ quote the next character
 `...` quote the enclosed characters except for ` `"
 "... " quote the enclosed characters except for \$ ` \ "

e) reserved words

if then else elif fi
 case in esac
 for while until do done
 { }

January 1981

A TROFF Tutorial

Brian W. Kernighan

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Troff is a text-formatting program for driving a phototypesetter on the UNIX† and GCOS operating systems to produce high-quality printed text; this paper is an example of **troff** output.

The phototypesetter itself normally runs with four fonts, containing roman, italic and bold letters (as on this page), a full Greek alphabet, and a substantial number of special characters and mathematical symbols. Characters can be printed in a range of sizes, and placed anywhere on the page.

Troff allows the user full control over fonts, sizes, and character positions, as well as the usual features of a formatter—right-margin justification, automatic hyphenation, page titling and numbering, and so on. It also provides macros, arithmetic variables and operations, and conditional testing, for more complicated formatting tasks.

This document is an introduction to the most basic use of **troff**. It presents just enough information to enable the user to do simple formatting tasks such as making view graphs, and to make incremental changes to existing packages of **troff** macros. In most respects, the UNIX formatter **nroff** is identical to **troff**, so this document also serves as a tutorial on **nroff**.

1. Introduction

Troff [1] is a text-formatting program for phototypesetting high-quality, printed output on the UNIX and GCOS operating systems. This document is an example of **troff** output.

The single most important rule of using **troff** is *not* to use it directly, but through some intermediary. In many ways, **troff** resembles an assembly language—a remarkably powerful and flexible one—but nonetheless such that many operations must be specified at a level of detail and in a form that is too hard for most people to use effectively.

For two special applications, there are programs that provide an interface to **troff** for the majority of users. **Eqn** [2] provides an easy to learn language for typesetting mathematics; the **eqn** user need know no **troff** whatsoever to typeset mathematics. **Tbl** [3] provides the same convenience for producing tables of arbitrary complexity.

For producing straight text (which may well contain mathematics or tables), there are a number of “macro packages” that define formatting rules and operations for specific styles of documents, and reduce the amount of direct contact with **troff**. In particular, the “ms” [4] and MM [5] packages for Bell Labs internal memoranda and external papers provide most of the facilities needed for a wide range of document preparation. There are also packages for view graphs and for other special applications. Typically you will find these packages easier to use than **troff** once you get beyond the most trivial operations; you should always consider them first.

In the few cases where existing packages don’t do the whole job, the solution is *not* to write an entirely new set of **troff** instructions from scratch, but to make small changes to adapt packages that already exist.

† UNIX is a trademark of Bell Laboratories.

In accordance with this philosophy of letting someone else do the work, the part of **troff** described here is only a small part of the whole, although it tries to concentrate on the more useful parts. In any case, there is no attempt to be complete. Rather, the emphasis is on showing how to do simple things, and how to make incremental changes to what already exists. The contents of the remaining sections are:

2. Point Sizes and Line Spacing
 3. Fonts and Special Characters
 4. Indents and Line Lengths
 5. Tabs
 6. Local Motions: Drawing Lines and Characters
 7. Strings
 8. Introduction to Macros
 9. Titles, Pages, and Numbering
 10. Number Registers and Arithmetic
 11. Macros with Arguments
 12. Conditionals
 13. Environments
 14. Diversions
- Appendix: Typesetter Character Set

The **troff** described here is the C-language version running on UNIX, as documented in [1].

To use **troff** you have to prepare not only the actual text you want printed, but some information that tells *how* you want it printed. For **troff** the text and the formatting information are often intertwined quite intimately. Most **troff** *commands* (sometimes referred to as *requests*) are placed on a line separate from the text itself, beginning with a period (one command per line). For example,

```
Some text.
.ps 14
Some more text.
```

will change the "point size", that is, the size of the letters being printed, to "14-point" (one point is 1/72 inch) like this:

```
Some text. Some more text.
```

Occasionally, though, something special occurs in the middle of a line—to produce

$$\text{Area} = \pi r^2$$

you have to type

```
Area = \>(*p\fr\fr\|s8\u2\d\s0
```

(which we will explain shortly). The backslash character, \ is used to introduce **troff** commands and special characters within a line of text.

2. Point Sizes and Line Spacing

As mentioned above, the command **.ps** sets the point size. One point is 1/72 inch, so 6-point characters are at most 1/12 inch high, and 36-point characters are 1/2 inch. There are 15 point sizes, listed below:

```
6 point: Pack my box with five dozen liquor jugs.
7 point: Pack my box with five dozen liquor jugs.
8 point: Pack my box with five dozen liquor jugs.
9 point: Pack my box with five dozen liquor jugs.
10 point: Pack my box with five dozen liquor
11 point: Pack my box with five dozen
12 point: Pack my box with five dozen
14 point: Pack my box with five
16 point 18 point 20 point
22 24 28 36
```

If the number after **.ps** is not one of these legal sizes, it is rounded up to the next valid value, with a maximum of 36. If no number follows **.ps**, **troff** reverts to the previous size, whatever it was. **Troff** begins with point size 10, which is usually fine. This document is in 9-point.

The point size can also be changed in the middle of a line or even a word with the in-line command **\s**. To produce

```
UNIX runs on a VAX-11/780
```

type

```
\s8UNIX\s10 runs on a \s8VAX-\s1011/780
```

As above, **\s** should be followed by a legal point size, except that **\s0** causes the size to revert to its previous value. Notice that **\s1011** can be understood correctly as "size 10, followed by an 11", if the size is legal, but not otherwise. Be cautious with similar constructions.

Relative size changes are also legal and useful:

```
\s-2UNIX\s+2
```

temporarily decreases the size, whatever it is, by two points, then restores it. Relative size changes have the advantage that the size difference is independent of the starting size of the document. The amount of the relative change is restricted to a single digit.

The other parameter that determines what the type looks like is the spacing between lines, which is set independently of the point size. Vertical spacing is measured from the bottom of one line to the bottom of the next. The command to control vertical spacing is **.vs**. For running text, it is usually best to set the vertical

spacing about 20% bigger than the character size. For example, so far in this document, we have used "9 on 11", that is,

```
.ps 9
.vs 11p
```

If we changed to

```
.ps 9
.vs 9p
```

the running text would look like this. After a few lines, you will agree it looks a little cramped. The right vertical spacing is partly a matter of taste, depending on how much text you want to squeeze into a given space, and partly a matter of traditional printing style. By default, **troff** uses 10 on 12.

Point size and vertical spacing make a substantial difference in the amount of text per square inch. This is 12 on 14.

Point size and vertical spacing make a substantial difference in the amount of text per square inch. For example, 10 on 12 uses about twice as much space as 7 on 8. This is 6 on 7, which is even smaller. It packs a lot more words per line, but you can go blind trying to read it.

When used without arguments, **.ps** and **.vs** revert to the previous size and vertical spacing respectively.

The command **.sp** is used to get extra vertical space. Unadorned, it gives you one extra blank line (one **.vs**, whatever that has been set to). Typically, that's more or less than you want, so **.sp** can be followed by information about how much space you want:

```
.sp 2i
```

means "two inches of vertical space".

```
.sp 2p
```

means "two points of vertical space"; and

```
.sp 2
```

means "two vertical spaces"—two of whatever **.vs** is set to (this can also be made explicit with **.sp 2v**); **troff** also understands decimal fractions in most places, so

```
.sp 1.5i
```

is a space of 1.5 inches. These same scale factors can be used after **.vs** to define line spacing, and in fact after most commands that deal with physical dimensions.

It should be noted that all size numbers are converted internally to "machine units", which are 1/432 inch (1/6 point). For most purposes, this is enough resolution that you don't have to worry about the accuracy of the representation. The situation is not quite so good vertically, where resolution is 1/144 inch (1/2 point).

3. Fonts and Special Characters

Troff and the typesetter allow four different fonts at any one time. Normally three fonts (Times roman, italic, and bold) and one collection of special characters are permanently mounted:

```
abcdefghijklmnopqrstuvwxyz 0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz 0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz 0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

The Greek, mathematical, and other symbols on the special font are listed in the Appendix.

Troff prints in roman unless told otherwise. To switch into bold, use the **.ft** command

```
.ft B
```

and for italics,

```
.ft I
```

To return to roman, use **.ft R**; to return to the previous font, whatever it was, use either **.ft P** or just **.ft**. The "underline" command

```
.ul
```

causes the next input line to print in italics. **.ul** can be followed by a count to indicate that more than one line is to be italicized.

Fonts can also be changed within a line or word with the in-line command **\f**:

```
bold/ace text
```

is produced by

```
\fBbold\fiface\fR text
```

If you want to do this so the previous font, whatever it was, is left undisturbed (a good practice), insert extra **\FP** commands, like this:

```
\fBbold\fP\fiface\fP\fR text\fP
```

Because only the immediately previous font is remembered, you have to restore the previous font after each change or you can lose it. The same is true of **.ps** and **.vs** when used without an argument.

There are other fonts available besides the standard set, although you can still use only four at any given time. The command **.fp** tells **troff** what fonts are physically mounted on the typesetter:

```
.fp 3 H
```

says that the Helvetica font is mounted on position 3. (For a complete list of fonts and what they look like, see the **troff** manual.) Appropriate **.fp** commands should appear at the beginning

of your document if you do not use the standard fonts.

It is possible to make a document relatively independent of the actual fonts used to print it by using font numbers instead of names; for example, `\f3` and `.ft 3` mean "whatever font is mounted at position 3", and thus work for any setting. Normal settings are roman font on 1, italic on 2, bold on 3, and special on 4.

There is also a way to get "synthetic" bold fonts by overstriking letters with a slight offset. Look at the `.bd` command in [1].

Special characters have four-character names beginning with `\`, and they may be inserted anywhere. For example,

$$\frac{1}{4} + \frac{1}{2} = \frac{3}{4}$$

is produced by

```
\(14 + \ (12 = \ (34
```

In particular, Greek letters are all of the form `\(*?`, where `?` is an upper- or lower-case Roman letter reminiscent of the Greek. Thus to get

$$\Sigma(\alpha \times \beta) \rightarrow \infty$$

in bare troff we have to type

```
\(*S\(*a\(\mu\(*b)\(->\(if
```

That line is unscrambled as follows:

<code>\(*S</code>	Σ
<code>(</code>	<code>(</code>
<code>\(*a</code>	α
<code>\(\mu</code>	\times
<code>\(*b</code>	β
<code>)</code>	<code>)</code>
<code>\(-></code>	\rightarrow
<code>\(if</code>	∞

A complete list of these special names is given in the Appendix.

In `eqn` [2] the same effect can be achieved with the input

```
SIGMA ( alpha times beta ) -> inf
```

which is less concise, but clearer to the uninitiated.

Notice that each four-character name is a single character as far as troff is concerned—the "translate" command

```
.tr \(\mi\(\em
```

is perfectly clear, meaning

```
.tr —
```

that is, to translate `—` into `—`.

Some characters are automatically translated into others: grave ``` and acute `´` accents

(apostrophes) become open and close single quotes `'`; the combination of `"..."` is generally preferable to the double quotes `"..."`. Similarly, a typed minus sign `-` becomes a hyphen `-`. To print an explicit `-` sign, use `\-`. To get a backslash printed, use `\e`.

4. Indents and Line Lengths

Troff starts with a line length of 6.5 inches, too wide for $8\frac{1}{2} \times 11$ inch paper. To reset the line length, use the `.ll` command, as in

```
.ll 6i
```

As with `.sp`, the actual length can be specified in several ways; inches are probably the most intuitive.

The maximum line length provided by the typesetter is 7.5 inches, by the way. To use the full width, you will have to reset the default physical left margin ("page offset"), which is normally slightly less than one inch from the left edge of the paper. This is done by the `.po` command.

```
.po 0
```

sets the offset as far to the left as it will go.

The indent command `.in` causes the left margin to be indented by some specified amount from the page offset. If we use `.in` to move the left margin in, and `.ll` to move the right margin to the left, we can make offset blocks of text:

```
.in 0.3i
.ll -0.3i
... text to be set as a block ...
.ll +0.3i
.in -0.3i
```

will create a block that looks like this:

```
Pater noster qui est in caelis
sanctificetur nomen tuum; adveniat
regnum tuum; fiat voluntas tua, sicut in
caelo, et in terra. ... Amen.
```

Notice the use of `+` and `-` to specify the amount of change. These change the previous setting by the specified amount, rather than just overriding it. The distinction is quite important: `.ll +1i` makes lines one inch longer; `.ll 1i` makes them one inch *long*.

With `.in`, `.ll` and `.po`, the previous value is used if no argument is specified.

To indent a single line, use the "temporary indent" command `.ti`. For example, all paragraphs in this memo effectively begin with the command

```
.ti 3
```

Three of what? The default unit for `.ti`, as for most horizontally oriented commands (`.ll`, `.in`, `.po`), is ems; an em is roughly the width of the letter "m" in the current point size. (Precisely, a em in size p is p points.) Although inches are usually clearer than ems to people who don't set type for a living, ems have a place: they are a measure of size that is proportional to the current point size. If you want to make text that keeps its proportions regardless of point size, you should use ems for all dimensions. Ems can be specified as scale factors directly, as in `.ti 2.5m`.

Lines can also be indented negatively if the indent is already positive:

```
.ti -0.3i
```

causes the next line to be moved back three tenths of an inch. Thus to make a decorative initial capital, we indent the whole paragraph, then move the letter "P" back with a `.ti` command:

```
Pater noster qui est in caelis
sanctificetur nomen tuum; adveniat
regnum tuum; fiat voluntas tua,
sicut in caelo, et in terra. ... Amen.
```

Of course, there is also some trickery to make the "P" bigger (just a "`\s36P\s0`"), and to move it down from its normal position (see Section 6 on local motions).

5. Tabs

Tabs (the ASCII "horizontal tab" character) can be used to produce output in columns, or to set the horizontal position of output. Typically tabs are used only in unfilled text. Tab stops are set by default every half inch from the current indent, but can be changed by the `.ta` command. To set stops every inch, for example,

```
.ta 1i 2i 3i 4i 5i 6i
```

Unfortunately the stops are left-justified only (as on a typewriter), so lining up columns of right-justified numbers can be painful. If you have many numbers, or if you need more complicated table layout, *don't* use `troff` directly; use the `tbl` program described in [3].

For a handful of numeric columns, you can do it this way: Precede every number by enough blanks to make it line up when typed.

```
.nf
.ta 1i 2i 3i
  1 tab  2 tab  3
 40 tab 50 tab 60
 700 tab 800 tab 900
.fi
```

Then change each leading blank into the string `\0`. This is a character that does not print, but that has the same width as a digit. When printed, this will produce

```
      1          2          3
    40         50         60
   700        800        900
```

It is also possible to fill up tabbed-over space with a character other than a blank by setting the "tab replacement character" with the `.tc` command:

```
.ta 1.5i 2.5i
.tc \ (ru      (\ (ru is "_")
Name tab Age tab
```

produces:

```
Name _____ Age _____
```

To reset the tab replacement character to a blank, use `.tc` without argument. (Lines can also be drawn with the `\l` command; see Section 6.)

`Troff` also provides a very general mechanism called "fields" for setting up complicated columns (it is used by `tbl`). We will not go into it in this paper.

6. Local Motions: Drawing Lines and Characters

Remember " $\text{Area} = \pi r^2$ " and the big "P" in the Paternoster. How are they done? `Troff` provides a host of commands for placing characters of any size at any place. You can use them to draw special characters or to tune your output for a particular appearance. Most of these commands are straightforward, but messy to read and tough to type correctly.

If you won't use `eqn`, subscripts and superscripts are most easily done with the half-line local motions `\u` and `\d`. To go back up the page half a point-size, insert a `\u` at the desired place; to go down, insert a `\d`. (`\u` and `\d` should always be used in pairs, as explained below.) Thus

```
Area = \(*pr\u2\d
```

produces:

```
Area =  $\pi r^2$ 
```

To make the "2" smaller, bracket it with `\s-2...s0`. Since `\u` and `\d` refer to the current point size, be sure to put them either both inside or both outside the size changes, or you will get an unbalanced vertical motion.

Sometimes the space given by `\u` and `\d` isn't the right amount. The `\v` command can be used to request an arbitrary amount of vertical motion. The in-line command

`\v'(amount)'`

causes motion up or down the page by the amount specified in "(amount)". For example, to move the "P" down, we used

```
.in +0.6i      (move paragraph in)
.ll -0.3i      (shorten lines)
.ti -0.3i      (move P back)
\v'2^s36P\s0\v'-2'ater noster qui est
in caelis ...
```

A minus sign causes upward motion, while no sign or a plus sign means down the page. Thus `\v'-2'` causes an upward vertical motion of two line spaces.

There are many other ways to specify the amount of motion:

```
\v'0.1i'
\v'3p'
\v'-0.5m'
```

and so on are all legal. Notice that the scale specifier `i` or `p` or `m` goes inside the quotes. Any character can be used in place of the quotes; this is also true of all other **troff** commands described in this section.

Since **troff** does not take within-the-line vertical motions into account when figuring out where it is on the page, output lines can have unexpected positions if the left and right ends aren't at the same vertical position. Thus `\v`, like `\u` and `\d`, should always balance upward vertical motion in a line with the same amount in the downward direction.

Arbitrary horizontal motions are also available: `\h` is quite analogous to `\v`, except that the default scale factor is ems instead of line spaces. As an example,

```
\h'-0.1i'
```

causes a backwards motion of a tenth of an inch. As a practical matter, consider printing the mathematical symbol "`>>`". The default spacing is too wide, so `eqn` replaces this by

```
>\h'-0.3m'>
```

to produce `>>`.

Frequently `\h` is used with the "width function" `\w` to generate motions equal to the width of some character string. The construction

```
\w'thing'
```

is a number equal to the width of "thing" in machine units (1/432 inch). All **troff** computations are ultimately done in these units. To move horizontally the width of an "x", we can say

```
\h\w'x'u'
```

As we mentioned above, the default scale factor for all horizontal dimensions is `m`, ems, so here we must have the `u` for machine units, or the motion produced will be far too large. **Troff** is quite happy with the nested quotes, by the way, so long as you don't leave any out.

As a live example of this kind of construction, all of the command names in the text, like `.sp`, were done by overstriking with a slight offset. The commands to print `.sp` are

```
.sp\h'-\w'.sp'u'h'2u'.sp
```

That is, put out "`.sp`", move left by the width of "`.sp`", move right 2 units, and print "`.sp`" again. (Of course there is a way to avoid typing that much input for each command name, which we will discuss in Section 11.)

There are also several special-purpose **troff** commands for local motion. In Section 5, we have already seen `\0`, which is an unpaddable white space of the same width as a digit. "Unpaddable" means that it will never be widened or split across a line by line justification and filling. There is also `\(blank)`, which is an unpaddable character the width of a space, `\|`, which is half that width, `\^`, which is one quarter of the width of a space, and `\&`, which has zero width; this last one may be used, for example, to "protect" from **troff** a line of text that begins with a ".".

The command `\o`, used like

```
\o'set of characters'
```

causes (up to 9) characters to be overstruck, centered on the widest. This is nice for accents, as in:

```
syst\o"e\"me t\o"e\"l\o"e\"phonique
```

which makes:

```
systeme téléphonique
```

The accents are `\`` and `\^`, or `\(ga` and `\(aa`; remember that each is just one character to **troff**.

You can make your own overstrikes with another special convention, `\z`, the zero-motion command; `\zx` suppresses the normal horizontal motion after printing the single character `x`, so another character can be laid on top of it. Although sizes can be changed within `\o`, it centers the characters on the widest, and there can be no horizontal or vertical motions, so `\z` may be the only way to get what you want:



of the definition. In between is the text, which is simply inserted whenever **troff** sees the "command" or macro call

```
.PP
```

A macro can contain any mixture of text and formatting commands.

The definition of **.PP** has to precede its first use; undefined macros are simply ignored. Names are restricted to one or two characters.

Using macros for commonly occurring sequences of commands is critically important. Not only does it save typing, but it makes later changes much easier. Suppose we decide that the paragraph indent is too small, the vertical space is much too big, and roman font should be forced. Instead of changing the whole document, we need only change the definition of **.PP** to something like

```
.de PP      \" paragraph macro
.sp 2p
.ti +3m
.ft R
..
```

and the change takes effect everywhere we used **.PP**.

***** is a **troff** command that causes the rest of the line to be ignored. We use it here to add comments to the macro definition (a wise idea once definitions get complicated).

As another example of macros, consider these two which start and end a block of offset, unfilled text, like most of the examples in this paper:

```
.de BS      \" start indented block
.sp
.nf
.in +0.3i
..
.de BE      \" end indented block
.sp
.fi
.in -0.3i
..
```

Now we can surround text like

```
Copy to
John Doe
Richard Roberts
Stanley Smith
```

by the commands **.BS** and **.BE**, and it will come out as it did above. Notice that we indented by **.in +0.3i** instead of **.in 0.3i**. This way we can nest our uses of **.BS** and **.BE** to get blocks within blocks.

If later on we decide that the indent should be 0.5i, then it is only necessary to change the definitions of **.BS** and **.BE**, not the whole paper.

9. Titles, Pages, and Numbering

This is an area where things get tougher, because nothing is done for you automatically. Of necessity, some of this section is a cookbook, to be copied literally until you get some experience.

Suppose you want a title at the top of each page, saying simply:

```
left top      center top      right top
```

It would be nice if one could just say:

```
.he 'left top'center top'right top'
.fo 'left bottom'center bottom'right bottom'
```

to get headers and footers automatically on every page (as was possible in an older system called **roff**). Alas, this doesn't work in **troff**, a serious hardship for the novice. Instead you have to do a lot of specification.

You have to say what the actual title is (easy); when to print it (easy enough); and what to do at and around the title line (harder). Taking these in reverse order, first we define a macro **.NP** (for "new page") to process titles and the like at the end of one page and the beginning of the next:

```
.de NP
'bp
'sp 0.5i
.tl 'left top'center top'right top'
'sp 0.3i
..
```

To make sure we're at the top of a page, we issue a "begin page" command **'bp**, which causes a skip to top-of-page (we'll explain the ' shortly). Then we space down half an inch, print the title (the use of **.tl** should be self explanatory; later we will discuss parameterizing the titles), space another 0.3 inches, and we're done.

To ask for **.NP** at the bottom of each page, we have to say something like "when the text is within an inch of the bottom of the page, start the processing for a new page". This is done with a "when" command **.wh**:

```
.wh -1i NP
```

(No "." is used before **NP**; this is simply the name of a macro, not a macro call.) The minus sign means "measure up from the bottom of the page", so "-1i" means "one inch from the bottom".

The `.wh` command appears in the input outside the definition of `.NP`; typically the input would be

```
.dc NP
...
..
.wh -li NP
```

Now what happens? As text is actually being output, `troff` keeps track of its vertical position on the page, and after a line is printed within one inch from the bottom, the `.NP` macro is activated. (In the jargon, the `.wh` command sets a *trap* at the specified place, which is “sprung” when that point is passed.) `.NP` causes a skip to the top of the next page (that’s what the `’bp` was for), then prints the title with the appropriate margins.

Why `’bp` and `’sp` instead of `.bp` and `.sp`? The answer is that `.sp` and `.bp`, like several other commands, cause a *break* to take place. That is, all the input text collected but not yet printed is flushed out as soon as possible, and the next input line is guaranteed to start a new line of output. If we had used `.sp` or `.bp` in the `.NP` macro, this would cause a break in the middle of the current output line when a new page is started. The effect would be to print the left-over part of that line at the top of the page, followed by the next input line on a new output line. This is *not* what we want. Using `’` instead of `.` for a command tells `troff` that no break is to take place—the output line currently being filled should *not* be forced out before the space or new page.

The list of commands that cause a break is short and natural:

```
.bp .br .ce .fi .nf .sp .in .ti
```

All others cause *no* break, regardless of whether you use a `.` or a `’`. If you really need a break, add a `.br` command at the appropriate place.

One other thing to beware of—if you’re changing fonts or point sizes a lot, you may find that if you cross a page boundary in an unexpected font or size, your titles come out in that size and font instead of what you intended. Furthermore, the length of a title is independent of the current line length, so titles will come out at the default length of 6.5 inches unless you change it, which is done with the `.lt` command.

There are several ways to fix the problems of point sizes and fonts in titles. For the simplest applications, we can change `.NP` to set the proper size and font for the title, then restore the previous values, like this:

```
.de NP
’bp
’sp 0.5i
.ft R      \” set title font to roman
.ps 10     \” and size to 10 point
.lt 6i     \” and length to 6 inches
.tl ‘left’center’right’
.ps       \” revert to previous size
.ft P     \” and to previous font
’sp 0.3i
..
```

This version of `.NP` does *not* work if the fields in the `.tl` command contain size or font changes. To cope with that requires `troff`’s “environment” mechanism, which we will discuss in Section 13.

To get a footer at the bottom of a page, you can modify `.NP` so it does some processing before the `’bp` command, or split the job into a footer macro invoked at the bottom margin and a header macro invoked at the top of the page. These variations are left as exercises.

Output page numbers are computed automatically as each page is produced (starting at 1), but no numbers are printed unless you ask for them explicitly. To get page numbers printed, include the character `%` in the `.tl` line at the position where you want the number to appear. For example

```
.tl “- % -”
```

centers the page number inside hyphens. You can set the page number at any time with either `.bp n`, which immediately starts a new page numbered `n`, or with `.pn n`, which sets the page number for the next page but doesn’t cause a skip to the new page. Again, `.bp +n` sets the page number to `n` more than its current value; `.bp` means `.bp +1`.

10. Number Registers and Arithmetic

`Troff` has a facility for doing arithmetic, and for defining and using variables with numeric values, called *number registers*. Number registers, like strings and macros, can be useful in setting up a document so it is easy to change later. And of course they serve for any sort of arithmetic computation.

Like strings, number registers have one or two character names. They are set by the `.nr` command, and are referenced anywhere by `\nx` (one character name) or `\n(xy)` (two character name).

There are quite a few pre-defined number registers maintained by `troff`, among them `%` for the current page number; `nl` for the current

vertical position on the page; `dy`, `mo` and `yr` for the current day, month and year; and `.s` and `.f` for the current size and font. (The font is a number from 1 to 4.) Any of these can be used in computations like any other register, but some, like `.s` and `.f`, cannot be changed with `.nr`.

As an example of the use of number registers, in the "ms" macro package [4], most significant parameters are defined in terms of the values of a handful of number registers. These include the point size for text, the vertical spacing, and the line and title lengths. To set the point size and vertical spacing for the following paragraphs, for example, a user may say

```
.nr PS 9
.nr VS 11
```

The paragraph macro `.PP` is defined (roughly) as follows:

```
.de PP
.ps \n(PS      \" reset size
.vs \n(VSPP    \" spacing
.ft R          \" font
.sp 0.5v      \" half a line
.ti +3m
..
```

This sets the font to Roman and the point size and line spacing to whatever values are stored in the number registers `PS` and `VS`.

Why are there two backslashes? This is the eternal problem of how to quote a quote. When `troff` originally reads the macro definition, it peels off one backslash to see what's coming next. To ensure that another is left in the definition when the macro is *used*, we have to put in two backslashes in the definition. If only one backslash is used, point size and vertical spacing will be frozen at the time the macro is defined, not when it is used.

Protecting by an extra layer of backslashes is only needed for `\n`, `*`, `\$` (which we haven't come to yet), and `\` itself. Things like `\s`, `\f`, `\h`, `\v`, and so on do not need an extra backslash, since they are converted by `troff` to an internal code immediately upon being seen.

Arithmetic expressions can appear anywhere that a number is expected. As a trivial example,

```
.nr PS \n(PS-2
```

decrements `PS` by 2. Expressions can use the arithmetic operators `+`, `-`, `*`, `/`, `%` (mod), the relational operators `>`, `>=`, `<`, `<=`, `=`, and `!=` (not equal), and parentheses.

Although the arithmetic we have done so far has been straightforward, more complicated things are somewhat tricky. First, number registers hold only integers. `Troff` arithmetic uses

truncating integer division, just like Fortran. Second, in the absence of parentheses, evaluation is done left-to-right without any operator precedence (including relational operators). Thus `7*-4+3/13` becomes `"-1"`.

Number registers can occur anywhere in an expression, and so can scale indicators like `p`, `i`, `m`, and so on (but no spaces). Although integer division causes truncation, each number and its scale indicator is converted to machine units (1/432 inch) before any arithmetic is done, so `1i/2u` evaluates to 0.5i correctly.

The scale indicator `u` often has to appear when you wouldn't expect it—in particular, when arithmetic is being done in a context that implies horizontal or vertical dimensions. For example,

```
.ll 7/2i
```

would seem obvious enough— $3\frac{1}{2}$ inches. Sorry. Remember that the default units for horizontal parameters like `.ll` are ems. That's really "7 ems/2 inches", and when translated into machine units, it becomes zero. How about

```
.ll 7i/2
```

Sorry, still no good—the "2" is "2 ems", so "7i/2" is small, even if not zero. You *must* use

```
.ll 7i/2u
```

So again, a safe rule is to attach a scale indicator to every number, even constants.

For arithmetic done within a `.nr` command, there is no implication of horizontal or vertical dimension, so the default units are "units", and `7i/2` and `7i/2u` mean the same thing. Thus

```
.nr ll 7i/2
.ll \n(llu
```

does just what you want, so long as you don't forget the `u` on the `.ll` command.

11. Macros with Arguments

The next step is to define macros that can change from one use to the next according to parameters supplied as arguments. To make this work, we need two things: first, when we define the macro, we have to indicate that some parts of it will be provided as arguments when the macro is called. Then when the macro is called we have to provide actual arguments to be plugged into the definition.

Let us illustrate by defining a macro `SM` that will print its argument two points smaller than the surrounding text. That is, the macro call

```
.SM TROFF
```

will produce TROFF.

The definition of **.SM** is

```
.de SM
\s-2\\$1\s+2
..
```

Within a macro definition, the symbol `\\$n` refers to the *n*th argument that the macro was called with. Thus `\\$1` is the string to be placed in a smaller point size when **.SM** is called.

As a slightly more complicated version, the following definition of **.SM** permits optional second and third arguments that will be printed in the normal size:

```
.de SM
\\$3\s-2\\$1\s+2\\$2
..
```

Arguments not provided when the macro is called are treated as empty, so

```
.SM TROFF ),
```

produces TROFF), while

```
.SM TROFF ). (
```

produces (TROFF). It is convenient to reverse the order of arguments because trailing punctuation is much more common than leading.

By the way, the number of arguments that a macro was called with is available in number register **.S**.

The following macro **.BD** is the one used to make the “bold roman” we have been using for **troff** command names in text. It combines horizontal motions, width computations, and argument rearrangement:

```
.de BD
&\\$3\\fI\\$1\\h'--\\w\\$1'u+2u\\$1\\fP\\$2
..
```

The `\\h` and `\\w` commands need no extra backslash, as we discussed above. The `\\&` is there in case the argument begins with a period.

Two backslashes are needed with the `\\$n` commands, though, to protect one of them when the macro is being defined. Perhaps a second example will make this clearer. Consider a macro called **.SH** which produces section headings rather like those in this paper, with the sections numbered automatically, and the title in bold in a smaller size. The use is

```
.SH "Section title ..."
```

(If the argument to a macro is to contain blanks, then it must be *surrounded* by double quotes, unlike a string, where only one leading quote is permitted.)

Here is the definition of the **.SH** macro:

```
.nr SH 0  \ " initialize section number
.de SH
.sp 0.3i
.ft B
.nr SH \\n(SH+1  \ " increment number
.ps \\n(PS-1  \ " decrease PS
\\n(SH. \\$1  \ " number. title
.ps \\n(PS  \ " restore PS
.sp 0.3i
.ft R
..
```

The section number is kept in number register **SH**, which is incremented each time just before it is used. (A number register may have the same name as a macro without conflict but a string may not.)

We used `\\n(SH` instead of `\\n(SH` and `\\n(PS` instead of `\\n(PS`. If we had used `\\n(SH`, we would get the value of the register at the time the macro was *defined*, not at the time it was *used*. If that's what you want, fine, but not here. Similarly, by using `\\n(PS`, we get the point size at the time the macro is called.

As an example that does not involve numbers, recall our **.NP** macro which had a

```
.tl 'left'center'right'
```

We could make these into parameters by using instead

```
.tl \\*(LT\\*(CT\\*(RT'
```

so the title comes from three strings called **LT**, **CT** and **RT**. If these are empty, then the title will be a blank line. Normally **CT** would be set to something like

```
.ds CT - % -
```

to give just the page number between hyphens, but a user could supply private definitions for any of the strings.

12. Conditionals

Suppose we want the **.SH** macro to leave two extra inches of space just before Section 1, but nowhere else. The cleanest way to do that is to test inside the **.SH** macro whether the section number is 1, and add some space if it is. The `.if` command provides the conditional test that we can add just before the heading line is output:

```
.if \\n(SH=1 .sp 2i  \ " Section 1 only
```

The condition after the `.if` can be any arithmetic or logical expression. If the condition is logically true, or arithmetically greater than zero, the rest of the line is treated as if it were text—here a command. If the condition is false,

or zero or negative, the rest of the line is skipped.

It is possible to do more than one command if a condition is true. Suppose several operations are to be done before Section 1. One possibility is to define a macro `.S1` and invoke it if we are about to do Section 1 (as determined by an `.if`).

```
.de S1
  ... processing for Section 1 ...
..
.de SH
  ...
.if \n(SH=1 .S1
  ...
..
```

An alternate way is to use the extended form of the `.if`, like this:

```
.if \n(SH=1 \{... processing
for Section 1 ...\}
```

The braces `\{` and `\}` must occur in the positions shown or you will get unexpected extra lines in your output. **Troff** also provides an “if-else” construction, which we will not go into here.

A condition can be negated by preceding it with `!`; we get the same effect as above (but less clearly) by using

```
.if !\n(SH>1 .S1
```

There are a handful of other conditions that can be tested with `.if`. For example, is the current page even or odd?

```
.if e .tl "even page title"
.if o .tl "odd page title"
```

gives facing pages different titles when used inside an appropriate new page macro.

Two other conditions are `t` and `n`, which tell you whether the formatter is **troff** or **nroff**.

```
.if t troff stuff ...
.if n nroff stuff ...
```

Finally, string comparisons may be made in an `.if`:

```
.if 'string1'string2' stuff
```

does “stuff” if *string1* is the same as *string2*. The character separating the strings can be anything reasonable that is not contained in either string. The strings themselves can reference strings with `*`, arguments with `\$`, and so on.

13. Environments

As we mentioned, there is a potential problem when going across a page boundary: parameters like size and font for a page title may well be

different from those in effect in the text when the page boundary occurs. **Troff** provides a very general way to deal with this and similar situations. There are three “environments”, each of which has independently settable versions of many of the parameters associated with processing, including size, font, line and title lengths, fill/no-fill mode, tab stops, and even partially collected lines. Thus the titling problem may be readily solved by processing the main text in one environment and titles in a separate one with its own suitable parameters.

The command `.ev n` shifts to environment `n`; `n` must be 0, 1 or 2. The command `.ev` with no argument returns to the previous environment. Environment names are maintained in a stack, so calls for different environments may be nested and unwound consistently.

Suppose we say that the main text is processed in environment 0, which is where **troff** begins by default. Then we can modify the new page macro `.NP` to process titles in environment 1 like this:

```
.de NP
.ev 1      \" shift to new environment
.lt 6i    \" set parameters here
.ft R
.ps 10
  ... any other processing ...
.ev      \" return to previous environment
..
```

One can also initialize an environment’s parameters outside the `.NP` macro, but the version shown keeps all the processing in one place and is thus easier to understand and change.

14. Diversions

There are numerous occasions in page layout when it is necessary to store some text for a period of time without actually printing it. Footnotes are the most obvious example: the text of the footnote usually appears in the input well before the place on the page where it is to be printed is reached. In fact, the place where it is output normally depends on how big it is, which implies that there must be a way to process the footnote at least enough to decide its size without printing it.

Troff provides a mechanism called a diversion for doing this processing. Any part of the output may be diverted into a macro instead of being printed, and then at some convenient time the macro may be put back into the input.

The command `.di xy` begins a diversion—all subsequent output is collected into the macro `xy` until the command `.di` with no arguments is encountered. This terminates the diversion.

The processed text is available at any time thereafter, simply by giving the command

```
.xy
```

The vertical size of the last finished diversion is contained in the built-in number register `dn`.

As a simple example, suppose we want to implement a “keep-release” operation, so that text between the commands `.KS` and `.KE` will not be split across a page boundary (as for a figure or table). Clearly, when a `.KS` is encountered, we have to begin diverting the output so we can find out how big it is. Then when a `.KE` is seen, we decide whether the diverted text will fit on the current page, and print it either there if it fits, or at the top of the next page if it doesn't. So:

```
.de KS  \" start keep
.br    \" start fresh line
.cv 1  \" collect in new environment
.fi    \" make it filled text
.di XX \" collect in XX
..
.de KE  \" end keep
.br    \" get last partial line
.di    \" end diversion
.if \\n(dn>=\\n(.t.bp \" .bp if doesn't fit
.nf    \" bring it back in no-fill
.XX    \" text
.ev    \" return to normal environment
..
```

Recall that number register `nl` is the current position on the output page. Since output was being diverted, this remains at its value when the diversion started; `dn` is the amount of text in the diversion; `.t` (another built-in register) is the distance to the next trap, which we assume is at the bottom margin of the page. If the diversion is large enough to go past the trap, the `.if` is satisfied, and a `.bp` is issued. In either case, the diverted output is then brought back with `.XX`. It is essential to bring it back in no-fill mode so `troff` will do no further processing on it.

This is not the most general keep-release, nor is it robust in the face of all conceivable inputs, but it would require more space than we have here to write it in full generality. This section is not intended to teach everything about diversions, but to sketch out enough that you can read existing macro packages with some comprehension.

Acknowledgements

I am deeply indebted to J. F. Ossanna, the author of `troff`, for his repeated patient explanations of fine points, and for his continuing willingness to adapt `troff` to make other uses easier.

I am also grateful to Jim Blinn, Ted Dolotta, Doug McIlroy, Mike Lesk, and Joel Sturman for helpful comments on this paper.

References

- [1] J. F. Ossanna. *NROFF/TROFF User's Manual*, Bell Laboratories.
- [2] B. W. Kernighan and L. L. Cherry. *A System for Typesetting Mathematics—User's Guide (Second Edition)*, Bell Laboratories.
- [3] M. E. Lesk. *TBL—A Program to Format Tables*, Bell Laboratories.
- [4] M. E. Lesk. *Typing Documents on UNIX*, Bell Laboratories.
- [5] J. R. Mashey and D. W. Smith. *MM—Memorandum Macros*, Bell Laboratories.

Appendix: Typesetter Character Set

The following characters exist in roman, italic, and bold; they are entered as themselves:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
 a b c d e f g h i j k l m n o p q r s t u v w x y z
 1 2 3 4 5 6 7 8 9 0
 ! \$ % & () = | [] + * ; : , . / ?

The following characters also exist in roman, italic, and bold; to get the one on the left, type the one-, two-, or four-character name on the right (in what follows, the symbol $\acute{}$ is the acute accent or apostrophe on most keyboards, while $\grave{}$ is the other—or grave—accent; $-$ is the minus sign on the keyboard):

ff	\backslash (ff	fi	\backslash (fi	fl	\backslash (fl	ffi	\backslash (Fi
ffi	\backslash (Fi	-	\backslash (ru	¼	\backslash (14	¾	\backslash (34
½	\backslash (12	-	\backslash (em	©	\backslash (co	•	\backslash (de
†	\backslash (dg	-	\backslash (-	¢	\backslash (ct	•	\backslash (rg
•	\backslash (bu	-	-
ˆ	\backslash (fm	□	\backslash (sq (in bold, \backslash (sq prints as ■)				

The following characters appear only on the special font:

\backslash	\backslash (e	-	\backslash (rn	ˆ	\backslash (ˆ	\backslash (
+	\backslash (pl	-	\backslash (mi	×	\backslash (mu	÷	\backslash (di
=	\backslash (eq	=	\backslash (==	≥	\backslash (>=	≤	\backslash (<=
≠	\backslash (!=	±	\backslash (+-	∩	\backslash (no	/	\backslash (sl
~	\backslash (ap	≈	\backslash (~=	∞	\backslash (pt	∇	\backslash (gr
→	\backslash (->	←	\backslash (<-	↑	\backslash (ua	↓	\backslash (da
f	\backslash (is	∂	\backslash (pd	∞	\backslash (if	√	\backslash (sr
⊂	\backslash (sb	⊃	\backslash (sp	∪	\backslash (cu	∩	\backslash (ca
⊆	\backslash (ib	⊇	\backslash (ip	∈	\backslash (mo	∅	\backslash (es
§	\backslash (sc	‡	\backslash (dd	⊗	\backslash (lh	⊗	\backslash (rh
{	\backslash (lt	}	\backslash (rt	⌈	\backslash (lc	⌋	\backslash (rc
[\backslash (lb]	\backslash (rb	⌊	\backslash (lf	⌋	\backslash (rf
}	\backslash (lk	}	\backslash (rk		\backslash (bv	*	\backslash (**
	\backslash (br		\backslash (or	○	\backslash (ci	⊙	\backslash (bs
s	\backslash (ts						

The special-font (“math”) characters named \backslash (pl, \backslash (mi, \backslash (**, \backslash (sl, and \backslash (eq (i.e., +, -, *, /, and =) are *not* the same as the “current-font” characters named +, -, *, /, and = (i.e., +, -, *, /, and =).

The following characters are also found only on the special font; they are entered as themselves (but remember to escape with a \backslash the # and @ if these are your “erase” and “kill” characters):

@ " { } < > ~ ^ _

The following pairs of *input names* are synonyms for each other:

\backslash ˆ \backslash (ga \backslash ˆ \backslash (aa - \backslash (ul - \backslash (hy

All Greek letters are also on the special font: all the lower-case letters (including the terminal sigma s from the list above) and some upper-case letters (Γ, Δ, Θ, Λ, Ξ, Π, Σ, Τ, Φ, Ψ, and Ω); the remaining upper-case Greek letters are “faked” by using the corresponding upper-case Roman letters; precede the Roman letter by \backslash (* to get the corresponding Greek letter (for example, \backslash (*a prints as α):

a b g d e z y h i k l m n c o p r s t u f x q w
 α β γ δ ε ζ η θ ι κ λ μ ν ξ ο π ρ σ τ υ φ χ ψ ω
 A B G D E Z Y H I K L M N C O P R S T U F X Q W
 A B Γ Δ E Z H Θ I K Λ M N Ξ O Π Ρ Σ Τ Τ Φ Ψ Ω

NROFF/TROFF User's Manual

Joseph F. Ossanna

Bell Laboratories
Murray Hill, New Jersey 07974

Introduction

NROFF and TROFF are text processors under the UNIX† Time-Sharing System [1] that format text for typewriter-like terminals and for a phototypesetter, respectively. They accept lines of text interspersed with lines of format control information and format the text into a printable, paginated document having a user-designed style. NROFF and TROFF offer unusual freedom in document styling, including: arbitrary style headers and footers; arbitrary style footnotes; multiple automatic sequence numbering for paragraphs, sections, etc; multiple column output; dynamic font and point-size control; arbitrary horizontal and vertical local motions at any point; and a family of automatic overstriking, bracket construction, and line drawing functions.

NROFF and TROFF are highly compatible with each other and it is almost always possible to prepare input acceptable to both. Conditional input is provided that enables the user to embed input expressly destined for either program. NROFF can prepare output directly for a variety of terminal types and is capable of utilizing the full resolution of each terminal.

Usage

The general form of invoking NROFF (or TROFF) at UNIX command level is:

```
nroff options files          or
troff options files
```

where *options* represents any of a number of option arguments and *files* represents the list of files containing the document to be formatted. An argument consisting of a single minus (–) is taken to be a file name corresponding to the standard input. If no file names are given input is taken from the standard input. The options, which may appear in any order so long as they appear before the files, are:

<i>Option</i>	<i>Effect</i>
– <i>olist</i>	Print only pages whose page numbers appear in <i>list</i> , which consists of comma-separated numbers and number ranges. A number range has the form $N-M$ and means pages N through M ; a initial $-N$ means from the beginning to page N ; and a final $N-$ means from N to the end.
– <i>nN</i>	Number first generated page N .
– <i>sN</i>	Stop every N pages. NROFF will halt prior to every N pages (default $N=1$) to allow paper loading or changing, and will resume upon receipt of a new-line. TROFF will stop the phototypesetter every N pages, produce a trailer to allow changing cassettes, and will resume after the phototypesetter START button is pressed. See §1.2 of the Addendum for additional details.
– <i>mname</i>	Prepends the macro file <code>/usr/lib/tmac.name</code> to the input <i>files</i> .
– <i>raN</i>	The number register whose (one-character) name is <i>a</i> is set to N .
– <i>i</i>	Read standard input after the input files are exhausted.
– <i>q</i>	Invoke the simultaneous input-output mode of the <code>rd</code> request.

† UNIX is a trademark of Bell Laboratories.

NROFF Only

- Tname** Specifies the name of the output terminal type. Currently defined names are **37** for the (default) *TELETYPE*® Model 37, **tn300** for the GE TermiNet 300 (or any terminal without half-line capabilities), **300** for the DASI 300, **300s** for the DASI 300s, and **450** for the DASI 450; **300-12**, **300s-12**, and **450-12**, respectively, are used to print in 12-pitch (12 characters per inch) on the three DASI terminals.
- e** Produce equally-spaced words in adjusted lines, using full terminal resolution.

TROFF Only

- t** Direct output to the standard output instead of the phototypesetter.
- f** Refrain from feeding out paper and stopping phototypesetter at the end of the run.
- w** Wait until phototypesetter is available, if currently busy.
- b** TROFF will report whether the phototypesetter is busy or available. No text processing is done.
- a** Send a printable (ASCII) approximation of the results to the standard output.
- pN** Print all characters in point size *N* while retaining all prescribed spacings and motions, to reduce phototypesetter elapsed time.
- g** Prepare output for the Murray Hill Computation Center phototypesetter and direct it to the standard output.

☞ See §1 of the Addendum for additional and modified command-line options.

Each option is invoked as a separate argument; for example:

```
nroff -o4,8-10 -T300-12 -mabc file1 file2
```

requests formatting of pages 4, 8, 9, and 10 of the document contained in files named **file1** and **file2**, specifies the output terminal as a DASI 300 in 12-pitch, and invokes the macro package **abc**.

Various pre- and postprocessors are available for use with NROFF and TROFF. These include the equation preprocessors NEQN and EQN [2] (for NROFF and TROFF respectively), the table-construction preprocessor TBL [3], and the constant-width preprocessor CW [1]. A reverse-line postprocessor COL [1] is available for multiple-column NROFF output on terminals without reverse-line ability; COL expects the *TELETYPE* Model 37 escape sequences that NROFF produces by default. 4014 [1] is a *TELETYPE* Model 37-simulator postprocessor for printing NROFF output on a Tektronix 4014. TC [1] is phototypesetter-simulator postprocessor for TROFF that produces an approximation of phototypesetter output on a Tektronix 4014. For example, in:

```
tbl files | eqn | troff -t options | tc
```

the first | indicates the piping of TBL's output to EQN's input; the second the piping of EQN's output to TROFF's input; and the third indicates the piping of TROFF's output to TC. GCAT [1] can be used to send TROFF (-g) output to the Murray Hill Computation Center.

The remainder of this manual consists of: a Summary and Index; a Reference Manual keyed to the Index; a set of Tutorial Examples (see also [4]); and an Addendum.

Joseph F. Ossanna

References

- [1] T. A. Dolotta, S. B. Olsson, and A. G. Petrucci (eds.). *UNIX User's Manual—Release 3.0*, June 1980, Bell Laboratories.
- [2] B. W. Kernighan and L. L. Cherry. *Typesetting Mathematics—User's Guide (Second Edition)*, Bell Laboratories.
- [3] M. E. Lesk. *TBL—A Program to Format Tables*, Bell Laboratories.
- [4] B. W. Kernighan. *A TROFF Tutorial*, Bell Laboratories.

SUMMARY AND INDEX

<i>Request Form</i>	<i>Initial Value*</i>	<i>If No Argument</i>	<i>Notes#</i>	<i>Explanation</i>
1. General Explanation				
2. Font and Character Size Control				
.ps ±N	10 point	previous	E	Point size; also \s±N.†
.ss N	12/36 em	ignored	E	Space-character size set to N/36 em.†
.cs FNM	off	-	P	Constant character space (width) mode (font F).†
.bd FN	off	-	P	Embolden font F by N-1 units.†
.bd SFN	off	-	P	Embolden Special Font when current font is F.†
.ft F	Roman	previous	E	Change to font F = x, xx, or 1-4. Also \fx, \f(xx, \fN.
.fp NF	R,I,B,S	ignored	-	Font named F mounted on physical position 1≤N≤4.
3. Page Control				
.pl ±N	11 in	11 in	v	Page length.
.bp ±N	N=1	-	B‡,v	Eject current page; next page number N.
.pn ±N	N=1	ignored	-	Next page number N.
.po ±N	0; 26/27 in	previous	v	Page offset.
.ne N	-	N=1V	D,v	Need N vertical space (V = vertical spacing).
.mk R	none	internal	D	Mark current vertical place in register R.
.rt ±N	none	internal	D,v	Return (<i>upward only</i>) to marked vertical place.
4. Text Filling, Adjusting, and Centering				
.br	-	-	B	Break.
.fi	fill	-	B,E	Fill output lines.
.nf	fill	-	B,E	No filling or adjusting of output lines.
.ad c	adj,both	adjust	E	Adjust output lines with mode c.
.na	adjust	-	E	No output line adjusting.
.ce N	off	N=1	B,E	Center following N input text lines.
5. Vertical Spacing				
.vs N	1/6in;12pts	previous	E,p	Vertical base line spacing (V).
.ls N	N=1	previous	E	Output N-1 Vs after each text output line.
.sp N	-	N=1V	B,v	Space vertical distance N in either direction.
.sv N	-	N=1V	v	Save vertical distance N.
.os	-	-	-	Output saved vertical distance.
.ns	space	-	D	Turn no-space mode on.
.rs	-	-	D	Restore spacing; turn no-space mode off.
6. Line Length and Indenting				
.ll ±N	6.5 in	previous	E,m	Line length.
.in ±N	N=0	previous	B,E,m	Indent.
.ti ±N	-	ignored	B,E,m	Temporary indent.
7. Macros, Strings, Diversion, and Position Traps				
.de xx yy	-	.yy=..	-	Define or redefine macro xx; end at call of yy.
.am xx yy	-	.yy=..	-	Append to a macro.
.ds xx string	-	ignored	-	Define a string xx containing string.
.as xx string	-	ignored	-	Append string to string xx.

* Values separated by ";" are for NROFF and TROFF respectively.

Notes are explained at the end of this Summary and Index.

† No effect in NROFF.

‡ The use of "" as control character (instead of ".") suppresses the break function.

<i>Request Form</i>	<i>Initial Value</i>	<i>If No Argument</i>	<i>Notes</i>	<i>Explanation</i>
.rm xx	-	ignored	-	Remove request, macro, or string.
.rn xx yy	-	ignored	-	Rename request, macro, or string xx to yy.
.di xx	-	end	D	Divert output to macro xx.
.da xx	-	end	D	Divert and append to xx.
.wh N xx	-	-	v	Set location trap; negative is w.r.t. page bottom.
.ch xx N	-	-	v	Change trap location.
.dt N xx	-	off	D,v	Set a diversion trap.
.it N xx	-	off	E	Set an input-line count trap.
.em xx	none	none	-	End macro is xx.

8. Number Registers

.nr R ±N M	-	-	u	Define and set number register R; auto-increment by M.
.af R c	arabic	-	-	Assign format to register R (c=1, i, I, a, A).
.rr R	-	-	-	Remove register R.

9. Tabs, Leaders, and Fields

.ta Nt ...	0.8; 0.5in	none	E,m	Tab settings; <i>left</i> type, unless t=R(right), C(centered).
.tc c	none	none	E	Tab repetition character.
.lc c	.	none	E	Leader repetition character.
.fc a b	off	off	-	Set field delimiter a and pad character b.

10. Input and Output Conventions and Character Translations

.ec c	\	\	-	Set escape character.
.eo	on	-	-	Turn off escape character mechanism.
.lg N	-; on	on	-	Ligature mode on if N>0.
.ul N	off	N=1	E	Underline (italicize in TROFF) N input lines.
.cu N	off	N=1	E	Continuous underline in NROFF; like ul in TROFF.
.uf F	Italic	Italic	-	Underline font set to F (to be switched to by ul).
.cc c	:	:	E	Set control character to c.
.c2 c	:	:	E	Set no-break control character to c.
.tr abcd...	none	-	O	Translate a to b, etc. on output.

11. Local Horizontal and Vertical Motions, and the Width Function

12. Overstrike, Bracket, Line-drawing, and Zero-width Functions

13. Hyphenation.

.nh	no hyphen.	-	E	No hyphenation.
.hy N	no hyphen.	hyphenate	E	Hyphenate; N = mode.
.hc c	\%	\%	E	Hyphenation indicator character c.
.hw word1 ...		ignored	-	Exception words.

14. Three Part Titles.

.tl left center right		-	-	Three part title.
.pc c	%	off	-	Page number character.
.lt ±N	6.5in	previous	E,m	Length of title.

15. Output Line Numbering.

.nm ±N M S I		off	E	Number mode on or off, set parameters.
.nn N		N=1	E	Do not number next N lines.

<i>Request Form</i>	<i>Initial Value</i>	<i>If No Argument</i>	<i>Notes</i>	<i>Explanation</i>
---------------------	----------------------	-----------------------	--------------	--------------------

16. Conditional Acceptance of Input

<i>.if c anything</i>	-	-	-	If condition <i>c</i> true, accept <i>anything</i> as input, for multi-line use $\backslash\{anything\}$.
<i>.if !c anything</i>	-	-	-	If condition <i>c</i> false, accept <i>anything</i> .
<i>.if N anything</i>	-	-	u	If expression $N > 0$, accept <i>anything</i> .
<i>.if !N anything</i>	-	-	u	If expression $N \leq 0$, accept <i>anything</i> .
<i>.if string1 string2 anything</i>	-	-	-	If <i>string1</i> identical to <i>string2</i> , accept <i>anything</i> .
<i>.if ! string1 string2 anything</i>	-	-	-	If <i>string1</i> not identical to <i>string2</i> , accept <i>anything</i> .
<i>.ie c anything</i>	-	-	u	If portion of if-else; all above forms (like if).
<i>.el anything</i>	-	-	-	Else portion of if-else.

17. Environment Switching.

<i>.ev N</i>	$N=0$	previous	-	Environment switched (<i>push down</i>).
--------------	-------	----------	---	--

18. Insertions from the Standard Input

<i>.rd prompt</i>	-	<i>prompt=BEL-</i>	-	Read insertion.
<i>.ex</i>	-	-	-	Exit from NROFF/TROFF.

19. Input/Output File Switching

<i>.so filename</i>	-	-	-	Switch source file (<i>push down</i>).
<i>.nx filename</i>	-	end-of-file	-	Next file.
<i>.pi program</i>	-	-	-	Pipe output to <i>program</i> (NROFF only).

20. Miscellaneous

<i>.mc c N</i>	-	off	E, m	Set margin character <i>c</i> and separation <i>N</i> .
<i>.tm string</i>	-	new-line	-	Print <i>string</i> on terminal (UNIX standard message output).
<i>.ig yy</i>	-	<i>.yy=..</i>	-	Ignore till call of <i>yy</i> .
<i>.pm t</i>	-	all	-	Print macro names and sizes; if <i>t</i> present, print only total of sizes.
<i>.fl</i>	-	-	B	Flush output buffer.

21. Output and Error Messages

Notes:

- B** Request normally causes a break.
- D** Mode or relevant parameters associated with current diversion level.
- E** Relevant parameters are a part of the current environment.
- O** Must stay in effect until logical output.
- P** Mode must be still or again in effect at the time of physical output.
- v, p, m, u** Default scale indicator; if not specified, scale indicators are *ignored*.

Alphabetical Request and Section Number Cross Reference

ab *	c2 10	de 7	ev 17	hw 13	lg 10	nf 4	pc 14	rm 7	sv 5	ul 10
ad 4	cc 10	di 7	ex 18	hy 13	ll 6	nh 13	pi 19	rn 7	ta 9	vs 5
af 8	ce 4	ds 7	fc 9	ie 16	ls 5	nm 15	pl 3	rr 8	tc 9	wh 7
am 7	ch 7	dt 7	fi 4	if 16	lt 14	nn 15	pm 20	rs 5	ti 6	! *
as 7	co *	ec 10	fl 20	ig 20	mc 20	nr 8	pn 3	rt 3	tl 14	
bd 2	cs 2	el 16	fp 2	in 6	mk 3	ns 5	po 3	so 19	tm 20	
bp 3	cu 10	em 7	ft 2	it 7	na 4	nx 19	ps 2	sp 5	tr 10	
br 4	da 7	eo 10	hc 13	lc 9	ne 3	os 5	rd 18	ss 2	uf 10	

* This request is described in §2.1 of the Addendum.

Escape Sequences for Characters, Indicators, and Functions

<i>Section Reference</i>	<i>Escape Sequence</i>	<i>Meaning</i>
10.1	\\	\ (to prevent or delay the interpretation of \)
10.1	\e	Printable version of the <i>current</i> escape character.
2.1	\`	` (acute accent); equivalent to \(\aa
2.1	\~	~ (grave accent); equivalent to \(\ga
2.1	\-	- Minus sign in the <i>current</i> font
7	\.	Period (dot) (see <i>de</i>)
11.1	\(space)	Unpaddable space-size space character
11.1	\0	Digit width space
11.1	\	1/6 em narrow space character (zero width in NROFF)
11.1	\^	1/12 em half-narrow space character (zero width in NROFF)
4.1	\&	Non-printing, zero width character
10.6	\!	Transparent line indicator
10.7	*	Beginning of comment
7.3	\\$N	Interpolate argument $1 \leq N \leq 9$
13	\%	Default optional hyphenation character
2.1	\(xx	Character named <i>xx</i>
7.1	*x, *(xx	Interpolate string <i>x</i> or <i>xx</i>
9.1	\a	Non-interpreted leader character
12.3	\b abc...	Bracket building function
4.2	\c	Interrupt text processing
11.1	\d	Forward (down) 1/2 em vertical motion (1/2 line in NROFF)
2.2	\fx, \f(xx, \fN	Change to font named <i>x</i> or <i>xx</i> , or position <i>N</i>
11.1	\h N	Local horizontal motion; move right <i>N</i> (<i>negative left</i>)
11.3	\kx	Mark horizontal <i>input</i> place in register <i>x</i>
12.4	\l Nc	Horizontal line drawing function (optionally with <i>c</i>)
12.4	\L Nc	Vertical line drawing function (optionally with <i>c</i>)
8	\nx, \n(xx	Interpolate number register <i>x</i> or <i>xx</i>
12.1	\o abc...	Overstrike characters <i>a</i> , <i>b</i> , <i>c</i> , ...
4.1	\p	Break and spread output line
11.1	\r	Reverse 1 em vertical motion (reverse line in NROFF)
2.3	\sN, \s±N	Point-size change function
9.1	\t	Non-interpreted horizontal tab
11.1	\u	Reverse (up) 1/2 em vertical motion (1/2 line in NROFF)
11.1	\v N	Local vertical motion; move down <i>N</i> (<i>negative up</i>)
11.2	\w string	Interpolate width of <i>string</i>
5.2	\x N	Extra line-space function (<i>negative before, positive after</i>)
12.2	\zc	Print <i>c</i> with zero width (without spacing)
16	\{	Begin conditional input
16	\}	End conditional input
10.7	\(new-line)	Concealed (ignored) new-line
-	\X	<i>X</i> , any character <i>not</i> listed above

The escape sequences \\, \., *, \\$, \a, \n, \t, and \(\new-line) are interpreted in *copy mode* (§7.2).

See §3 of the Addendum for additional escape sequences.

Predefined General Number Registers

<i>Section Reference</i>	<i>Register Name</i>	<i>Description</i>
3	%	Current page number.
11.2	ct	Character type (set by <i>width</i> function).
7.4	dl	Width (maximum) of last completed diversion.
7.4	dn	Height (vertical size) of last completed diversion.
-	dw	Current day of the week (1-7).
-	dy	Current day of the month (1-31).
11.3	hp	Current horizontal place on <i>input</i> line.
15	ln	Output line number.
-	mo	Current month (1-12).
4.1	nl	Vertical position of last printed text base-line.
11.2	sb	Depth of string below base line (generated by <i>width</i> function).
11.2	st	Height of string above base line (generated by <i>width</i> function).
-	yr	Last two digits of current year.

Predefined Read-Only Number Registers

<i>Section Reference</i>	<i>Register Name</i>	<i>Description</i>
7.3	.\$	Number of arguments available at the current macro level.
-	.A	Set to 1 in TROFF, if <i>-a</i> option used; always 1 in NROFF.
11.1	.H	Available horizontal resolution in basic units.
-	.T	Set to 1 in NROFF, if <i>-T</i> option used; always 0 in TROFF.
11.1	.V	Available vertical resolution in basic units.
5.2	.a	Post-line extra line-space most recently utilized using <i>\x'N'</i> .
-	.c	Number of <i>lines</i> read from current input file.
7.4	.d	Current vertical place in current diversion; equal to <i>nl</i> , if no diversion.
2.2	.f	Current font as physical quadrant (1-4).
4	.h	Text base-line high-water mark on current page or diversion.
6	.i	Current indent.
6	.l	Current line length.
4	.n	Length of text portion on previous output line.
3	.o	Current page offset.
3	.p	Current page length.
2.3	.s	Current point size.
7.5	.t	Distance to the next trap.
4.1	.u	Equal to 1 in fill mode and 0 in no-fill mode.
5.1	.v	Current vertical line spacing.
11.2	.w	Width of previous character.
-	.x	Reserved version-dependent register.
-	.y	Reserved version-dependent register.
7.4	.z	Name of current diversion.

☞ See §4 of the Addendum for additional predefined number registers.

REFERENCE MANUAL

1. General Explanation

1.1. Form of input. Input consists of *text lines*, which are destined to be printed, interspersed with *control lines*, which set parameters or otherwise control subsequent processing. Control lines begin with a *control character*—normally . (period) or ` (acute accent)—followed by a one or two character name that specifies a basic *request* or the substitution of a user-defined *macro* in place of the control line. The control character ` suppresses the *break* function—the forced output of a partially filled line—caused by certain requests. The control character may be separated from the request/macro name by white space (spaces and/or tabs) for esthetic reasons. Names must be followed by either space or new-line. Control lines with unrecognized names are ignored.

Various special functions may be introduced anywhere in the input by means of an *escape* character, normally \. For example, the function \nR causes the interpolation of the contents of the *number register R* in place of the function; here *R* is either a single character name as in \nx, or left-parenthesis-introduced, two-character name as in \n(xx).

1.2. Formatter and device resolution. For historical reasons, TROFF internally uses 432 units/inch, and has a horizontal resolution of 1/432 inch and a vertical resolution of 1/144 inch. NROFF internally uses 240 units/inch, corresponding to the least common multiple of the horizontal and vertical resolutions of various current typewriter-like output devices. TROFF rounds horizontal/vertical numerical parameter input to its internal horizontal/vertical resolution. NROFF similarly rounds numerical input to the actual resolution of the output device indicated by the -T option (default *TELETYPE* Model 37).

1.3. Numerical parameter input. Both NROFF and TROFF accept numerical input with the appended scale indicators shown in the following table, where *S* is the current type size in points, *V* is the current vertical line spacing in basic units, and *C* is a *nominal character width* in basic units.

Scale Indicator	Meaning	Number of basic units	
		TROFF	NROFF
i	Inch	432	240
c	Centimeter	$432 \times 50 / 127$	$240 \times 50 / 127$
P	Pica = 1/6 inch	72	240/6
m	Em = <i>S</i> points	$6 \times S$	<i>C</i>
n	En = Em/2	$3 \times S$	<i>C, same as Em</i>
p	Point = 1/72 inch	6	240/72
u	Basic unit	1	1
v	Vertical line space	<i>V</i>	<i>V</i>
none	Default, see below		

In NROFF, *both* the em and the en are taken to be equal to the *C*, which is output-device dependent; common values are 1/10 and 1/12 inch. Actual character widths in NROFF need not be all the same and constructed characters such as -> (→) are often extra wide. The default scaling is ems for the horizontally-oriented requests and functions ll, in, ti, ta, lt, po, mc, \h, and \l; Vs for the vertically-oriented requests and functions pl, wh, ch, dt, sp, sv, ne, rt, \v, \x, and \L; p for the vs request; and u for the requests nr, if, and ie. All other requests ignore any scale indicators. When a number register containing an already appropriately scaled number is interpolated to provide numerical input, the unit scale indicator u may need to be appended to prevent an additional inappropriate default scaling. The number, *N*, may be specified in decimal-fraction form but the parameter finally stored is rounded to an integer number of basic units.

The *absolute position* indicator | may be prepended to a number *N* to generate the distance to the vertical or horizontal place *N*. For vertically-oriented requests and functions, |*N* becomes the distance in basic units from the current vertical place on the page or in a *diversion* (§7.4) to the the vertical place *N*. For *all* other requests and functions, |*N* becomes the distance from the current horizontal place on the *input* line to the horizontal place *N*. For example,

.sp |3.2c

will space *in the required direction* to 3.2 centimeters from the top of the page.

1.4. Numerical expressions. Wherever numerical input is expected an expression involving parentheses, the arithmetic operators +, -, /, *, % (mod), and the logical operators <, >, <=, >=, = (or ==), & (and), : (or) may be used. Except where controlled by parentheses, evaluation of expressions is left-to-right; there is no operator precedence. In the case of certain requests, an initial + or - is stripped and interpreted as an increment or decrement indicator respectively. In the presence of default scaling, the desired scale indicator must be attached to *every* number in an expression for which the desired and default scaling differ. For example, if the number register *x* contains 2 and the current point size is 10, then

.ll (4.25i+\nxP+3)/2u

will set the line length to 1/2 the sum of 4.25 inches + 2 picas + 30 points.

1.5. Notation. Numerical parameters are indicated in this manual in two ways. ±*N* means that the argument may take the forms *N*, +*N*, or -*N* and that the corresponding effect is to set the affected parameter to *N*, to increment it by *N*, or to decrement it by *N* respectively. Plain *N* means that an initial algebraic sign is *not* an increment indicator, but merely the sign of *N*. Generally, unreasonable numerical input is either ignored or truncated to a reasonable value. For example, most requests expect to set parameters to non-negative values; exceptions are **sp**, **wh**, **ch**, **nr**, and **if**. The requests **ps**, **ft**, **po**, **vs**, **ls**, **ll**, **in**, and **lt** restore the *previous* parameter value in the *absence* of an argument.

Single character arguments are indicated by single lower case letters and one/two character arguments are indicated by a pair of lower case letters. Character string arguments are indicated by multi-character mnemonics.

2. Font and Character Size Control

2.1. Character set. The TROFF character set consists of the so-called Commercial II character set plus a Special Mathematical Font character set—each having 102 characters. These character sets are shown in the attached Table I. All ASCII *graphic* characters are included, with some on the Special Font. With three exceptions, these ASCII characters are input as themselves, and non-ASCII characters are input in the form \(\i>xx where *xx* is a two-character name given in the attached Table II. The three ASCII exceptions are mapped as follows:

ASCII Input		Printed by TROFF	
Character	Name	Character	Name
·	acute accent	'	close quote
˘	grave accent	`	open quote
-	minus	-	hyphen

The characters ·, ˘, and - may be input by \·, \˘, and \- respectively or by their names (Table II). The ASCII characters @, #, \$, %, &, <, >, \, {, }, ~, ^, and _ exist only on the Special Font and are printed as a 1-em space if that font is not mounted. (ASCII *control* characters are discussed in §10.1.)

NROFF understands the entire TROFF character set, but can in general print only ASCII characters, additional characters as may be available on the output device, such characters as may be able to be constructed by overstriking or other combination, and those that can reasonably be mapped into other printable characters. The exact behavior is determined by a driving table prepared for each device. The characters ·, ˘, and _ print as themselves.

2.2. *Fonts.* The default mounted fonts are Times Roman (**R**), Times Italic (**I**), Times Bold (**B**), and the Special Mathematical Font (**S**) on physical typesetter positions 1, 2, 3, and 4 respectively. These fonts are used in this document. The *current* font, initially Roman, may be changed (among the mounted fonts) by use of the **ft** request, or by imbedding at any desired point either $\backslash fx$, $\backslash f(xx)$, or $\backslash fN$ where x and xx are the name of a mounted font and N is a numerical font position. It is *not* necessary to change to the Special Font; characters on that font are automatically handled. A request for a named but not-mounted font is *ignored*. TROFF can be informed that any particular font is mounted by use of the **fp** request. The list of known fonts is installation dependent. In the subsequent discussion of font-related requests, F represents either a one/two-character font name or the numerical font position, 1-4. The current font is available (as numerical position) in the read-only number register **.f**.

NROFF understands font control and normally underlines Italic characters (see §10.5).

2.3. *Character size.* Available character point sizes are 6, 7, 8, 9, 10, 11, 12, 14, 16, 18, 20, 22, 24, 28, and 36. This is a range of 1/12 inch to 1/2 inch. The **ps** request is used to change or restore the point size. Alternatively, the point size may be changed between any two characters by imbedding a $\backslash sN$ at the desired point to set the size to N , or a $\backslash s \pm N$ ($1 \leq N \leq 9$) to increment/decrement the size by N ; $\backslash s0$ restores the *previous* size. Requested point size values that are between two valid sizes yield the larger of the two. The current size is available in the **.s** register. NROFF ignores type size control.

<i>Request Form</i>	<i>Initial Value</i>	<i>If No Argument</i>	<i>Notes*</i>	<i>Explanation</i>
.ps $\pm N$	10 point	previous	E	Point size set to $\pm N$. Alternatively, imbed $\backslash sN$ or $\backslash s \pm N$. Any positive size value may be requested; if invalid, the next larger valid size will result, with a maximum of 36. A paired sequence $+N, -N$ will work because the previous requested value is also remembered. Ignored in NROFF.
.ss N	12/36 em	ignored	E	Space-character size is set to $N/36$ ems. This size is the minimum word spacing in adjusted text. Ignored in NROFF.
.cs FNM	off	-	P	Constant character space (width) mode is set on for font F (if mounted); the width of every character will be taken to be $N/36$ ems. If M is absent, the em is that of the character's point size; if M is given, the em is M -points. All affected characters are centered in this space, including those with an actual width larger than this space. Special Font characters occurring while the current font is F are also so treated. If N is absent, the mode is turned off. The mode must be still or again in effect when the characters are physically printed. Ignored in NROFF.
.bd FN	off	-	P	The characters in font F will be artificially emboldened by printing each one twice, separated by $N-1$ basic units. A reasonable value for N is 3 when the character size is in the vicinity of 10 points. If N is missing the embolden mode is turned off. The column heads above were printed with .bd I 3 . The mode must be still or again in effect when the characters are physically printed. See §2.2 of the Addendum for the effect of bd in NROFF.
.bd S FN	off	-	P	The characters in the Special Font will be emboldened whenever the current font is F . This manual was printed

* Notes are explained at the end of the Summary and Index above.

with **.bdSB3**. The mode must be still or again in effect when the characters are physically printed.

.ft <i>F</i>	Roman	previous	E	Font changed to <i>F</i> . Alternatively, imbed \fF . The font name P is reserved to mean the previous font.
.fp <i>N F</i>	R,I,B,S	ignored	-	Font position. This is a statement that a font named <i>F</i> is mounted on position <i>N</i> (1-4). It is a fatal error if <i>F</i> is not known. The phototypesetter has four fonts physically mounted. Each font consists of a film strip which can be mounted on a numbered quadrant of a wheel. The default mounting sequence assumed by TROFF is R, I, B, and S on positions 1, 2, 3 and 4.

3. Page control

Top and bottom margins are *not* automatically provided; it is conventional to define two *macros* and to set *traps* for them at vertical positions 0 (top) and $-N$ (*N* from the bottom). See §7 and Tutorial Examples §T2. A pseudo-page transition onto the *first* page occurs either when the first *break* occurs or when the first *non-diverted* text processing occurs. Arrangements for a trap to occur at the top of the first page must be completed before this transition. In the following, references to the *current diversion* (§7.4) mean that the mechanism being described works during both ordinary and diverted output (the former considered as the top diversion level).

The usable page width on the phototypesetter is about 7.54 inches, beginning about 1/27 inch from the left edge of the 8 inch wide, continuous roll paper. The physical limitations on NROFF output are output-device dependent.

Request Form	Initial Value	If No Argument	Notes	Explanation
.pl $\pm N$	11 in	11 in	v	Page length set to $\pm N$. The internal limitation is about 75 inches in TROFF and about 136 inches in NROFF. The current page length is available in the .p register.
.bp $\pm N$	$N=1$	-	B*,v	Begin page. The current page is ejected and a new page is begun. If $\pm N$ is given, the new page number will be $\pm N$. Also see request ns .
.pn $\pm N$	$N=1$	ignored	-	Page number. The next page (when it occurs) will have the page number $\pm N$. A pn must occur before the initial pseudo-page transition to affect the page number of the first page. The current page number is in the % register.
.po $\pm N$	0; 26/27 in†	previous	v	Page offset. The current <i>left margin</i> is set to $\pm N$. The TROFF initial value provides about 1 inch of paper margin including the physical typesetter margin of 1/27 inch. In TROFF the maximum (line-length)+(page-offset) is about 7.54 inches. See §6. The current page offset is available in the .o register.
.ne <i>N</i>	-	$N=1 V$	D,v	Need <i>N</i> vertical space. If the distance, <i>D</i> , to the next trap position (see §7.5) is less than <i>N</i> , a forward vertical space of size <i>D</i> occurs, which will spring the trap. If there are no remaining traps on the page, <i>D</i> is the distance to the bottom of the page. If $D < V$, another line could still be output and spring the trap. In a diversion,

* The use of "" as control character (instead of ".") suppresses the break function.

† Values separated by ";" are for NROFF and TROFF respectively.

D is the distance to the *diversion trap*, if any, or is very large.

<code>.mk R</code>	none	internal	D	Mark the <i>current</i> vertical place in an internal register (both associated with the current diversion level), or in register <i>R</i> , if given. See <code>rt</code> request.
<code>.rt ±N</code>	none	internal	D,v	Return <i>upward only</i> to a marked vertical place in the current diversion. If ± <i>N</i> (w.r.t. current place) is given, the place is ± <i>N</i> from the top of the page or diversion or, if <i>N</i> is absent, to a place marked by a previous <code>mk</code> . Note that the <code>sp</code> request (§5.3) may be used in all cases instead of <code>rt</code> by spacing to the absolute place stored in a explicit register; e.g., using the sequence <code>.mk Rsp \nRu</code> .

4. Text Filling, Adjusting, and Centering

4.1. Filling and adjusting. Normally, words are collected from input text lines and assembled into a output text line until some word doesn't fit. An attempt is then made the hyphenate the word in effort to assemble a part of it into the output line. The spaces between the words on the output line are then increased to spread out the line to the current *line length* minus any current *indent*. A *word* is any string of characters delimited by the *space* character or the beginning/end of the input line. Any adjacent pair of words that must be kept together (neither split across output lines nor spread apart in the adjustment process) can be tied together by separating them with the *unpaddable space* character “\ ” (backslash-space). The adjusted word spacings are uniform in TROFF and the minimum interword spacing can be controlled with the `ss` request (§2). In NROFF, they are normally nonuniform because of quantization to character-size spaces; however, the command-line option `-e` causes uniform spacing with full output device resolution. Filling, adjustment, and hyphenation (§13) can all be prevented or controlled. The *text length* on the last line output is available in the `.n` register, and text base-line position on the page for this line is in the `.nl` register. The text base-line high-water mark (lowest place) on the current page is in the `.h` register.

An input text line ending with `.`, `?`, or `!` is taken to be the end of a *sentence*, and an additional space character is automatically provided during filling. Multiple inter-word space characters found in the input are retained, except for trailing spaces; initial spaces also cause a *break*.

When filling is in effect, a `\p` may be imbedded or attached to a word to cause a *break* at the *end* of the word and have the resulting output line *spread out* to fill the current line length.

A text input line that happens to begin with a control character can be made to not look like a control line by prefacing it with the non-printing, zero-width filler character `\&`. Still another way is to specify output translation of some convenient character into the control character using `tr` (§10.5).

4.2. Interrupted text. The copying of a input line in *no-fill* (i.e., non-fill) mode can be *interrupted* by terminating the partial line with a `\c`. The *next* encountered input text line will be considered to be a continuation of the same line of input text. Similarly, a word within *filled* text may be interrupted by terminating the word (and line) with `\c`; the next encountered text will be taken as a continuation of the interrupted word. If the intervening control lines cause a break, any partial line will be forced out along with any partial word.

<i>Request Form</i>	<i>Initial Value</i>	<i>If No Argument</i>	<i>Notes</i>	<i>Explanation</i>
<code>.br</code>	-	-	B	Break. The filling of the line currently being collected is stopped and the line is output without adjustment. Text lines beginning with space characters and empty text lines (blank lines) also cause a break.
<code>.fi</code>	fill on	-	B,E	Fill subsequent output lines. The register <code>.u</code> is 1 in fill mode and 0 in no-fill mode.

- .nf** fill on - B,E No-fill. Subsequent output lines are *neither* filled *nor* adjusted. Input text lines are copied directly to output lines *without regard* for the current line length.
- .ad c** adj,both adjust E Line adjustment is begun. If fill mode is not on, adjustment will be deferred until fill mode is back on. If the type indicator *c* is present, the adjustment type is changed as shown in the following table.

Indicator	Adjust Type
l	adjust left margin only
r	adjust right margin only
c	center
b or n	adjust both margins
absent	unchanged

See §2.2 of the Addendum for additional details.

- .na** adjust - E No-adjust. Adjustment is turned off; the right margin will be ragged. The adjustment type for **ad** is not changed. Output line filling still occurs if fill mode is on.
- .ce N** off N=1 B,E Center the next *N* input text lines within the current (line-length minus indent). If *N*=0, any residual count is cleared. A break occurs after each of the *N* input lines. If the input line is too long, it will be left adjusted.

5. Vertical Spacing

5.1. Base-line spacing. The vertical spacing (*V*) between the base-lines of successive output lines can be set using the **vs** request with a resolution of 1/144 inch = 1/2 point in TROFF, and to the output device resolution in NROFF. *V* must be large enough to accommodate the character sizes on the affected output lines. For the common type sizes (9-12 points), usual typesetting practice is to set *V* to 2 points greater than the point size; TROFF default is 10-point type on a 12-point spacing (as in this document). The current *V* is available in the **.v** register. Multiple-*V* line separation (e.g. double spacing) may be requested with **ls**.

5.2. Extra line-space. If a word contains a vertically tall construct requiring the output line containing it to have extra vertical space before and/or after it, the *extra-line-space* function $\backslash x N$ can be imbedded in or attached to that word. In this and other functions having a pair of delimiters around their parameter (here $\backslash x N$), the delimiter choice is arbitrary, except that it can't look like the continuation of a number expression for *N*. If *N* is negative, the output line containing the word will be preceded by *N* extra vertical space; if *N* is positive, the output line containing the word will be followed by *N* extra vertical space. If successive requests for extra space apply to the same line, the maximum values are used. The most recently utilized post-line extra line-space is available in the **.a** register.

5.3. Blocks of vertical space. A block of vertical space is ordinarily requested using **sp**, which honors the *no-space* mode and which does not space *past* a trap. A contiguous block of vertical space may be reserved using **sv**.

Request Form	Initial Value	If No Argument	Notes	Explanation
.vs N	1/6in;12pts	previous	E,p	Set vertical base-line spacing size <i>V</i> . Transient <i>extra</i> vertical space available with $\backslash x N$ (see above).
.ls N	N=1	previous	E	Line spacing set to $\pm N$. <i>N</i> -1 <i>V</i> s (<i>blank lines</i>) are appended to each output text line. Appended blank lines are omitted, if the text or previous appended blank line reached a trap position.

.sp <i>N</i>	-	<i>N=1V</i>	B,v	Space vertically in <i>either</i> direction. If <i>N</i> is negative, the motion is <i>backward</i> (upward) and is limited to the distance to the top of the page. Forward (downward) motion is truncated to the distance to the nearest trap. If the no-space mode is on, no spacing occurs (see ns and rs below).
.sv <i>N</i>	-	<i>N=1V</i>	v	Save a contiguous vertical block of size <i>N</i> . If the distance to the next trap is greater than <i>N</i> , <i>N</i> vertical space is output. No-space mode has <i>no</i> effect. If this distance is less than <i>N</i> , no vertical space is immediately output, but <i>N</i> is remembered for later output (see os). Subsequent sv requests will overwrite any still remembered <i>N</i> .
.os	-	-	-	Output saved vertical space. No-space mode has <i>no</i> effect. Used to finally output a block of vertical space requested by an earlier sv request.
.ns	space	-	D	No-space mode turned on. When on, the no-space mode inhibits sp requests and bp requests <i>without</i> a next page number. The no-space mode is turned off when a line of output occurs, or with rs .
.rs	space	-	D	Restore spacing. The no-space mode is turned off.
Blank text line	-	-	B	Causes a break and outputs a blank line exactly like sp 1 .

6. Line Length and Indenting

The maximum line length for fill mode may be set with **ll**. The indent may be set with **in**; an indent applicable to *only* the *next* output line may be set with **ti**. The line length includes indent space but *not* page offset space. The line-length minus the indent is the basis for centering with **ce**. The effect of **ll**, **in**, or **ti** is delayed, if a partially collected line exists, until after that line is output. In fill mode the length of text on an output line is less than or equal to the line length minus the indent. The current line length and indent are available in registers **.l** and **.i** respectively. The length of *three-part titles* produced by **tl** (see §14) is *independently* set by **lt**.

Request Form	Initial Value	If No Argument	Notes	Explanation
.ll $\pm N$	6.5in	previous	E,m	Line length is set to $\pm N$. In TROFF the maximum (line-length)+(page-offset) is about 7.54 inches.
.in $\pm N$	<i>N=0</i>	previous	B,E,m	Indent is set to $\pm N$. The indent is prepended to each output line.
.ti $\pm N$	-	ignored	B,E,m	Temporary indent. The <i>next</i> output text line will be indented a distance $\pm N$ with respect to the current indent. The resulting total indent may not be negative. The current indent is not changed.

7. Macros, Strings, Diversion, and Position Traps

7.1. Macros and strings. A *macro* is a named set of arbitrary *lines* that may be invoked by name or with a *trap*. A *string* is a named string of *characters*, *not* including a new-line character, that may be interpolated by name at any point. Request, macro, and string names share the *same* name list. Macro and string names may be one or two characters long and may usurp previously defined request, macro, or string names. Any of these entities may be renamed with **rn** or removed with **rm**. Macros are created by **de** and **di**, and appended to by **am** and **da**; **di** and **da** cause normal output to be stored in a macro. Strings are created by **ds** and appended to by **as**. A macro is invoked in the same way as a request; a control line beginning **.xx** will interpolate the contents of macro **xx**. The remainder of the line may contain up to nine *arguments*. The strings *x* and *xx* are interpolated at any desired point with ***x** and ***(xx** respectively. String references and macro invocations may be nested.

7.2. *Copy mode input interpretation.* During the definition and extension of strings and macros (not by diversion) the input is read in *copy mode*. The input is copied without interpretation *except* that:

- The contents of number registers indicated by `\n` are interpolated.
- Strings indicated by `*` are interpolated.
- Arguments indicated by `\$` are interpolated.
- Concealed new-lines indicated by `\(new-line)` are eliminated.
- Comments indicated by `*` are eliminated.
- `\t` and `\a` are interpreted as ASCII horizontal tab and SOH respectively (§9).
- `\\` is interpreted as `\`.
- `\.` is interpreted as `“.”`.

These interpretations can be suppressed by prepending a `\`. For example, since `\\` maps into a `\`, `\\n` will copy as `\n` which will be interpreted as a number register indicator when the macro or string is reread.

7.3. *Arguments.* When a macro is invoked by name, the remainder of the line is taken to contain up to nine arguments. The argument separator is the space character, and arguments may be surrounded by double-quotes to permit imbedded space characters. Pairs of double-quotes may be imbedded in double-quoted arguments to represent a single double-quote. If the desired arguments won't fit on a line, a concealed new-line may be used to continue on the next line.

When a macro is invoked the *input level* is *pushed down* and any arguments available at the previous level become unavailable until the macro is completely read and the previous level is restored. A macro's own arguments can be interpolated at *any* point within the macro with `\$N`, which interpolates the *N*th argument ($1 \leq N \leq 9$). If an invoked argument doesn't exist, a null string results. For example, the macro `xx` may be defined by

```
.de xx      \*begin definition
Today is \\$1 the \\$2.
..         \*end definition
```

and called by

```
.xx Monday 14th
```

to produce the text

```
Today is Monday the 14th.
```

Note that the `\$` was concealed in the definition with a prepended `\`. The number of currently available arguments is in the `.$` register.

No arguments are available at the top (non-macro) level in this implementation. Because string referencing is implemented as a input-level push down, no arguments are available from *within* a string. No arguments are available within a trap-invoked macro.

Arguments are copied in *copy mode* onto a stack where they are available for reference. The mechanism does not allow an argument to contain a direct reference to a *long* string (interpolated at copy time) and it is advisable to conceal string references (with an extra `\`) to delay interpolation until argument reference time.

7.4. *Diversions.* Processed output may be diverted into a macro for purposes such as footnote processing (see Tutorial §T5) or determining the horizontal and vertical size of some text for conditional changing of pages or columns. A single diversion trap may be set at a specified vertical position. The number registers `dn` and `dl` respectively contain the vertical and horizontal size of the most recently ended diversion. Processed text that is diverted into a macro retains the vertical size of each of its lines when reread in *no-fill* mode regardless of the current *V*. Constant-spaced (`cs`) or emboldened (`bd`) text that is diverted can be reread correctly only if these modes are again or still in effect at reread time. One way to do this is to imbed in the diversion the appropriate `cs` or `bd` requests with the *transparent* mechanism described in §10.6.

Diversions may be nested and certain parameters and registers are associated with the current diversion level (the top non-diversion level may be thought of as the 0th diversion level). These are the diversion trap and associated macro, no-space mode, the internally-saved marked place (see **mk** and **rt**), the current vertical place (**.d** register), the current high-water text base-line (**.h** register), and the current diversion name (**.z** register).

7.5. Traps. Three types of trap mechanisms are available—page traps, a diversion trap, and an input-line-count trap. Macro-invocation traps may be planted using **wh** at any page position including the top. This trap position may be changed using **ch**. Trap positions at or below the bottom of the page have no effect unless or until moved to within the page or rendered effective by an increase in page length. Two traps may be planted at the *same* position only by first planting them at different positions and then moving one of the traps; the first planted trap will conceal the second unless and until the first one is moved (see Tutorial Examples §T5). If the first one is moved back, it again conceals the second trap. The macro associated with a page trap is automatically invoked when a line of text is output whose vertical size *reaches* or *sweeps past* the trap position. Reaching the bottom of a page springs the top-of-page trap, if any, provided there is a next page. The distance to the next trap position is available in the **.t** register; if there are no traps between the current position and the bottom of the page, the distance returned is the distance to the page bottom.

A macro-invocation trap effective in the current diversion may be planted using **dt**. The **.t** register works in a diversion; if there is no subsequent trap a *large* distance is returned. For a description of input-line-count traps, see it below.

Request Form	Initial Value	If No Argument	Notes	Explanation
.de <i>xx yy</i>	-	<i>.yy=..</i>	-	Define or redefine the macro <i>xx</i> . The contents of the macro begin on the next input line. Input lines are copied in <i>copy mode</i> until the definition is terminated by a line beginning with <i>.yy</i> , whereupon the macro <i>yy</i> is called. In the absence of <i>yy</i> , the definition is terminated by a line beginning with <i>..</i> . A macro may contain de requests provided the terminating macros differ or the contained definition terminator is concealed; <i>..</i> can be concealed as <i>\\..</i> which will copy as <i>\\..</i> and be reread as <i>..</i> .
.am <i>xx yy</i>	-	<i>.yy=..</i>	-	Append to macro (append version of de).
.ds <i>xx string</i>	-	ignored	-	Define a string <i>xx</i> containing <i>string</i> . Any initial double-quote in <i>string</i> is stripped off to permit initial blanks.
.as <i>xx string</i>	-	ignored	-	Append <i>string</i> to string <i>xx</i> (append version of ds).
.rm <i>xx</i>	-	ignored	-	Remove request, macro, or string. The name <i>xx</i> is removed from the name list and any related storage space is freed. Subsequent references will have no effect.
.rn <i>xx yy</i>	-	ignored	-	Rename request, macro, or string <i>xx</i> to <i>yy</i> . If <i>yy</i> exists, it is first removed.
.di <i>xx</i>	-	end	D	Divert output to macro <i>xx</i> . Normal text processing occurs during diversion except that page offsetting is not done. The diversion ends when the request di or da is encountered without an argument; extraneous requests of this type should not appear when nested diversions are being used.
.da <i>xx</i>	-	end	D	Divert, appending to <i>xx</i> (append version of di).
.wh <i>N xx</i>	-	-	v	Install a trap to invoke <i>xx</i> at page position <i>N</i> ; a <i>negative N</i> will be interpreted with respect to the page <i>bottom</i> . Any

				macro previously planted at N is replaced by xx . A zero N refers to the <i>top</i> of a page. In the absence of xx , the first found trap at N , if any, is removed.
.ch	$xx\ N$	-	-	v Change the trap position for macro xx to be N . In the absence of N , the trap, if any, is removed.
.dt	$N\ xx$	-	off	D,v Install a diversion trap at position N in the <i>current</i> diversion to invoke macro xx . Another dt will redefine the diversion trap. If no arguments are given, the diversion trap is removed.
.it	$N\ xx$	-	off	E Set an input-line-count trap to invoke the macro xx after N lines of <i>text</i> input have been read (control or request lines don't count). The text may be in-line text or text interpolated by in-line or trap-invoked macros.
.em	xx	none	none	- The macro xx will be invoked when all input has ended. The effect is the same as if the contents of xx had been at the end of the last file processed.

8. Number Registers

A variety of parameters are available to the user as predefined, named *number registers* (see Summary and Index, page 7). In addition, the user may define his own named registers. Register names are one or two characters long and *do not* conflict with request, macro, or string names. Except for certain predefined read-only registers, a number register can be read, written, automatically incremented or decremented, and interpolated into the input in a variety of formats. One common use of user-defined registers is to automatically number sections, paragraphs, lines, etc. A number register may be used any time numerical input is expected or desired and may be used in numerical *expressions* (§1.4).

Number registers are created and modified using **nr**, which specifies the name, numerical value, and the auto-increment size. Registers are also modified, if accessed with an auto-incrementing sequence. If the registers x and xx both contain N and have the auto-increment size M , the following access sequences have the effect shown:

Sequence	Effect on Register	Value Interpolated
$\backslash n x$	none	N
$\backslash n (xx$	none	N
$\backslash n +x$	x incremented by M	$N+M$
$\backslash n -x$	x decremented by M	$N-M$
$\backslash n +(xx$	xx incremented by M	$N+M$
$\backslash n -(xx$	xx decremented by M	$N-M$

When interpolated, a number register is converted to decimal (default), decimal with leading zeros, lower-case Roman, upper-case Roman, lower-case sequential alphabetic, or upper-case sequential alphabetic according to the format specified by **af**.

Request Form	Initial Value	If No Argument	Notes	Explanation
.nr $R \pm N M$		-	u	The number register R is assigned the value $\pm N$ with respect to the previous value, if any. The increment for auto-incrementing is set to M .
.af $R c$	arabic	-	-	Assign format c to register R . The available formats are:

Format	Numbering Sequence
1	0,1,2,3,4,5,...
001	000,001,002,003,004,005,...
i	0,i,ii,iii,iv,v,...
I	0,I,II,III,IV,V,...
a	0,a,b,c,....,z,aa,ab,....,zz,aaa,...
A	0,A,B,C,....,Z,AA,AB,....,ZZ,AAA,...

An arabic format having N digits specifies a field width of N digits (example 2 above). The read-only registers and the *width* function (§11.2) are always arabic.

`.rr R` - ignored

Remove register R . If many registers are being created dynamically, it may become necessary to remove no longer used registers to recapture internal storage space for newer registers.

9. Tabs, Leaders, and Fields

9.1. Tabs and leaders. The ASCII horizontal tab character and the ASCII SOH (hereafter known as the *leader* character) can both be used to generate either horizontal motion or a string of repeated characters. The length of the generated entity is governed by internal *tab stops* specifiable with `ta`. The default difference is that tabs generate motion and leaders generate a string of periods; `tc` and `lc` offer the choice of repeated character or motion. There are three types of internal tab stops—*left* adjusting, *right* adjusting, and *centering*. In the following table: D is the distance from the current position on the *input* line (where a tab or leader was found) to the next tab stop; *next-string* consists of the input characters following the tab (or leader) up to the next tab (or leader) or end of line; and W is the width of *next-string*.

Tab type	Length of motion or repeated characters	Location of <i>next-string</i>
Left	D	Following D
Right	$D - W$	Right adjusted within D
Centered	$D - W/2$	Centered on right end of D

The length of generated motion is allowed to be negative, but that of a repeated character string cannot be. Repeated character strings contain an integer number of characters, and any residual distance is prepended as motion. Tabs or leaders found after the last tab stop are ignored, but may be used as *next-string* terminators.

Tabs and leaders are not interpreted in *copy mode*. `\t` and `\a` always generate a non-interpreted tab and leader respectively, and are equivalent to actual tabs and leaders in *copy mode*.

9.2. Fields. A *field* is contained between a *pair* of *field delimiter* characters, and consists of sub-strings separated by *padding* indicator characters. The field length is the distance on the *input* line from the position where the field begins to the next tab stop. The difference between the total length of all the sub-strings and the field length is incorporated as horizontal padding space that is divided among the indicated padding places. The incorporated padding is allowed to be negative. For example, if the field delimiter is `#` and the padding indicator is `^`, `#^xxx^right#` specifies a right-adjusted string with the string `xxx` centered in the remaining space.

<i>Request Form</i>	<i>Initial Value</i>	<i>If No Argument</i>	<i>Notes</i>	<i>Explanation</i>
.ta <i>Nt</i> ...	8n; 0.5in	none	E,m	Set tab stops and types. <i>t</i> =R, right adjusting; <i>t</i> =C, centering; <i>t</i> absent, left adjusting. TROFF tab stops are preset every 0.5in.; NROFF every 8 nominal character widths. The stop values are separated by spaces, and a value preceded by + is treated as an increment to the previous stop value.
.tc <i>c</i>	none	none	E	The tab repetition character becomes <i>c</i> , or is removed specifying motion.
.lc <i>c</i>	.	none	E	The leader repetition character becomes <i>c</i> , or is removed specifying motion.
.fc <i>a b</i>	off	off	-	The field delimiter is set to <i>a</i> ; the padding indicator is set to the <i>space</i> character or to <i>b</i> , if given. In the absence of arguments the field mechanism is turned off.

10. Input and Output Conventions and Character Translations

10.1. Input character translations. Ways of typing in the graphic character set were discussed in §2.1. The ASCII control characters SOH (§9.1), horizontal tab (§9.1), and backspace (§10.3) are discussed elsewhere; the new-line delimits input lines. In addition, STX, ETX, ENQ, ACK, BEL, SO, SI, and ESC may be used as delimiters or translated into a graphic with **tr** (§10.5); TROFF normally passes none of these characters to its output (but it passes all 8 if invoked with the **-a** command-line option); NROFF passes the last 4, with their effect depending on the output device used. *All* others are ignored.

The *escape* character **** introduces *escape sequences*—causes the following character to mean another character, or to indicate some function. A complete list of such sequences is given in the Summary and Index on page 6. **** should not be confused with the ASCII control character ESC of the same name. The escape character **** can be input with the sequence ****. The escape character can be changed with **ec**, and all that has been said about the default **** becomes true for the new escape character. **\e** can be used to print whatever the current escape character is. If necessary or convenient, the escape mechanism may be turned off with **eo**, and restored with **ec**.

<i>Request Form</i>	<i>Initial Value</i>	<i>If No Argument</i>	<i>Notes</i>	<i>Explanation</i>
.ec <i>c</i>	\	\	-	Set escape character to \ , or to <i>c</i> , if given.
.eo	on	-	-	Turn escape mechanism off.

10.2. Ligatures. Five ligatures are available in the current TROFF character set: **fi**, **fl**, **ff**, **ffi**, and **ffl**. They may be input (even in NROFF) by **\(fi**, **\(fl**, **\(ff**, **\(Fi**, and **\(Fl** respectively. The ligature mode is normally on in TROFF, and *automatically* invokes ligatures during input.

<i>Request Form</i>	<i>Initial Value</i>	<i>If No Argument</i>	<i>Notes</i>	<i>Explanation</i>
.lg <i>N</i>	off; on	on	-	Ligature mode is turned on if <i>N</i> is absent or non-zero, and turned off if <i>N</i> =0. If <i>N</i> =2, only the two-character ligatures are automatically invoked. Ligature mode is inhibited for request, macro, string, register, or file names, and in <i>copy mode</i> . No effect in NROFF.

10.3. Backspacing, underlining, overstriking, etc. Unless in *copy mode*, the ASCII backspace character is replaced by a backward horizontal motion having the width of the space character. Underlining as a form of line-drawing is discussed in §12.4. A generalized overstriking function is described in §12.1.

NROFF automatically underlines characters in the *underline* font specifiable with **uf** (normally Times Italic on font position 2—see §2.2). In addition to **ft** and **\fF**, the underline font is selected by **ul** and **cu**. Underlining is restricted to an output-device-dependent subset of *reasonable* characters.

<i>Request Form</i>	<i>Initial Value</i>	<i>If No Argument</i>	<i>Notes</i>	<i>Explanation</i>
<code>.ul N</code>	off	$N=1$	E	Underline in NROFF (italicize in TROFF) the next N input text lines. Actually, switch to <i>underline</i> font, saving the current font for later restoration; <i>other</i> font changes within the span of a <code>ul</code> will take effect, but the restoration will undo the last change. Output generated by <code>tl</code> (§14) is affected by the font change, but does <i>not</i> decrement N . If $N > 1$, there is the risk that a trap interpolated macro may provide text lines within the span; environment switching can prevent this.
<code>.cu N</code>	off	$N=1$	E	A variant of <code>ul</code> that causes <i>every</i> character to be underlined in NROFF. Identical to <code>ul</code> in TROFF.
<code>.uf F</code>	Italic	Italic	-	Underline font set to F . In NROFF, F may <i>not</i> be on position 1 (initially Times Roman).

10.4. *Control characters.* Both the control character `.` and the *no-break* control character `~` may be changed, if desired. Such a change must be compatible with the design of any macros used in the span of the change, and particularly of any trap-invoked macros.

<i>Request Form</i>	<i>Initial Value</i>	<i>If No Argument</i>	<i>Notes</i>	<i>Explanation</i>
<code>.cc c</code>	.	.	E	The basic control character is set to c , or reset to <code>."</code> .
<code>.c2 c</code>	.	.	E	The <i>no-break</i> control character is set to c , or reset to <code>~"</code> .

10.5. *Output translation.* One character can be made a stand-in for another character using `tr`. All text processing (e.g., character comparisons) takes place with the input (stand-in) character which appears to have the width of the final character. The graphic translation occurs at the moment of output (including diversion).

<i>Request Form</i>	<i>Initial Value</i>	<i>If No Argument</i>	<i>Notes</i>	<i>Explanation</i>
<code>.tr abcd....</code>	none	-	O	Translate a into b , c into d , etc. If an odd number of characters is given, the last one will be mapped into the space character. To be consistent, a particular translation must stay in effect from <i>input</i> to <i>output</i> time.

10.6. *Transparent throughput.* An input line beginning with a `\!` is read in *copy mode* and *transparently* output (without the initial `\!`); the text processor is otherwise unaware of the line's presence. This mechanism may be used to pass control information to a post-processor or to imbed control lines in a macro created by a diversion.

10.7. *Comments and concealed new-lines.* An uncomfortably long input line that must stay one line (e.g., a string definition, or no-filled text) can be split into many physical lines by ending all but the last one with the escape `\`. The sequence `\(new-line)` is *always* ignored—except in a comment. Comments may be imbedded at the *end* of any line by prefacing them with `\`. The new-line at the end of a comment cannot be concealed. A line beginning with `\` will appear as a blank line and behave like `.sp 1`; a comment can be on a line by itself by beginning the line with `\`.

11. Local Horizontal and Vertical Motions, and the Width Function

11.1. *Local Motions.* The functions `\vN` and `\hN` can be used for *local* vertical and horizontal motion respectively. The distance N may be negative; the *positive* directions are *rightward* and *downward*. A *local* motion is one contained *within* a line. To avoid unexpected vertical dislocations, it is necessary that the *net* vertical local motion within a word in filled text and otherwise within a line balance to zero. The above and certain other escape sequences providing local motion are summarized in the following table.

Vertical Local Motion	Effect in		Horizontal Local Motion	Effect in	
	TROFF	NROFF		TROFF	NROFF
<code>\v N</code>	Move distance <i>N</i>		<code>\h N</code> <code>\(space)</code> <code>\0</code>	Move distance <i>N</i> Unpaddable space-size space Digit-size space	
<code>\u</code> <code>\d</code> <code>\r</code>	½ em up ½ em down 1 em up	½ line up ½ line down 1 line up	<code>\ </code> <code>\`</code>	1/6 em space 1/12 em space	ignored ignored

As an example, E^2 could be generated by the sequence `E\s-2\v-.04m^2\v.04m\s+2`; it should be noted in this example that the 0.4 em vertical motions are at the smaller size.

11.2. Width Function. The *width* function `\w string` generates the numerical width of *string* (in basic units). Size and font changes may be safely imbedded in *string*, and will not affect the current environment. For example, `.ti-\w 1. u` could be used to temporarily indent leftward a distance equal to the size of the string "1."

The width function also sets three number registers. The registers `st` and `sb` are set respectively to the highest and lowest extent of *string* relative to the baseline; then, for example, the total *height* of the string is `\n(stu-\n(sbu)`. In TROFF the number register `ct` is set to a value between 0 and 3: 0 means that all of the characters in *string* were short lower case characters without descenders (like *e*); 1 means that at least one character has a descender (like *y*); 2 means that at least one character is tall (like **H**); and 3 means that both tall characters and characters with descenders are present.

11.3. Mark horizontal place. The escape sequence `\kx` will cause the *current* horizontal position in the *input line* to be stored in register *x*. As an example, the construction `\kxword\h\|nxu+2u word` will embolden *word* by backing up to almost its beginning and overprinting it, resulting in **word**.

12. Overstrike, Bracket, Line-drawing, and Zero-width Functions

12.1. Overstriking. Automatically centered overstriking of up to nine characters is provided by the *overstrike* function `\o string`. The characters in *string* are overprinted with centers aligned; the total width is that of the widest character. *string* should *not* contain local vertical motion. As examples, `\o e` produces \acute{e} , and `\o\mo\sl` produces \pounds .

12.2. Zero-width characters. The function `\zc` will output *c* without spacing over it, and can be used to produce left-aligned overstruck combinations. As examples, `\z\ci\pl` will produce \oplus , and `\br\z\rn\ul\br` will produce the smallest possible constructed box \square .

12.3. Large Brackets. The Special Mathematical Font contains a number of bracket construction pieces (() [] { } [] []) that can be combined into various bracket styles. The function `\b string` may be used to pile up vertically the characters in *string* (the first character on top and the last at the bottom); the characters are vertically separated by 1 em and the total pile is centered 1/2 em above the current baseline (½ line in NROFF). For example, `\b\lc\lf E\|b\rc\rf x-.05m x .05m` produces $\left[E \right]$.

12.4. Line drawing. The function `\l Nc` will draw a string of repeated *c*'s towards the right for a distance *N*. (`\l` is `\(lower case L)`. If *c* looks like a continuation of an expression for *N*, it may insulated from *N* with a `\&`. If *c* is not specified, the `_` (baseline rule) is used (underline character in NROFF). If *N* is negative, a backward horizontal motion of size *N* is made *before* drawing the string. Any space resulting from *N*/(size of *c*) having a remainder is put at the beginning (left end) of the string. In the case of characters that are designed to be connected such as baseline-rule `_`, underrule `_`, and root-en `^`, the remainder space is covered by over-lapping. If *N* is *less* than the width of *c*, a single *c* is centered on a distance *N*. As an example, a macro to underscore a string can be written

```
.de us
\|1\l^0\ul
..
```

or one to draw a box around a string

```
.de bx
\(\br\|\\$1\|(\br\|'|0\(\rn\|'|0\(\ul'
```

such that

```
.us "underlined words"
```

and

```
.bx "words in a box"
```

yield underlined words and words in a box.

The function $\backslash L^N c^$ will draw a vertical line consisting of the (optional) character c stacked vertically apart 1 em (1 line in NROFF), with the first two characters overlapped, if necessary, to form a continuous line. The default character is the *box rule* | ($\backslash(\br)$); the other suitable character is the *bold vertical* | ($\backslash(\bv)$). The line is begun without any initial motion relative to the current base line. A positive N specifies a line drawn downward and a negative N specifies a line drawn upward. After the line is drawn no compensating motions are made; the instantaneous baseline is at the *end* of the line.

The horizontal and vertical line drawing functions may be used in combination to produce large boxes. The zero-width *box-rule* and the 1/2-em wide *underrule* were *designed* to form corners when using 1-em vertical spacings. For example the macro

```
.de eb
.sp -1      \*compensate for next automatic base-line spacing
.nf        \*avoid possibly overflowing word buffer
\h-.5n\l'|\\nau-1\|'\n(.lu+1n\(\ul\l'-|\\nau+1\|'|0u-.5n\(\ul'  \*draw box
.fi
..
```

will draw a box around some text whose beginning vertical place was saved in number register a (e.g., using $\text{.mk } a$) as done for this paragraph.

13. Hyphenation.

The automatic hyphenation may be switched off and on. When switched on with **hy**, several variants may be set. A *hyphenation indicator* character may be imbedded in a word to specify desired hyphenation points, or may be prepended to suppress hyphenation. In addition, the user may specify a small exception word list.

Only words that consist of a central alphabetic string surrounded by (usually null) non-alphabetic strings are considered candidates for automatic hyphenation. Words that were input containing hyphens (minus), em-dashes ($\backslash(\em)$), or hyphenation indicator characters—such as mother-in-law—are *always* subject to splitting after those characters, whether or not automatic hyphenation is on or off.

<i>Request Form</i>	<i>Initial Value</i>	<i>If No Argument</i>	<i>Notes</i>	<i>Explanation</i>
.nh	no hyphen.	-	E	Automatic hyphenation is turned off.
.hy N	off, $N=0$	on, $N=1$	E	Automatic hyphenation is turned on for $N \geq 1$, or off for $N=0$. If $N=2$, <i>last</i> lines (ones that will cause a trap) are not hyphenated. For $N=4$ and 8, the last and first two characters respectively of a word are not split off. These values are additive; i.e., $N=14$ will invoke all three restrictions.
.hc c	\%	\%	E	Hyphenation indicator character is set to c or to the default \%. The indicator does not appear in the output.
.hw $word1$...		ignored	-	Specify hyphenation points in words with imbedded minus signs. Versions of a word with terminal s are

implied; i.e., *dig-it* implies *dig-its*. This list is examined initially *and* after each suffix stripping. The space available is small—about 128 characters.

14. Three Part Titles.

The titling function **tl** provides for automatic placement of three fields at the left, center, and right of a line with a title-length specifiable with **lt**. **tl** may be used anywhere, and is independent of the normal text collecting process. A common use is in header and footer macros.

<i>Request Form</i>	<i>Initial Value</i>	<i>If No Argument</i>	<i>Notes</i>	<i>Explanation</i>
.tl <i>left center right</i>		-	-	The strings <i>left</i> , <i>center</i> , and <i>right</i> are respectively left-adjusted, centered, and right-adjusted in the current title-length. Any of the strings may be empty, and overlapping is permitted. If the page-number character (initially %) is found within any of the fields it is replaced by the current page number having the format assigned to register %. Any character may be used as the string delimiter.
.pc <i>c</i>	%	off	-	The page number character is set to <i>c</i> , or removed. The page-number register remains %.
.lt $\pm N$	6.5in	previous	E,m	Length of title set to $\pm N$. The line-length and the title-length are <i>independent</i> . Indents do not apply to titles; page-offsets do.

15. Output Line Numbering.

Automatic sequence numbering of output lines may be requested with **nm**. When in effect, a three-digit, arabic number plus a digit-space is prepended to output text lines. The text lines are thus offset by four digit-spaces, and otherwise retain their line length; a reduction in line length may be desired to keep the right margin aligned with an earlier margin. Blank lines, other vertical spaces, and lines generated by **tl** are *not* numbered. Numbering can be temporarily suspended with **nn**, or with an **.nm** followed by a later **.nm +0**. In addition, a line number indent *I*, and the number-text separation *S* may be specified in digit-spaces. Further, it can be specified that only those line numbers that are multiples of some number *M* are to be printed (the others will appear as blank number fields).

<i>Request Form</i>	<i>Initial Value</i>	<i>If No Argument</i>	<i>Notes</i>	<i>Explanation</i>
.nm $\pm N M S I$		off	E	Line number mode. If $\pm N$ is given, line numbering is turned on, and the next output line numbered is numbered $\pm N$. Default values are $M=1$, $S=1$, and $I=0$. Parameters corresponding to missing arguments are unaffected; a non-numeric argument is considered missing. In the absence of all arguments, numbering is turned off; the next line number is preserved for possible further use in number register ln .
.nn <i>N</i>	-	$N=1$	E	The next <i>N</i> text output lines are not numbered.

As an example, the paragraph portions of this section are numbered with $M=3$: **.nm 1 3** was placed at the beginning; **.nm** was placed at the end of the first paragraph; and **.nm +0** was placed 12 in front of this paragraph; and **.nm** finally placed at the end. Line lengths were also changed (by **\w'0000'u**) to keep the right side aligned. Another example is **.nm +5 5 x 3** which turns on numbering with the line number of the next line to be 5 greater than the last numbered line, with 15 $M=5$, with spacing *S* untouched, and with the indent *I* set to 3.

16. Conditional Acceptance of Input

In the following, *c* is a one-character, built-in *condition* name, *!* signifies *not*, *N* is a numerical expression, *string1* and *string2* are strings delimited by any non-blank, non-numeric character *not* in the strings, and *anything* represents what is conditionally accepted.

<i>Request Form</i>	<i>Initial Value</i>	<i>If No Argument</i>	<i>Notes</i>	<i>Explanation</i>
<code>.if <i>c anything</i></code>		-	-	If condition <i>c</i> true, accept <i>anything</i> as input; in multi-line case use <code>\{anything\}</code> .
<code>.if !<i>c anything</i></code>		-	-	If condition <i>c</i> false, accept <i>anything</i> .
<code>.if <i>N anything</i></code>		-	u	If expression $N > 0$, accept <i>anything</i> .
<code>.if !<i>N anything</i></code>		-	u	If expression $N \leq 0$, accept <i>anything</i> .
<code>.if '<i>string1 string2</i>' <i>anything</i></code>		-	-	If <i>string1</i> identical to <i>string2</i> , accept <i>anything</i> .
<code>.if !'<i>string1 string2</i>' <i>anything</i></code>		-	-	If <i>string1</i> not identical to <i>string2</i> , accept <i>anything</i> .
<code>.ie <i>c anything</i></code>		-	u	If portion of if-else; all above forms (like <code>if</code>).
<code>.el <i>anything</i></code>		-	-	Else portion of if-else.

The built-in condition names are:

Condition Name	True If
o	Current page number is odd
e	Current page number is even
t	Formatter is TROFF
n	Formatter is NROFF

If the condition *c* is *true*, or if the number *N* is greater than zero, or if the strings compare identically (including motions and character size and font), *anything* is accepted as input. If a *!* precedes the condition, number, or string comparison, the sense of the acceptance is reversed.

Any spaces between the condition and the beginning of *anything* are skipped over. The *anything* can be either a single input line (text, macro, or whatever) or a number of input lines. In the multi-line case, the first line must begin with a left delimiter `\{` and the last line must end with a right delimiter `\}`.

The request `ie` (if-else) is identical to `if` except that the acceptance state is remembered. A subsequent and matching `el` (else) request then uses the reverse sense of that state. `ie - el` pairs may be nested.

Some examples are:

```
.if e .tl 'Even Page %'
```

which outputs a title if the page number is even; and

```
.ie \n%>1 \{\
.sp 0.5i
.tl 'Page %'
.sp |1.2i \}
.el .sp |2.5i
```

which treats page 1 differently from other pages.

17. Environment Switching.

A number of the parameters that control the text processing are gathered together into an *environment*, which can be switched by the user. The environment parameters are those associated with requests noting **E** in their *Notes* column; in addition, partially collected lines and words are in the environment. Everything else is global; examples are page-oriented parameters, diversion-oriented parameters,

number registers, and macro and string definitions. All environments are initialized with default parameter values.

<i>Request Form</i>	<i>Initial Value</i>	<i>If No Argument</i>	<i>Notes</i>	<i>Explanation</i>
<code>.ev N</code>	$N=0$	previous	-	Environment switched to environment $0 \leq N \leq 2$. Switching is done in push-down fashion so that restoring a previous environment <i>must</i> be done with <code>.ev</code> rather than specific reference.

18. Insertions from the Standard Input

The input can be temporarily switched to the system *standard input* with `rd`, which will switch back when *two* new-lines in a row are found (the *extra* blank line is not used). This mechanism is intended for insertions in form-letter-like documentation. On UNIX, the *standard input* can be the user's keyboard, a *pipe*, or a *file*.

<i>Request Form</i>	<i>Initial Value</i>	<i>If No Argument</i>	<i>Notes</i>	<i>Explanation</i>
<code>.rd prompt</code>	-	<code>prompt=BEL-</code>	-	Read insertion from the standard input until two new-lines in a row are found. If the standard input is the user's keyboard, <i>prompt</i> (or a BEL) is written onto the user's terminal. <code>rd</code> behaves like a macro, and arguments may be placed after <i>prompt</i> .
<code>.ex</code>	-	-	-	Exit from NROFF/TROFF. Text processing is terminated exactly as if all input had ended.

If insertions are to be taken from the terminal keyboard *while* output is being printed on the terminal, the command-line option `-q` will turn off the echoing of keyboard input and prompt only with BEL. The regular input and insertion input *cannot* simultaneously come from the standard input.

As an example, multiple copies of a form letter may be prepared by entering the insertions for all the copies in one file to be used as the standard input, and causing the file containing the letter to reinvoke itself using `nx` (§19); the process would ultimately be ended by an `ex` in the insertion file.

19. Input/Output File Switching

<i>Request Form</i>	<i>Initial Value</i>	<i>If No Argument</i>	<i>Notes</i>	<i>Explanation</i>
<code>.so filename</code>	-	-	-	Switch source file. The top input (file reading) level is switched to <i>filename</i> . When the new file ends, input is again taken from the original file; <code>so</code> 's may be nested. See §2.2 of the Addendum for additional details.
<code>.nx filename</code>	-	end-of-file	-	Next file is <i>filename</i> . The current file is considered ended, and the input is immediately switched to <i>filename</i> .
<code>.pi program</code>	-	-	-	Pipe output to <i>program</i> (NROFF only). This request must occur <i>before</i> any printing occurs. No arguments are transmitted to <i>program</i> .

20. Miscellaneous

<i>Request Form</i>	<i>Initial Value</i>	<i>If No Argument</i>	<i>Notes</i>	<i>Explanation</i>
<code>.mc c N</code>	-	off	E,m	Specifies that a <i>margin</i> character <i>c</i> appear a distance <i>N</i> to the right of the right margin after each non-empty text line (except those produced by <code>tl</code>). If the output line is too-long (as can happen in no-fill mode) the character will be appended to the line. If <i>N</i> is not given, the

			previous <i>N</i> is used; the initial <i>N</i> is 0.2 inches in NROFF and 1 em in TROFF. The margin character used with this paragraph was a 12-point box-rule.	
.tm <i>string</i>	-	new-line	-	After skipping initial blanks, <i>string</i> (rest of the line) is read in <i>copy mode</i> and written on the user's terminal.
.ig <i>yy</i>	-	.yy=..	-	Ignore input lines. ig behaves exactly like de (§7) except that the input is discarded. The input is read in <i>copy mode</i> , and any auto-incremented registers will be affected.
.pm <i>t</i>	-	all	-	Print macros. The names and sizes of all of the defined macros and strings are printed on the user's terminal; if <i>t</i> is given, only the total of the sizes is printed. The sizes is given in <i>blocks</i> of 128 characters.
.fl	-	-	B	Flush output buffer. Used in interactive debugging to force output.

21. Output and Error Messages.

The output from **tm**, **pm**, and the prompt from **rd**, as well as various *error* messages are written onto UNIX's *standard message* output. The latter is different from the *standard output*, where NROFF formatted output goes. By default, both are written onto the user's terminal, but they can be independently redirected.

Various *error* conditions may occur during the operation of NROFF and TROFF. Certain less serious errors having only local impact do not cause processing to terminate. Two examples are *word overflow*, caused by a word that is too large to fit into the word buffer (in fill mode), and *line overflow*, caused by an output line that grew too large to fit in the line buffer; in both cases, a message is printed, the offending excess is discarded, and the affected word or line is marked at the point of truncation with a * in NROFF and a ~~~~~ in TROFF. The philosophy is to continue processing, if possible, on the grounds that output useful for debugging may be produced. If a serious error occurs, processing terminates, and an appropriate message is printed. Examples are the inability to create, read, or write files, and the exceeding of certain internal limits that make future output unlikely to be useful.

TUTORIAL EXAMPLES

T1. Introduction

Although NROFF and TROFF have by design a syntax reminiscent of earlier text processors* with the intent of easing their use, it is almost always necessary to prepare at least a small set of macro definitions to describe most documents. Such common formatting needs as page margins and footnotes are deliberately not built into NROFF and TROFF. Instead, the macro and string definition, number register, diversion, environment switching, page-position trap, and conditional input mechanisms provide the basis for user-defined implementations.

The examples to be discussed are intended to be useful and somewhat realistic, but won't necessarily cover all relevant contingencies. Explicit numerical parameters are used in the examples to make them easier to read and to illustrate typical values. In many cases, number registers would really be used to reduce the number of places where numerical information is kept, and to concentrate conditional parameter initialization like that which depends on whether TROFF or NROFF is being used.

T2. Page Margins

As discussed in §3, *header* and *footer* macros are usually defined to describe the top and bottom page margin areas respectively. A trap is planted at page position 0 for the header, and at $-N$ (N from the page bottom) for the footer. The simplest such definitions might be

```
.de hd      \define header
.sp 1i
..         \end definition
.de fo      \define footer
.bp
..         \end definition
.wh 0 hd
.wh -1i fo
```

which provide blank 1 inch top and bottom margins. The header will occur on the *first* page, only if the definition and trap exist prior to the

initial pseudo-page transition (§3). In fill mode, the output line that springs the footer trap was typically forced out because some part or whole word didn't fit on it. If anything in the footer and header that follows causes a *break*, that word or part word will be forced out. In this and other examples, requests like **bp** and **sp** that normally cause breaks are invoked using the *no-break* control character `^` to avoid this. When the header/footer design contains material requiring independent text processing, the environment may be switched, avoiding most interaction with the running text.

A more realistic example would be

```
.de hd      \header
.if t .tl ^\(\rn^\(\rn ^troff cut mark
.if \n%>1 \{\
.sp |0.5i-1 ^tl base at 0.5i
.tl ^- % -^ \centered page number
.ps        \restore size
.ft        \restore font
.vs \}     \restore vs
.sp |1.0i   \space to 1.0i
.ns        \turn on no-space mode
..
.de fo      \footer
.ps 10     \set footer/header size
.ft R      \set font
.vs 12p    \set base-line spacing
.if \n%=1 \{\
.sp |\n(.pu-0.5i-1 ^tl base 0.5i up
.tl ^- % -^ \} \first page number
.bp
..
.wh 0 hd
.wh -1i fo
```

which sets the size, font, and base-line spacing for the header/footer material, and ultimately restores them. The material in this case is a page number at the bottom of the first page and at the top of the remaining pages. If TROFF is used, a *cut mark* is drawn in the form of *root-en's* at each margin. The **sp's** refer to absolute positions to avoid dependence on the base-line spacing. Another reason for this in the footer is that the footer is invoked by printing a line whose vertical spacing swept past the trap position by possibly as

* For example: P. A. Crisman, Ed., *The Compatible Time-Sharing System*, MIT Press, 1965, Section AH9.01 (Description of RUNOFF program on MIT's CTSS system).

much as the base-line spacing. The *no-space* mode is turned on at the end of **hd** to render ineffective accidental occurrences of **sp** at the top of the running text.

The above method of restoring size, font, etc. presupposes that such requests (that set *previous* value) are *not* used in the running text. A better scheme is save and restore both the current *and* previous values as shown for size in the following:

```
.de fo
.nr s1 \\n(.s  \\'current size
.ps
.nr s2 \\n(.s  \\'previous size
. ---        \\'rest of footer
..
.de hd
. ---        \\'header stuff
.ps \\n(s2    \\'restore previous size
.ps \\n(s1    \\'restore current size
..
```

Page numbers may be printed in the bottom margin by a separate macro triggered during the footer's page ejection:

```
.de bn        \\'bottom number
.tl -- % --  \\'centered page number
..
.wh -0.5i-1v bn \\tl base 0.5i up
```

T3. Paragraphs and Headings

The housekeeping associated with starting a new paragraph should be collected in a paragraph macro that, for example, does the desired preparagraph spacing, forces the correct font, size, base-line spacing, and indent, checks that enough space remains for *more than one* line, and requests a temporary indent.

```
.de pg        \\'paragraph
.br          \\'break
.ft R       \\'force font,
.ps 10      \\'size,
.vs 12p     \\'spacing,
.in 0       \\'and indent
.sp 0.4     \\'prespace
.ne 1+\\n(.Vu \\'want more than 1 line
.ti 0.2i    \\'temp indent
..
```

The first break in **pg** will force out any previous partial lines, and must occur before the **vs**. The forcing of font, etc. is partly a defense against prior error and partly to permit things like section heading macros to set parameters only once.

The prespacing parameter is suitable for TROFF; a larger space, at least as big as the output device vertical resolution, would be more suitable in NROFF. The choice of remaining space to test for in the **ne** is the smallest amount greater than one line (the **.V** is the available vertical resolution).

A macro to automatically number section headings might look like:

```
.de sc        \\'section
. ---        \\'force font, etc.
.sp 0.4      \\'prespace
.ne 2.4+\\n(.Vu \\'want 2.4+ lines
.fi
\\n+S.
..
.nr S 0 1    \\'init S
```

The usage is **.sc**, followed by the section heading text, followed by **.pg**. The **ne** test value includes one line of heading, 0.4 line in the following **pg**, and one line of the paragraph text. A word consisting of the next section number and a period is produced to begin the heading line. The format of the number may be set by **af** (§8).

Another common form is the labeled, indented paragraph, where the label protrudes left into the indent space.

```
.de lp        \\'labeled paragraph
.pg
.in 0.5i     \\'paragraph indent
.ta 0.2i 0.5i \\'label, paragraph
.ti 0
\\t\\$1\\t\\c \\'flow into paragraph
..
```

The intended usage is **“.lp label”**; *label* will begin at 0.2inch, and cannot exceed a length of 0.3inch without intruding into the paragraph. The label could be right adjusted against 0.4inch by setting the tabs instead with **.ta 0.4iR 0.5i**. The last line of **lp** ends with **\\c** so that it will become a part of the first line of the text that follows.

T4. Multiple Column Output

The production of multiple column pages requires the footer macro to decide whether it was invoked by other than the last column, so that it will begin a new column rather than produce the bottom margin. The header can initialize a column register that the footer will increment and test. The following is arranged for two columns, but is easily modified for more.

```
.de hd      \header
. ---
.nr cl 0 1  \init column count
.mk        \mark top of text
..
.de fo      \footer
.ie \n+(cl<2 \{\
.po +3.4i   \next column; 3.1+0.3
.rt        \back to mark
.ns \}     \no-space mode
.el \{\
.po \nMu    \restore left margin
. ---
.bp \}
..
.ll 3.1i    \column width
.nr M \n(.o \save left margin
```

Typically a portion of the top of the first page contains full width text; the request for the narrower line length, as well as another .mk would be made where the two column output was to begin.

T5. Footnote Processing

The footnote mechanism to be described is used by imbedding the footnotes in the input text at the point of reference, demarcated by an initial .fn and a terminal .ef:

```
.fn
Footnote text and control lines...
.ef
```

In the following, footnotes are processed in a separate environment and diverted for later printing in the space immediately prior to the bottom margin. There is provision for the case where the last collected footnote doesn't completely fit in the available space.

```
.de hd      \header
. ---
.nr x 0 1   \init footnote count
.nr y 0-\nb \current footer place
.ch fo -\nbu \reset footer trap
.if \n(dn .fz \leftover footnote
..
.de fo      \footer
.nr dn 0    \zero last diversion size
.if \nx \{\
.ev 1       \expand footnotes in ev1
.nf        \retain vertical size
.FN        \footnotes
.rm FN     \delete it
.if "\n(.z"fy" .di \end overflow diversion
.nr x 0     \disable fx
```

```
.ev \}     \pop environment
. ---
.bp
..
.de fx      \process footnote overflow
.if \nx .di fy \divert overflow
..
.de fn      \start footnote
.da FN     \divert (append) footnote
.ev 1      \in environment 1
.if \n+x=1 .fs \if first, include separator
.fi        \fill mode
..
.de ef      \end footnote
.br        \finish output
.nr z \n(.v \save spacing
.ev        \pop ev
.di        \end diversion
.nr y -\n(dn \new footer position,
.if \nx=1 .nr y -(\n(.v-\nz) \
\uncertainty correction
.ch fo \nyu \y is negative
.if (\n(nl+1v)>(\n(.p+\ny) \
.ch fo \n(nlu+1v \it didn't fit
..
.de fs      \separator
\l 1i      \1 inch rule
.br
..
.de fz      \get leftover footnote
.fn
.nf        \retain vertical size
.fy        \where fx put it
.ef
..
.nr b 1.0i  \bottom margin size
.wh 0 hd   \header trap
.wh 12i fo \footer trap, temp position
.ch fo -\nbu \conceal fx with fo
```

The header **hd** initializes a footnote count register **x**, and sets both the current footer trap position register **y** and the footer trap itself to a nominal position specified in register **b**. In addition, if the register **dn** indicates a leftover footnote, **fz** is invoked to reprocess it. The footnote start macro **fn** begins a diversion (append) in environment 1, and increments the count **x**; if the count is one, the footnote separator **fs** is interpolated. The separator is kept in a separate macro to permit user redefinition. The footnote end macro **ef** restores the previous environment and ends the diversion after saving the spacing size in register **z**; **y** is then decremented by the size of the

footnote, available in **dn**; then on the first footnote, **y** is further decremented by the difference in vertical base-line spacings of the two environments, to prevent the late triggering the footer trap from causing the last line of the combined footnotes to overflow. The footer trap is then set to the lower (on the page) of **y** or the current page position (**nl**) plus one line, to allow for printing the reference line. If indicated by **x**, the footer **fo** rereads the footnotes from **FN** in no-fill mode in environment 1, and deletes **FN**. If the footnotes were too large to fit, the macro **fx** will be trap-invoked to redirect the overflow into **fy**, and the register **dn** will later indicate to the header whether **fy** is empty. Both **fo** and **fx** are planted in the nominal footer trap position in an order that causes **fx** to be concealed unless the **fo** trap is moved. The footer then terminates the overflow diversion, if necessary, and zeros **x** to disable **fx**, because the uncertainty correction together with a not-too-late triggering of the footer can result in the footnote rereading finishing before reaching the **fx** trap.

A good exercise for the student is to combine the multiple-column and footnote mechanisms.

T6. The Last Page

After the last input file has ended, NROFF and TROFF invoke the *end macro* (§7), if any, and when it finishes, eject the remainder of the page. During the eject, any traps encountered are processed normally. At the *end* of this last page, processing terminates *unless* a partial line, word, or partial word remains. If it is desired that another page be started, the end-macro

```
.de en      \end-macro
\c
bp
..
.em en
```

will deposit a null partial word, and effect another last page.

Table I

Font Style Examples

The following font examples are printed in 12-point, with a vertical spacing of 14-point, and with non-alphanumeric characters separated by ¼ em space. The original Special Mathematical Font was specially prepared for Bell Laboratories by Wang Laboratories, Inc., of Hudson, New Hampshire. The Times Roman, Italic, and Bold are among the many standard fonts available.

Times Roman

abcdefghijklmnopqrstuvwxy
ABCDEFGHIJKLMN**OP**QRSTUVWXYZ
1234567890
!\$%&()'*+ -.,/;= ? [] |
● □ — - ¼ ½ ¾ fi fl ff ffi ffl ° † ' ¢ ® ©

Times Italic

abcdefghijklmnopqrstuvwxy
*ABCDEFGHIJKLMN**OP**QRSTUVWXYZ*
1234567890
!\$%&()'+ -.,/;= ? [] |*
● □ — - ¼ ½ ¾ *fi fl ff ffi ffl* ° † ' ¢ ® ©

Times Bold

abcdefghijklmnopqrstuvwxy
ABCDEFGHIJKLMNOP**QRSTUVWXYZ**
1234567890
!\$%&()'*+ -.,/;= ? [] |
● ■ — - ¼ ½ ¾ **fi fl ff ffi ffl** ° † ' ¢ ® ©

Special Mathematical Font

" ` \ ^ _ ` ~ / < > { } # @ + - = *
 $\alpha \beta \gamma \delta \epsilon \zeta \eta \theta \iota \kappa \lambda \mu \nu \xi \omicron \pi \rho \sigma \tau \upsilon \phi \chi \psi \omega$
 $\Gamma \Delta \Theta \Lambda \Xi \Pi \Sigma \Upsilon \Phi \Psi \Omega$
 $\sqrt{\quad} \geq \leq \equiv \sim \cong \neq \rightarrow \leftarrow \uparrow \downarrow \times \div \pm \cup \cap \subset \supset \subseteq \supseteq \infty \partial$
§ ∇ ∫ α ∅ ∈ ‡ † ⊗ ⊕ ⊖ ⊗ ⊕ ⊖ ⊗ ⊕ ⊖

Table II

Input Naming Conventions for $\acute{\ } \grave{\ } _$, and $-$
 and for Non-ASCII Special Characters

Non-ASCII characters and *minus* on the standard fonts.


<i>Char</i>	<i>Input Name</i>	<i>Character Name</i>	<i>Char</i>	<i>Input Name</i>	<i>Character Name</i>
'		close quote	fi	\(fi	fi
'		open quote	fl	\(fl	fl
--	\(em	3/4 Em dash	ff	\(ff	ff
-	-	hyphen or	ffi	\(Fi	ffi
-	\(hy	hyphen	ffl	\(Fl	ffl
-	\(-	current font minus	°	\(de	degree
●	\(bu	bullet	†	\(dg	dagger
□	\(sq	square	'	\(fm	foot mark
-	\(ru	rule	¢	\(ct	cent sign
¼	\(14	1/4	®	\(rg	registered
½	\(12	1/2	©	\(co	copyright
¾	\(34	3/4			

Non-ASCII characters and $\acute{\ } \grave{\ } _$, $+$, $-$, $=$, and $*$ on the special font.

The ASCII characters @, #, ", ~, ^, <, >, \, {, }, ~, ~, and _ exist *only* on the special font and are printed as a 1-em space if that font is not mounted. The following characters exist only on the special font except for the upper case Greek letter names followed by † which are mapped into upper case English letters in whatever font is mounted on font position one (default Times Roman). The special math plus, minus, and equals are provided to insulate the appearance of equations from the choice of standard fonts.

<i>Char</i>	<i>Input Name</i>	<i>Character Name</i>	<i>Char</i>	<i>Input Name</i>	<i>Character Name</i>
+	\(pl	math plus	κ	\(*k	kappa
-	\(mi	math minus	λ	\(*l	lambda
=	\(eq	math equals	μ	\(*m	mu
*	\(**	math star	ν	\(*n	nu
§	\(sc	section	ξ	\(*c	xi
·	\(aa	acute accent	ο	\(*o	omicron
·	\(ga	grave accent	π	\(*p	pi
-	\(ul	underrule	ρ	\(*r	rho
/	\(sl	slash (matching backslash)	σ	\(*s	sigma
α	\(*a	alpha	ς	\(ts	terminal sigma
β	\(*b	beta	τ	\(*t	tau
γ	\(*g	gamma	υ	\(*u	upsilon
δ	\(*d	delta	φ	\(*f	phi
ε	\(*e	epsilon	χ	\(*x	chi
ζ	\(*z	zeta	ψ	\(*q	psi
η	\(*y	eta	ω	\(*w	omega
θ	\(*h	theta	Α	\(*A	Alpha†
ι	\(*i	iota	Β	\(*B	Beta†

<i>Char</i>	<i>Input Name</i>	<i>Character Name</i>
Γ	<code>\(*G</code>	Gamma
Δ	<code>\(*D</code>	Delta
E	<code>\(*E</code>	Epsilon†
Z	<code>\(*Z</code>	Zeta†
H	<code>\(*Y</code>	Eta†
Θ	<code>\(*H</code>	Theta
I	<code>\(*I</code>	Iota†
K	<code>\(*K</code>	Kappa†
Λ	<code>\(*L</code>	Lambda
M	<code>\(*M</code>	Mu†
N	<code>\(*N</code>	Nu†
Ξ	<code>\(*C</code>	Xi
O	<code>\(*O</code>	Omicron†
Π	<code>\(*P</code>	Pi
ρ	<code>\(*R</code>	Rho†
Σ	<code>\(*S</code>	Sigma
T	<code>\(*T</code>	Tau†
Υ	<code>\(*U</code>	Upsilon
Φ	<code>\(*F</code>	Phi
χ	<code>\(*X</code>	Chi†
Ψ	<code>\(*Q</code>	Psi
Ω	<code>\(*W</code>	Omega
$\sqrt{\quad}$	<code>\(sr</code>	square root
$\sqrt[n]{\quad}$	<code>\(rn</code>	root en extender
\cong	<code>\(>=</code>	$>=$
\leq	<code>\(<=</code>	$<=$
\equiv	<code>\(==</code>	identically equal
\approx	<code>\(^\sim=</code>	approx =
\sim	<code>\(ap</code>	approximates
\neq	<code>\(!=</code>	not equal
\rightarrow	<code>\(-></code>	right arrow
\leftarrow	<code>\(<-</code>	left arrow
\uparrow	<code>\(ua</code>	up arrow
\downarrow	<code>\(da</code>	down arrow
\times	<code>\(mu</code>	multiply
\div	<code>\(di</code>	divide
\pm	<code>\(+-</code>	plus-minus
\cup	<code>\(cu</code>	cup (union)
\cap	<code>\(ca</code>	cap (intersection)
\subset	<code>\(sb</code>	subset of
\supset	<code>\(sp</code>	superset of
\subseteq	<code>\(ib</code>	improper subset
\supseteq	<code>\(ip</code>	improper superset
∞	<code>\(if</code>	infinity
∂	<code>\(pd</code>	partial derivative
∇	<code>\(gr</code>	gradient
\neg	<code>\(no</code>	not
\int	<code>\(is</code>	integral sign
\propto	<code>\(pt</code>	proportional to
\emptyset	<code>\(es</code>	empty set
\in	<code>\(mo</code>	member of

<i>Char</i>	<i>Input Name</i>	<i>Character Name</i>
	<code>\(br</code>	box vertical rule
‡	<code>\(dd</code>	double dagger
R	<code>\(rh</code>	right hand
L	<code>\(lh</code>	left hand
	<code>\(bs</code>	Bell System logo
	<code>\(or</code>	or
○	<code>\(ci</code>	circle
{	<code>\(lt</code>	left top of big curly bracket
{	<code>\(lb</code>	left bottom
}	<code>\(rt</code>	right top
}	<code>\(rb</code>	right bot
{	<code>\(lk</code>	left center of big curly bracket
}	<code>\(rk</code>	right center of big curly bracket
	<code>\(bv</code>	bold vertical
	<code>\(lf</code>	left floor (left bottom of big square bracket)
	<code>\(rf</code>	right floor (right bottom)
	<code>\(lc</code>	left ceiling (left top)
	<code>\(rc</code>	right ceiling (right top)

Addendum to the
NROFF/TROFF User's Manual

☞ This addendum supercedes all previous addenda to this manual.

1. Command-Line Options

1.1 New options

- cname** Use the compacted version of macro package *name*, if it exists. If it does not, NROFF/TROFF will try the equivalent **-mname** option. The **-c** option should be used in preference to the **-m**, because it makes NROFF/TROFF execute significantly faster.
- kname** Produce a compacted macro package from this invocation of NROFF/TROFF. The compacted output is produced in files *d.name* and *t.name*. This option has no effect if a **co** request is not used in the NROFF/TROFF input. See Section 5 below.
- h** (NROFF only.) Use output tabs during horizontal spacing to speed up output as well as to reduce output byte count. Device tabs settings are assumed to be every 8 nominal character widths, as are the settings of input (logical) tabs.
- uN** (NROFF only.) Set the emboldening factor (number of character overstrikes in NROFF) for the third font position (bold) to be *N* (zero if *N* is missing); see the **bd** request in Section 2.2 below and the **.b** number register in Section 4 below. Note that it is *not* possible to turn off the emboldening in NROFF if the output device used is such that the overstriking is done locally by that device (e.g., DASI 300s).
- z** Suppress formatted output. Only diagnostics and messages from **tm** requests will occur.

1.2 Modified options

- Tname** There are additional defined device *names* for NROFF:
 - 2631** for the Hewlett-Packard 2631 printer in regular mode
 - 2631-c** for the Hewlett-Packard 2631 printer in compressed mode
 - 2631-e** for the Hewlett-Packard 2631 printer in expanded mode
 - 382** for the Anderson Jacobson 832 terminal
 - 4000a** for the Trendata 4000A terminal
 - 832** for the DCT832 terminal
 - X** for printers equipped with the TX print train
 - lp** for (generic) printers that can underline and tab

The **X** driving table includes special escape sequences for EBCDIC character codes that may be used by NROFF postprocessors.

In TROFF, the **-T** option may be used to specify the output device. The default TROFF output device (which is also the only device supported at the moment) is the Wang Laboratories' C/A/T phototypesetter. Other devices may be supported via this mechanism in future releases of TROFF.

- sN** As well as stopping the output every *N* pages, this option also causes the ASCII BEL character to be sent to the terminal when stopping between pages. In TROFF, the message "page stop" is printed on the diagnostic output (normally, the terminal).

2. Requests

2.1 New requests

- .ab** *text* Prints *text* on the diagnostic output (normally, the terminal) and terminates without further processing. If *text* is missing, the message "User Abort" is printed. This request does *not* cause a break. The output buffer is flushed.
- .co** If the **-kname** command-line option was given, compact the current state of NROFF/TROFF. If the **-kname** option was not used, **co** has no effect. See Section 5 below.
- .! cmd args** The UNIX command *cmd* is executed and its output is interpolated at this point. The standard input for *cmd* is closed.

2.2 Modified requests

- .ad** *c* The adjustment type indicator *c* may now also be a number obtained from the **.j** register; see Section 4 below.
- .bd** *F N* The emboldening request **bd** (q.v.) now also works in NROFF and causes overprinting of characters in the bold font. The default setting is **.bd 3 3**, specifying that characters on the font in position 3 (normally bold) are to be overstruck 3 times (i.e., printed in place a total of 4 times). The **-u** command-line option may be used to change the emboldening factor (i.e., the second argument of **bd**); see Section 1.1 above. This request may affect the contents of the number register **.b**; see Section 4 below. Note that it is *not* possible to turn off the emboldening in NROFF if the output device used is such that the overstriking is done locally by that device (e.g., DASI 300s).
- .so** *file* The contents of *file* will be interpolated at the point the **so** request is encountered. Previously, if an **so** was encountered inside a macro, the interpolation was delayed until the input level returned to the file level (i.e., at least until the end of the macro).

3. New Escape Sequences

- \gx,\g(xx)** Return the format of register *x* or *xx*; return nothing if *x* (*xx*) has not yet been referenced. Can be saved and used later as the second argument of the **af** request (q.v.) to restore the previous format of a register.
- \jx,\j(xx)** Mark in register *x* or *xx* the current horizontal position on the *output* line; see also the **\k** register described in §11.3 of this manual.

4. New Predefined Number Registers

- .F** Read-only. The value is a *string* that is the name of the current input file.
- .L** Read-only. Contains the current line-spacing parameter, i.e., the value of the argument of the most recent **ls** request.
- .P** Read-only. Contains the value 1 if the current page is being printed, and 0 otherwise, e.g., if the current page does *not* appear in the **-o** option list.
- .R** Count of number registers that remain available for use.
- .b** Emboldening factor of the current font (NROFF and TROFF); see the **-u** option in Section 1.1 above and the **bd** request in Section 2.2 above.
- .j** Read-only. Indicates the current adjustment mode and type. Can be saved and used later as the argument to the **ad** request (q.v.) to restore a previous adjustment mode.

- .k Read-only. Contains the horizontal size of the text portion (*not* including the size of the current indent, if any) of the current, partially-collected *output* line, if any, in the current environment.
- c. Provides general register access to the input line-number in the current input file. Contains the same value as the read-only .c register.

5. Compacted Macros

5.1 *User information.* The time required by NROFF/TROFF to read in a macro package may be greatly lessened by using a pre-processed or *compacted* version of that package. The compacted version of a macro package is completely equivalent to the non-compacted version, except that a compacted macro package can *not* be read in by the *so* request.

A compacted version of a macro package is obtained by the *-cname* command-line option, while the *-mname* option obtains the uncompact version; see Section 1.1 above. Because *-cname* reverts to *-mname* if the named macro package has not been compacted, one should normally use *-c* rather than *-m*.

5.2 *Building a compacted macro package.* If one has a macro package and wishes to make a compacted version of it, the following steps should be followed:

1. Separate the compactable part from the non-compactable part:

Only the following can be compacted: macro, string, and diversion definitions; number register definitions and values; environment settings; and trap settings. For example, the following are *not* compactable: end macro (*em*) requests and any commands that may interact with command-line settings (e.g., references, in the MM macro package, to the number register *P*, which can be set from the command line).

All the non-compactable material must be placed at the end of the macro package, with a *co* request separating the compactable from non-compactable parts:

```
  . Compactable part
    :
    :
  .co
  Non-compactable part
    :
    :
```

The *co* request indicates to NROFF/TROFF when to compact its internal state.

2. Produce compacted files:

Once compactable and non-compactable segments have been set up as above, NROFF/TROFF may be run with the *-kname* option to build the compacted files; see Section 1.1 above.

For example, if the macro file produced by Step 1 above is called *macs*, then the following may be used to build the compacted files:

```
  nroff -kmacs macs          or
  troff -kmacs macs
```

Each of these commands causes NROFF/TROFF to create two files in the current directory, *d.macs* and *t.macs*.

3. Install compacted files:

The two compacted files produced in Step 2 must be installed into the system macro library (*/usr/lib/macros*) with appropriate names: prepend *cmp.n.* to files produced by NROFF, and *cmp.t.* to files produced by TROFF:

```
cp d.macs /usr/lib/macros/cmp.n.d.macs
cp t.macs /usr/lib/macros/cmp.n.t.macs
```

will install the two files produced by compacting **macs** with NROFF.

4. Install non-compactable segment:

The non-compactable segment from the original macro package must also be installed on the system as:

```
/usr/lib/macros/ucmp.[nt].name
```

where **n** of **[nt]** indicates the NROFF version, while **t** indicates the TROFF version.

The non-compactable segment must be produced "by hand," e.g., by using the editor:

```
ed macs
/^\.co$/+1,$w /usr/lib/macros/ucmp.n.macs
Q
```

would create the (NROFF) non-compactable segment. Note that the non-compactable segment *must* exist even if it is empty (i.e., if the entire macro package is compactable).

Thus, once the macro package **macs** is compacted by both NROFF and TROFF, and the resulting files installed, the directory **/usr/lib/macros** will contain the following six files:

```
cmp.[nt].d.macs
cmp.[nt].t.macs
ucmp.[nt].macs
```

where the first **t** applies to the macros as compacted by TROFF, while **n** indicates the NROFF macros. (The **d** and the second **t** in the above names stand for "data" and "text," respectively.)

5.3 Warnings. A compacted macro package depends heavily on the particular version of NROFF/TROFF that produced it. This means that each package needs to be compacted *separately* by both NROFF and TROFF. It also means that *all* compacted macro packages must be recompactd whenever a new version of NROFF or TROFF is installed.

If NROFF/TROFF discovers that a macro package was produced by a different version of NROFF/TROFF than that attempting to read it, the **-c** option is abandoned and the corresponding **-m** option is attempted instead.

Should NROFF/TROFF actually read a compacted package that was produced by a different version of NROFF/TROFF (e.g., because the version number of NROFF/TROFF was *not* updated, but the code was changed), very peculiar behavior will result.

Finally, note that the existence of a compacted macro package in no way precludes the installation of the same package in non-compactd form, as explained on page 1 of this manual.

6. Other Important Changes

1. NROFF/TROFF can accept several **-m/-c** options on the command line, causing *all* macro packages thus named to be read in turn.
2. The conditionally accepted part of an **ie** or **if** request is now *completely ignored* if the test fails, rather than being read in copy mode.
3. The **cu** request has been improved to provide up to about three lines of continuously underlined text; the underlining is *not* lost when **cu** is used inside a diversion.

MM—Memorandum Macros

D. W. Smith
J. R. Mashey
E. C. Pariser (January 1980 Revision)
N. W. Smith (June 1980 Revision)

Bell Laboratories
Piscataway, New Jersey 08854

1. INTRODUCTION

1.1 Purpose

This memorandum is the user's guide and reference manual for the Memorandum Macros (MM), a general-purpose package of text formatting macros for use with the UNIX† text formatters *nroff* and *troff*.

The purpose of MM is to provide a unified, consistent, and flexible tool for producing many common types of documents. Although the UNIX time-sharing system provides other macro packages for various *specialized* formats, MM has become the standard, general-purpose macro package for most documents.

MM can be used to produce:

- Letters
- Reports
- Technical Memoranda
- Released Papers
- Manuals
- Books.

The uses of MM range from single-page letters to documents of several hundred pages in length, such as user guides, design proposals, etc.

1.2 Conventions

Each section of this memorandum explains a single facility of MM. In general, the earlier a section occurs, the more necessary it is for most users. Some of the later sections can be completely ignored if MM defaults are acceptable. Likewise, each section progresses from normal-case to special-case facilities. We recommend reading a section in detail only until there is enough information to obtain the desired format, then skimming the rest of it, because some details may be of use to just a few people.

Numbers enclosed in curly brackets ({}) refer to section numbers within this document. For example, this is {1.2}.

Sections that require knowledge of the formatters {1.4} have a bullet (●) at the end of their headings.

In the synopses of macro calls, square brackets ([]) surrounding an argument indicate that it is optional. Ellipses (...) show that the preceding argument may appear more than once.

A reference of the form *name*(*N*) points to page *name* in section *N* of the *UNIX User's Manual*^[1].

The examples of *output* in this manual are as produced by *troff*; *nroff* output would, of course, look somewhat different (Appendix C shows both *nroff* and *troff* outputs for a simple letter). In those cases in which the behavior of the two formatters is truly different, the *nroff* action is described first, with the *troff* action following in parentheses. For example:

† UNIX is a trademark of Bell Laboratories.

The title is underlined (italic).

means that the title is underlined in *nroff* and italic in *troff*.

1.3 Overall Structure of a Document

The input for a document that is to be formatted with MM possesses four major segments, any of which may be omitted; if present, they *must* occur in the following order:

- *Parameter-setting*— This segment sets the general style and appearance of a document. The user can control page width, margin justification, numbering styles for headings and lists, page headers and footers {9}, and many other properties of the document. Also, the user can add macros or redefine existing ones. This segment can be omitted entirely if one is satisfied with default values; it produces no actual output, but only performs the setup for the rest of the document.
- *Beginning*— This segment includes those items that occur only once, at the beginning of a document, e.g., title, author's name, date.
- *Body*— This segment is the actual text of the document. It may be as small as a single paragraph, or as large as hundreds of pages. It may have a hierarchy of *headings* up to seven levels deep {4}. Headings are automatically numbered (if desired) and can be saved to generate the table of contents. Five additional levels of subordination are provided by a set of *list* macros for automatic numbering, alphabetic sequencing, and "marking" of list items {5}. The body may also contain various types of displays, tables, figures, references, and footnotes {7, 8, 11}.
- *Ending*— This segment contains those items that occur once only, at the end of a document. Included here are signature(s) and lists of notations (e.g., "copy to" lists) {6.11}. Certain macros may be invoked here to print information that is wholly or partially derived from the rest of the document, such as the table of contents or the cover sheet for a document {10}.

The existence and size of these four segments varies widely among different document types. Although a specific item (such as date, title, author name(s), etc.) may be printed in several different ways depending on the document type, there is a uniform way of typing it in.

1.4 Definitions

The term *formatter* refers to either of the text-formatting programs *nroff* and *troff*.

Requests are built-in commands recognized by the formatters. Although one seldom needs to use these requests directly {3.10}, this document contains references to some of them. Full details are given in the *NROFF/TROFF User's Manual*^[2]. For example, the request:

```
.sp
```

inserts a blank line in the output.

Macros are named collections of requests. Each macro is an abbreviation for a collection of requests that would otherwise require repetition. MM supplies many macros, and the user can define additional ones. Macros and requests share the same set of names and are used in the same way.

Strings provide character variables, each of which names a string of characters. Strings are often used in page headers, page footers, and lists. They share the pool of names used by *requests* and *macros*. A string can be given a value via the *.ds* (define string) request, and its value can be obtained by referencing its name, preceded by "*" (for 1-character names) or "*(" (for 2-character names). For instance, the string *DT* in MM normally contains the current date, so that the *input* line:

```
Today is \*(DT.
```

may result in the following *output*:

```
Today is March 27, 1981.
```

The current date can be replaced, e.g.:

```
.ds DT 01/01/79
```

or by invoking a macro designed for that purpose {6.7.1}.

Number registers fill the role of integer variables. They are used for flags, for arithmetic, and for automatic numbering. A register can be given a value using a `.nr` request, and be referenced by preceding its name by “\n” (for 1-character names) or “\n(” (for 2-character names). For example, the following sets the value of the register *d* to 1 more than that of the register *dd*:

```
.nr d 1+\n(dd
```

See {14.1} regarding naming conventions for requests, macros, strings, and number registers. Appendix E list all macros, strings, and number registers defined in MM.

1.5 Prerequisites and Further Reading

1.5.1 Prerequisites. We assume familiarity with UNIX at the level given in *UNIX for Beginners*^[3] and *A Tutorial Introduction to the UNIX Text Editor*^[4]. Some familiarity with the request summary in the *NROFF/TROFF User's Manual*^[2] is helpful.

1.5.2 Further Reading. *NROFF/TROFF User's Manual*^[2] provides detailed descriptions of formatter capabilities, while *A TROFF Tutorial*^[5] provides a general overview. See *Typesetting Mathematics—User's Guide*^[6] for instructions on formatting mathematical expressions. See *tbl(1)* and *TBL—A Program to Format Tables*^[7] for instructions on formatting tabular data.

Examples of formatted documents and of their respective input, as well as a quick reference to the material in this manual are given in *Typing Documents with MM*^[8].

2. INVOKING THE MACROS

This section tells how to access MM, shows UNIX command lines appropriate for various output devices, and describes command-line flags for MM.

2.1 The mm Command

The *mm(1)* command can be used to print documents using *nroff* and MM; this command invokes *nroff* with the `-cm` flag {2.2}. It has options to specify preprocessing by *tbl* and/or by *neqn* and for postprocessing by various output filters. Any arguments or flags that are not recognized by *mm(1)*, e.g. `-rC3`, are passed to *nroff* or to MM, as appropriate. The options, which can occur in any order but *must* appear before the file names, are:

- e *neqn* is to be invoked; also causes *neqn* to read `/usr/pub/eqnchar` (see *eqnchar(7)*).
- t *tbl(1)* is to be invoked.
- c *col(1)* is to be invoked.
- E the `-e` option of *nroff* is to be invoked.
- y `-mm` (uncompacted macros) is to be used instead of `-cm`.
- 12 12-pitch mode is to be used. Be sure the pitch switch on the terminal is set to 12.
- T450 output is to a DASI 450. This is the *default* terminal type (unless \$TERM is set). It is also equivalent to `-T1620`.
- T450—12 output is to a DASI 450 in 12-pitch mode.
- T300 output is to a DASI 300 terminal.
- T300—12 output is to a DASI 300 in 12-pitch mode.
- T300s output is to a DASI 300S.
- T300s—12 output is to a DASI 300S in 12-pitch mode.
- T4014 output is to a Tektronix 4014.
- T37 output is to a TELETYPE® Model 37.
- T382 output is to a DTC-382.
- T4000a output is to a Trendata 4000A.

- TX output is prepared for an EBCDIC line printer.
- Thp output is to a HP264x (implies -c).
- T43 output is to a *TELETYPE* Model 43 (implies -c).
- T40/4 output is to a *TELETYPE* Model 40/4 (implies -c).
- T745 output is to a Texas Instrument 700 series terminal (implies -c).
- T2631 output is prepared for a HP2631 printer (where -T2631-e and -T2631-c may be used for expanded and compressed modes, respectively) (implies -c).
- Tlp output is to a device with no reverse or partial line motions or other special features (implies -c).

Any other -T option given does not produce an error; it is equivalent to -Tlp.

A similar command is available for use with *troff* (see *mmt*(1)).

2.2 The -cm or -mm Flag

The MM package can also be invoked by including the -cm or -mm flag as an argument to the formatter. The -cm flag causes the pre-compacted version of the macros to be loaded. The -mm flag causes the file /usr/lib/tmac/tmac.m to be read and processed before any other files. This action defines the MM macros, sets default values for various parameters, and initializes the formatter to be ready to process the files of input text.

2.3 Typical Command Lines

The prototype command lines are as follows (with the various options explained in {2.4} and in the *NROFF/TROFF User's Manual*^[2]).

- Text without tables or equations:

```
mm [options] file-name ...
or nroff [options] -cm file-name ...
mmt [options] file-name ...
or troff [options] -cm file-name ...
```

- Text with tables:

```
mm -t [options] file-name ...
or tbl file-name ... | nroff [options] -cm
mmt -t [options] file-name ...
or tbl file-name ... | troff [options] -cm
```

- Text with equations:

```
mm -e [options] file-name ...
or neqn /usr/pub/eqnchar file-name ... | nroff [options] -cm
mmt -e [options] file-name ...
or eqn /usr/pub/eqnchar file-name ... | troff [options] -cm
```

- Text with both tables and equations:

```
mm -t -e [options] file-name ...
or tbl file-name ... | neqn /usr/pub/eqnchar - | nroff [options] -cm
mmt -t -e [options] file-name ...
or tbl file-name ... | eqn /usr/pub/eqnchar - | troff [options] -cm
```

When formatting a document with *nroff*, the output should normally be processed for a specific type of terminal, because the output may require some features that are specific to a given terminal, e.g., reverse paper motion or half-line paper motion in both directions. Some commonly-used terminal types and the command lines appropriate for them are given below. See {2.4} as well as *300*(1), *450*(1), *4014*(1), *hp*(1), *col*(1), and *terminals*(7) for further information.

- DASI 450 in 10-pitch, 6 lines/inch mode, with 0.75 inch offset, and a line length of 6 inches (60 characters) where this is the default terminal type so no `-T` option is needed (unless `$TERM` is set to another value):

```
mm file-name ...
or nroff -T450 -h -cm file-name ...
```

- DASI 450 in 12-pitch, 6 lines/inch mode, with 0.75 inch offset, and a line length of 6 inches (72 characters):

```
mm -12 file-name ...
or nroff -T450-12 -h -cm file-name ...
```

or, to increase the line length to 80 characters and decrease the offset to 3 characters:

```
mm -12 -rW80 -rO3 file-name ...
or nroff -T450-12 -rW80 -rO3 -h -cm file-name ...
```

- Hewlett-Packard HP264x CRT family:

```
mm -Thp file-name ...
or nroff -cm file-name ... | col | hp
```

- Any terminal incapable of reverse paper motion and also lacking hardware tab stops (Texas Instruments 700 series, etc.):

```
mm -T745 file-name ...
or nroff -cm file-name ... | col -x
```

- Versatec printer (see `vp(1)` for additional details):

```
vp [vp-options] "mm -rT2 -c file-name ..."
or vp [vp-options] "nroff -rT2 -cm file-name ... | col"
```

Of course, `tbl(1)` and `eqn(1)/neqn`, if needed, must be invoked as shown in the command line prototypes at the beginning of this section.

If two-column processing {12.4} is used with `nroff`, either the `-c` option must be specified to `mm(1)`,¹ or the `nroff` output must be postprocessed by `col(1)`. In the latter case, the `-T37` terminal type must be specified to `nroff`, the `-h` option must *not* be specified, and the output of `col(1)` must be processed by the appropriate terminal filter (e.g., `450(1)`); `mm(1)` with the `-c` option handles all this automatically.

2.4 Parameters that Can Be Set from the Command Line

Number registers are commonly used within MM to hold parameter values that control various aspects of output style. Many of these can be changed within the text files via `.nr` requests. In addition, some of these registers can be set from the command line itself, a useful feature for those parameters that should *not* be permanently embedded within the input text itself. If used, these registers (with the possible exception of the register *P*—see below) *must* be set on the command line (or before the MM macro definitions are processed) and their meanings are:

`-rAn` for $n = 1$ has the effect of invoking the `.AF` macro without an argument {6.7.2}. If $n = 2$ allows for usage of the Bell System logo, if available, on a printing device (currently available for the Xerox 9700 only).

1. Note that `mm(1)` uses `col(1)` automatically for many of the terminal types {2.1}.

- rCn *n* sets the type of copy (e.g., DRAFT) to be printed at the bottom of each page. See {9.5}.
 - n* = 1 for OFFICIAL FILE COPY.
 - n* = 2 for DATE FILE COPY.
 - n* = 3 for DRAFT with single-spacing and default paragraph style.
 - n* = 4 for DRAFT with double-spacing and 10 space paragraph indent.
- rD1 sets *debug mode*. This flag requests the formatter to attempt to continue processing even if MM detects errors that would otherwise cause termination. It also includes some debugging information in the default page header {9.2, 12.3}.
- rEn controls the font of the Subject/Date/From fields. If *n* is 1 then these fields are bold (default for *troff*) and if *n* is 0 then these fields are roman (regular text—default for *nroff*).
- rLk sets the length of the physical page to *k* lines.² The default value is 66 lines per page. This parameter is used, for example, when directing output to a Versatec printer.
- rNn specifies the page numbering style. When *n* is 0 (default), all pages get the (prevailing) header {9.2}. When *n* is 1, the page header replaces the footer on page 1 only. When *n* is 2, the page header is omitted from page 1. When *n* is 3, “section-page” numbering {4.5} occurs (see .FD {8.3} and .RP {11.4} for footnote and reference numbering in sections). When *n* is 4, the *default* page header is suppressed; however a user-specified header is not affected. When *n* is 5, “section-page” and “section-figure” numbering occurs.

<i>n</i>	Page 1	Pages 2ff.
0	header	header
1	header replaces footer	header
2	no header	header
3	“section-page” as footer	same as page 1
4	no header	no header unless .PH defined
5	same as 3—with “section-figure”	same as page 1

The contents of the prevailing header and footer do *not* depend on the value of the number register *N*; *N* only controls whether and where the header (and, for *N*=3 or 5, the footer) is printed, as well as the page numbering style. In particular, if the header and footer are null {9.2, 9.5}, the value of *N* is irrelevant.

- rOk offsets output *k* spaces to the right.² It is helpful for adjusting output positioning on some terminals. The default offset if this register is not set on the command line is 0.75 inches. NOTE: The register name is the capital letter “O”, *not* the digit zero (0).
- rPn specifies that the pages of the document are to be numbered starting with *n*. This register may also be set via a .nr request in the input text.
- rSn sets the point size and vertical spacing for the document. The default *n* is 10, i.e., 10-point type on 12-point vertical spacing, giving 6 lines per inch {12.9}. This parameter applies to *troff* only.
- rTn provides register settings for certain devices. If *n* is 1, then the line length and page offset are set to 80 and 3, respectively. Setting *n* to 2 changes the page length to 84 lines per page and inhibits underlining; it is meant for output sent to the Versatec printer. The default value for *n* is 0. This parameter applies to *nroff* only.
- rU1 controls underlining of section headings. This flag causes only letters and digits to be underlined. Otherwise, all characters (including spaces) are underlined {4.2.2.4.2}. This parameter applies to *nroff* only.

2. For *nroff*, *k* is an *unscaled* number representing lines or character positions; for *troff*, *k* must be *scaled*.

— `rWk` page width (i.e., line length and title length) is set to k .² This can be used to change the page width from the default value of 6 inches (60 characters in 10 pitch or 72 characters in 12 pitch).

2.5 Omission of `-cm` or `-mm`

If a large number of arguments is required on the command line, it may be convenient to set up the first (or only) input file of a document as follows:

```
zero or more initializations of registers listed in {2.4}
.so /usr/lib/tmac/tmac.m
remainder of text
```

In this case, one must *not* use the `-cm` or `-mm` flag (nor the `mm(1)` or `mmt(1)` command); the `.so` request has the equivalent effect, but the registers in {2.4} must be initialized *before* the `.so` request, because their values are meaningful only if set before the macro definitions are processed. When using this method, it is best to “lock” into the input file only those parameters that are seldom changed. For example:

```
.nr W 80
.nr O 10
.nr N 3
.so /usr/lib/tmac/tmac.m
.H 1 "INTRODUCTION"
⋮
⋮
```

specifies, for `nroff`, a line length of 80, a page offset of 10, and “section-page” numbering.

3. FORMATTING CONCEPTS

3.1 Basic Terms

The normal action of the formatters is to *fill* output lines from one or more input lines. The output lines may be *justified* so that both the left and right margins are aligned. As the lines are being filled, words may also be hyphenated {3.4} as necessary. It is possible to turn any of these modes on and off (see `.SA` {12.2}, `Hy` {3.4}, and the formatter `.nf` and `.fi` requests^[2]). Turning off fill mode also turns off justification and hyphenation.

Certain formatting commands (requests and macros) cause the filling of the current output line to cease, the line (of whatever length) to be printed, and the subsequent text to begin a new output line. This printing of a partially filled output line is known as a *break*. A few formatter requests and most of the MM macros cause a break.

While formatter requests can be used with MM, one must fully understand the consequences and side-effects that each such request might have. Actually, there is little need to use formatter requests; the macros described here should be used in most cases because:

- it is much easier to control (and change at any later point in time) the overall style of the document.
- complicated features (such as footnotes or tables of contents) can be obtained with ease.
- the user is insulated from the peculiarities of the formatter language.

A good rule is to use formatter requests only when absolutely necessary {3.10}.

In order to make it easy to revise the input text at a later time, input lines should be kept short and should be broken at the end of clauses; each new full *sentence must* begin on a new line.

3.2 Arguments and Double Quotes

For any macro call, a *null argument* is an argument whose width is zero. Such an argument often has a special meaning; the preferred form for a null argument is `"`. Note that *omitting* an argument is *not* the same as supplying a *null argument* (for example, see the `.MT` macro in {6.6}). Furthermore, omitted arguments can occur only at the end of an argument list, while null arguments can occur anywhere.

Any macro argument containing ordinary (paddable) spaces *must* be enclosed in double quotes ("").³ Otherwise, it will be treated as several separate arguments.

Double quotes (") are *not* permitted as part of the value of a macro argument or of a string that is to be used as a macro argument. If you must, use two grave accents (` `) and/or two acute accents (´ ´) instead. This restriction is necessary because many macro arguments are processed (interpreted) a variable number of times; for example, headings are first printed in the text and may be (re)printed in the table of contents.

3.3 Unpaddable Spaces

When output lines are *justified* to give an even right margin, existing spaces in a line may have additional spaces appended to them. This may harm the desired alignment of text. To avoid this problem, it is necessary to be able to specify a space that cannot be expanded during justification, i.e., an *unpaddable space*. There are several ways to accomplish this.

First, one may type a backslash followed by a space (" \ "). This pair of characters directly generates an *unpaddable space*. Second, one may sacrifice some seldom-used character to be translated into a space upon output. Because this translation occurs after justification, the chosen character may be used anywhere an unpaddable space is desired. The tilde (~) is often used for this purpose. To use it in this way, insert the following at the beginning of the document:

```
.tr ~
```

If a tilde must actually appear in the output, it can be temporarily "recovered" by inserting:

```
.tr ~^
```

before the place where it is needed. Its previous usage is restored by repeating the ".tr ~^", but only after a break or after the line containing the tilde has been forced out. Note that the use of the tilde in this fashion is *not* recommended for documents in which the tilde is used within equations.

3.4 Hyphenation

The formatters do not perform hyphenation unless the user requests it. Hyphenation can be turned on in the body of the text by specifying:

```
.nr Hy 1
```

once at the beginning of the document. For hyphenation within footnotes and across pages, see {8.3}.

If hyphenation is requested, the formatters will automatically hyphenate words, if need be. However, the user may specify the hyphenation points for a specific occurrence of any word by the use of a special character known as a hyphenation indicator, or may specify hyphenation points for a small list of words (about 128 characters).

If the *hyphenation indicator* (initially, the two-character sequence "\\%") appears at the beginning of a word, the word is *not* hyphenated. Alternatively, it can be used to indicate legal hyphenation point(s) inside a word. In any case, *all* occurrences of the hyphenation indicator disappear on output.

The user may specify a different hyphenation indicator:

```
.HC [hyphenation-indicator]
```

The circumflex (^) is often used for this purpose; this is done by inserting the following at the beginning of a document:

3. A double quote (") is a *single* character that must not be confused with two apostrophes or acute accents (´ ´), or with two grave accents (` `).

.HC

Note that any word containing hyphens or dashes—also known as *em* dashes—will be hyphenated immediately after a hyphen or dash if it is necessary to hyphenate the word, *even if the formatter hyphenation function is turned off*.

The user may supply, via the .hw request, a small list of words with the proper hyphenation points indicated. For example, to indicate the proper hyphenation of the word “printout,” one may specify:

```
.hw print-out
```

3.5 Tabs

The macros .MT {6.6}, .TC {10.1}, and .CS {10.2} use the formatter .ta request to set tab stops, and then restore the *default* values⁴ of tab settings. Thus, setting tabs to other than the default values is the user’s responsibility.

Note that a tab character is always interpreted with respect to its position on the *input line*, rather than its position on the output line. In general, tab characters should appear only on lines processed in “no-fill” mode {3.1}.

Also note that *tbl*(1) {7.3} changes tab stops, but does *not* restore the default tab settings.

3.6 Special Use of the BEL Character

The non-printing character BEL is used as a delimiter in many macros where it is necessary to compute the width of an argument or to delimit arbitrary text, e.g., in headers and footers {9}, headings {4}, and list marks {5}. Users who include BEL characters in their input text (especially in arguments to macros) will receive mangled output.

3.7 Bullets

A bullet (●) is often obtained on a typewriter terminal by using an “o” overstruck by a “+”. For compatibility with *troff*, a bullet string is provided by MM. Rather than overstriking, use the sequence:

```
\*(BU
```

wherever a bullet is desired. Note that the bullet list (.BL) macros {5.3.3.2} use this string to automatically generate the bullets for the list items.

3.8 Dashes, Minus Signs, and Hyphens

Troff has distinct graphics for a dash, a minus sign, and a hyphen, while *nroff* does not. Those who intend to use *nroff* only may use the minus sign (“-”) for all three.

Those who wish mainly to use *troff* should follow the escape conventions of the *NROFF/TROFF User’s Manual*^[2].

Those who want to use both formatters must take care during text preparation. Unfortunately, these characters cannot be represented in a way that is both compatible and convenient. We suggest the following approach:

- | | |
|--------|---|
| Dash | Type *(EM for each text dash for both <i>nroff</i> and <i>troff</i> . This string generates an em dash (—) in <i>troff</i> and “-” in <i>nroff</i> . Note that the dash list (.DL) macros {5.3.3.3} automatically generate the em dash for each list item. |
| Hyphen | Type “-” and use as is for both formatters. <i>Nroff</i> will print it as is, and <i>troff</i> will print “-” (a true hyphen). |

4. Every eight characters in *nroff*; every ½ inch in *troff*.

Minus Type “\—” for a true minus sign, regardless of formatter. *Nroff* will effectively ignore the “\”, while *troff* will print a true minus sign.

3.9 Trademark String

A trademark string *(Tm is available with MM. This places the letters “TM” one-half line above the text that it follows.

For example:

The UNIX*(Tm User’s Manual is available from the library.

yields:

The UNIX™ User’s Manual is available from the library.

3.10 Use of Formatter Requests •

Most formatter requests^[2] should *not* be used with MM because MM provides the corresponding formatting functions in a much more user-oriented and surprise-free fashion than do the basic formatter requests {3.1}. However, some formatter requests *are* useful with MM, namely:

```
.af .br .ce .de .ds .fi .hw .ls .nf .nr .nx
.rm .rr .rs .so .sp .ta .ti .tl .tr .!
```

The .fp, .lg, and .ss requests are also sometimes useful for *troff*. Use of other requests without fully understanding their implications very often leads to disaster.

4. PARAGRAPHS AND HEADINGS

This section describes simple paragraphs and section headings. Additional paragraph and list styles are covered in {5}.

4.1 Paragraphs

```
.P [type]
one or more lines of text.
```

This macro is used to begin two kinds of paragraphs. In a *left-justified* paragraph, the first line begins at the left margin, while in an *indented* paragraph, it is indented five spaces (see below).

A document possesses a *default paragraph style* obtained by specifying “.P” before each paragraph that does *not* follow a heading {4.2}. The default style is controlled by the register *Pt*. The initial value of *Pt* is 0, which always provides left-justified paragraphs. All paragraphs can be forced to be indented by inserting the following at the beginning of the document:

```
.nr Pt 1
```

All paragraphs will be indented except after headings, lists, and displays if the following:

```
.nr Pt 2
```

is inserted at the beginning of the document.

The amount a paragraph is indented is contained in the register *Pi*, whose default value is 5. To indent paragraphs by, say, 10 spaces, insert:

```
.nr Pi 10
```

at the beginning of the document. Of course, both the *Pi* and *Pt* register values must be greater than zero for any paragraphs to be indented.

The number register *Ps* controls the amount of spacing between paragraphs. By default, *Ps* is set to 1, yielding one blank space (½ a vertical space).

☞ *Values that specify indentation must be unscaled and are treated as "character positions," i.e., as a number of ens. In troff, an en is the number of points (1 point = 1/72 of an inch) equal to half the current point size. In nroff, an en is equal to the width of a character.*

Regardless of the value of *Pt*, an *individual* paragraph can be forced to be left-justified or indented. ".P 0" always forces left justification; ".P 1" always causes indentation by the amount specified by the register *Pi*.

If .P occurs inside a *list*, the indent (if any) of the paragraph is added to the current list indent {5}.

Numbered paragraphs may be produced by setting the register *Np* to 1. This produces paragraphs numbered within first level headings, e.g., 1.01, 1.02, 1.03, 2.01, etc.

A different style of numbered paragraphs is obtained by using the

```
.nP
```

macro rather than the .P macro for paragraphs. This produces paragraphs that are numbered within second level headings and contain a "double-line indent" in which the text of the second line is indented to be aligned with the text of the first line so that the number stands out.

```
.H 1 "FIRST HEADING"
.H 2 "Second Heading"
.nP
one or more lines of text
```

4.2 Numbered Headings

```
.H level [heading-text] [heading-suffix]
zero or more lines of text
```

The .H macro provides seven levels of numbered headings, as illustrated by this document. Level 1 is the most major or highest; level 7 the lowest.

The *heading-suffix* is appended to the *heading-text* and may be used for footnote marks which should not appear with the heading text in the Table of Contents.

☞ *Strictly speaking, there is no need for a .P macro immediately after a .H (or .HU {4.3}), because the .H macro also performs the function of the .P macro, and an immediately following .P is ignored {4.2.2.2}. It is, however, good practice to start every paragraph with a .P macro, thereby ensuring that all paragraphs uniformly begin with a .P throughout an entire document.*

4.2.1 Normal Appearance. The normal appearance of headings is as shown in this document. The effect of .H varies according to the *level* argument. First-level headings are *preceded* by two blank lines (one vertical space); all others are *preceded* by one blank line (½ a vertical space).

.H 1 heading-text gives a bold heading *followed* by a single blank line (½ a vertical space). The following text begins on a new line and is indented according to the current paragraph type. Full capital letters should normally be used to make the heading stand out.

.H 2 heading-text yields a bold heading followed by a single blank line (½ a vertical space). The following text begins on a new line and is indented according to the current paragraph type. Normally, initial capitals are used.

.H *n* heading-text for $3 \leq n \leq 7$, produces an underlined (italic) heading followed by two spaces. The following text appears on the same line, i.e., these are *run-in* headings.

Appropriate numbering and spacing (horizontal and vertical) occur even if the heading text is omitted from a .H macro call.

Here are the first few .H calls of {4}:

```
.H 1 "PARAGRAPHS AND HEADINGS"
.H 2 "Paragraphs"
.H 2 "Numbered Headings"
.H 3 "Normal Appearance."
.H 3 "Altering Appearance of Headings."
.H 4 "Pre-Spacing and Page Ejection."
.H 4 "Spacing After Headings."
.H 4 "Centered Headings."
.H 4 "Bold, Italic, and Underlined Headings"
.H 5 "Control by Level."
```

4.2.2 Altering Appearance of Headings. Users satisfied with the default appearance of headings may skip to {4.3}. One can modify the appearance of headings quite easily by setting certain registers and strings at the beginning of the document. This permits quick alteration of a document's style, because this style-control information is concentrated in a few lines, rather than being distributed throughout the document.

4.2.2.1 Pre-Spacing and Page Ejection. A first-level heading normally has two blank lines (one vertical space) preceding it, and all others have one blank line ($\frac{1}{2}$ a vertical space). If a multi-line heading were to be split across pages, it is automatically moved to the top of the next page. Every first-level heading may be forced to the top of a new page by inserting:

```
.nr Ej 1
```

at the beginning of the document. Long documents may be made more manageable if each section starts on a new page. Setting *Ej* to a higher value causes the same effect for headings up to that level, i.e., a page eject occurs if the heading level is less than or equal to *Ej*.

4.2.2.2 Spacing After Headings. Three registers control the appearance of text immediately following a .H call. They are *Hb* (heading break level), *Hs* (heading space level), and *Hi* (post-heading indent).

If the heading level is less than or equal to *Hb*, a break {3.1} occurs after the heading. If the heading level is less than or equal to *Hs*, a blank line ($\frac{1}{2}$ a vertical space) is inserted after the heading. Defaults for *Hb* and *Hs* are 2. If a heading level is greater than *Hb* and also greater than *Hs*, then the heading (if any) is run into the following text. These registers permit headings to be separated from the text in a consistent way throughout a document, while allowing easy alteration of white space and heading emphasis.

For any *stand-alone* heading, i.e., a heading not run into the following text, the alignment of the next line of output is controlled by the register *Hi*. If *Hi* is 0, text is left-justified. If *Hi* is 1 (the *default* value), the text is indented according to the paragraph type as specified by the register *Pt* {4.1}. Finally, if *Hi* is 2, text is indented to line up with the first word of the heading itself, so that the heading number stands out more clearly.

For example, to cause a blank line ($\frac{1}{2}$ a vertical space) to appear after the first three heading levels, to have no run-in headings, and to force the text following all headings to be left-justified (regardless of the value of *Pt*), the following should appear at the top of the document:

```
.nr Hs 3
.nr Hb 7
.nr Hi 0
```

4.2.2.3 Centered Headings. The register *Hc* can be used to obtain centered headings. A heading is centered if its level is less than or equal to *Hc*, and if it is also stand-alone {4.2.2.2}. *Hc* is 0 initially (no centered headings).

4.2.2.4 *Bold, Italic, and Underlined Headings*

4.2.2.4.1 *Control by Level.* Any heading that is underlined by *nroff* is made italic by *troff*. The string *HF* (heading font) contains seven codes that specify the fonts for heading levels 1-7. The legal codes, their interpretations, and the defaults for *HF* are:

Formatter	HF Code			Default HF
	1	2	3	
nroff	no underline	underline	bold	3 3 2 2 2 2 2
troff	roman	italic	bold	3 3 2 2 2 2 2

Thus, levels 1 and 2 are bold; levels 3 through 7 are underlined in *nroff* and italic in *troff*. The user may reset *HF* as desired. Any value omitted from the right end of the list is taken to be 1. For example, the following would result in five bold levels and two non-underlined (roman) levels:

```
.ds HF 3 3 3 3 3
```

4.2.2.4.2 *Nroff Underlining Style.* *Nroff* can underline in two ways. The normal style (*.ul* request) is to underline only letters and digits. The continuous style (*.cu* request) underlines all characters, including spaces. By default, *MM* attempts to use the continuous style on any heading that is to be underlined and is short enough to fit on a single line. If a heading is to be underlined, but is too long, it is underlined the normal way (i.e., only letters and digits are underlined).

All underlining of headings can be forced to the normal way by using the *-rU1* flag when invoking *nroff* {2.4}.

4.2.2.4.3 *Heading Point Sizes.* The user may also specify the desired point size for each heading level with the *HP* string (for use with *troff* only).

```
.ds HP [ps1] [ps2] [ps3] [ps4] [ps5] [ps6] [ps7]
```

By default, the text of headings (*.H* and *.HU*) is printed in the same point size as the body *except* that bold stand-alone headings are printed in a size one point smaller than the body. The string *HP*, similar to the string *HF*, can be specified to contain up to seven values, corresponding to the seven levels of headings. For example:

```
.ds HP 12 12 10 10 10 10 10
```

specifies that the first and second level headings are to be printed in 12-point type, with the remainder printed in 10-point. Note that the specified values may also be *relative* point-size changes, e.g.:

```
.ds HP +2 +2 -1 -1
```

If absolute point sizes are specified, then those sizes will be used regardless of the point size of the body of the document. If relative point sizes are specified, then the point sizes for the headings will be relative to the point size of the body, even if the latter is changed.

Null or zero values imply that the *default* size will be used for the corresponding heading level.

*Only the point size of the headings is affected. Specifying a large point size without providing increased vertical spacing (via *.HX* and/or *.HZ*) may cause overprinting.*

4.2.2.5 *Marking Styles—Numerals and Concatenation*

```
.HM [arg1] ... [arg7]
```

The registers named *H1* through *H7* are used as counters for the seven levels of headings. Their values are normally printed using Arabic numerals. The *.HM* macro (heading mark style) allows this choice to be overridden, thus providing “outline” and other document styles. This macro can have up to seven arguments; each argument is a string indicating the type of marking to be used. Legal values and their meanings are shown below; omitted values are interpreted as 1, while illegal values have no effect.

<i>Value</i>	<i>Interpretation</i>
1	Arabic (default for all levels)
0001	Arabic with enough leading zeroes to get the specified number of digits
A	Upper-case alphabetic
a	Lower-case alphabetic
I	Upper-case Roman
i	Lower-case Roman

By default, the complete heading mark for a given level is built by concatenating the mark for that level to the right of all marks for all levels of higher value. To inhibit the concatenation of heading level marks, i.e., to obtain just the current level mark followed by a period, set the register *Ht* (heading-mark type) to 1.

For example, a commonly-used “outline” style is obtained by:

```
.HM 1 A 1 a i
.nr Ht 1
```

4.3 Unnumbered Headings

```
.HU heading-text
```

.HU is a special case of *.H*; it is handled in the same way as *.H*, except that no heading mark is printed. In order to preserve the hierarchical structure of headings when *.H* and *.HU* calls are intermixed, each *.HU* heading is considered to exist at the level given by register *Hu*, whose initial value is 2. Thus, in the normal case, the only difference between:

```
.HU heading-text
```

and:

```
.H 2 heading-text
```

is the printing of the heading mark for the latter. Both have the effect of incrementing the numbering counter for level 2, and resetting to zero the counters for levels 3 through 7. Typically, the value of *Hu* should be set to make unnumbered headings (if any) be the lowest-level headings in a document.

.HU can be especially helpful in setting up appendices and other sections that may not fit well into the numbering scheme of the main body of a document {14.2.1}.

4.4 Headings and the Table of Contents

The text of headings and their corresponding page numbers can be automatically collected for a table of contents. This is accomplished by doing the following two things:

- specifying in the register *Ci* what level headings are to be saved
- invoking the *.TC* macro {10.1} at the end of the document

Any heading whose level is less than or equal to the value of the register *Ci* (contents level) is saved and later displayed in the table of contents. The default value for *Ci* is 2, i.e., the first two levels of headings are saved.

Due to the way the headings are saved, it is possible to exceed the formatter’s storage capacity, particularly when saving many levels of many headings, while also processing displays {7} and footnotes {8}. If this happens, the “Out of temp file space” diagnostic {Appendix D} will be issued; the only remedy is to save fewer levels and/or to have fewer words in the heading text.

4.5 First-Level Headings and Page Numbering Style

By default, pages are numbered sequentially at the top of the page. For large documents, it may be desirable to use page numbering of the form “section-page,” where *section* is the number of the

current first-level heading. This page numbering style can be achieved by specifying the `-rN3` or `-rN5` flag on the command line {9.9}. As a side effect, this also has the effect of setting *Ej* to 1, i.e., each section begins on a new page. In this style, the page number is printed at the *bottom* of the page, so that the correct section number is printed.

4.6 User Exit Macros •

This section is intended only for users who are accustomed to writing formatter macros.

```
.HX dlevel rlevel heading-text
.HY dlevel rlevel heading-text
.HZ dlevel rlevel heading-text
```

The `.HX`, `.HY`, and `.HZ` macros are the means by which the user obtains a final level of control over the previously-described heading mechanism. `MM` does not define `.HX`, `.HY`, and `.HZ`; they are intended to be defined by the user. The `.H` macro invokes `.HX` shortly before the actual heading text is printed; it calls `.HZ` as its last action. After `.HX` is invoked, the size of the heading is calculated. This processing causes certain features that may have been included in `.HX`, such as `.ti` for temporary indent, to be lost. After the size calculation, `.HY` is invoked so that the user may respecify these features. All the default actions occur if these macros are not defined. If the `.HX`, `.HY`, or `.HZ` are defined by the user, the user-supplied definition is interpreted at the appropriate point. These macros can therefore influence the handling of all headings, because the `.HU` macro is actually a special case of the `.H` macro.

If the user originally invoked the `.H` macro, then the derived level (*dlevel*) and the real level (*rlevel*) are both equal to the level given in the `.H` invocation. If the user originally invoked the `.HU` macro {4.3}, *dlevel* is equal to the contents of register *Hu*, and *rlevel* is 0. In both cases, *heading-text* is the text of the original invocation.

By the time `.H` calls `.HX`, it has already incremented the heading counter of the specified level {4.2.2.5}, produced blank line(s) (vertical space) to precede the heading {4.2.2.1}, and accumulated the “heading mark”, i.e., the string of digits, letters, and periods needed for a numbered heading. When `.HX` is called, all user-accessible registers and strings can be referenced, as well as the following:

- string ;0 If *rlevel* is non-zero, this string contains the “heading mark.” Two unpadding spaces (to separate the *mark* from the *heading*) have been appended to this string. If *rlevel* is 0, this string is null.
- register ;0 This register indicates the type of spacing that is to follow the heading {4.2.2.2}. A value of 0 means that the heading is run-in. A value of 1 means a break (but no blank line) is to follow the heading. A value of 2 means that a blank line (½ a vertical space) is to follow the heading.
- string ;2 If register ;0 is 0, this string contains two unpadding spaces that will be used to separate the (run-in) *heading* from the following *text*. If register ;0 is non-zero, this string is null.
- register ;3 This register contains an adjustment factor for a `.ne` request issued before the heading is actually printed. On entry to `.HX`, it has the value 3 if *dlevel* equals 1, and 1 otherwise. The `.ne` request is for the following number of lines: the contents of the register ;0 taken as blank lines (halves of vertical space) plus the contents of register ;3 as blank lines (halves of vertical space) plus the number of lines of the heading.

The user may alter the values of ;0, ;2, and ;3 within `.HX` as desired. The following are examples of actions that might be performed by defining `.HX` to include the lines shown:

- The *List End* (.LE) macro that terminates the list and restores the previous indentation.

Lists may be nested up to six levels. The list-initialization macro saves the previous list status (indentation, marking style, etc.); the .LE macro restores it.

With this approach, the format of a list is specified only once at the beginning of that list. In addition, by building on the existing structure, users may create their own customized sets of list macros with relatively little effort {5.4, Appendix A}.

5.2 Sample Nested Lists

The input for several lists and the corresponding output are shown below. The .AL and .DL macro calls {5.3.3} contained therein are examples of the *list-initialization* macros. This example will help us to explain the material in the following sections. Input text:

```
.AL A
.LI
This is an alphabetized item.
This text shows the alignment of the second line of the item.
The quick brown fox jumped over the lazy dog's back.
.AL
.LI
This is a numbered item.
This text shows the alignment of the second line of the item.
The quick brown fox jumped over the lazy dog's back.
.DL
.LI
This is a dash item.
This text shows the alignment of the second line of the item.
The quick brown fox jumped over the lazy dog's back.
.LI + 1
This is a dash item with a ``plus`` as prefix.
This text shows the alignment of the second line of the item.
The quick brown fox jumped over the lazy dog's back.
.LE
.LI
This is numbered item 2.
.LE
.LI
This is another alphabetized item, B.
This text shows the alignment of the second line of the item.
The quick brown fox jumped over the lazy dog's back.
.LE
.P
This paragraph appears at the left margin.
```

Output:

- A. This is an alphabetized item. This text shows the alignment of the second line of the item. The quick brown fox jumped over the lazy dog's back.
 - 1. This is a numbered item. This text shows the alignment of the second line of the item. The quick brown fox jumped over the lazy dog's back.
 - This is a dash item. This text shows the alignment of the second line of the item. The quick brown fox jumped over the lazy dog's back.
 - + — This is a dash item with a "plus" as prefix. This text shows the alignment of the second line of the item. The quick brown fox jumped over the lazy dog's back.
 - 2. This is numbered item 2.
- B. This is another alphabetized item, B. This text shows the alignment of the second line of the item. The quick brown fox jumped over the lazy dog's back.

This paragraph appears at the left margin.

5.3 Basic List Macros

Because all lists share the same overall structure except for the list-initialization macro, we first discuss the macros common to all lists. Each list-initialization macro is covered in {5.3.3}.

5.3.1 List Item

```
.LI [mark] [1]
one or more lines of text that make up the list item.
```

The `.LI` macro is used with all lists. It normally causes the output of a single blank line ($\frac{1}{2}$ a vertical space) before its item, although this may be suppressed. If no arguments are given, it labels its item with the *current mark*, which is specified by the most recent list-initialization macro. If a single argument is given to `.LI`, that argument is output *instead of* the current mark. If two arguments are given, the first argument becomes a *prefix* to the current mark, thus allowing the user to emphasize one or more items in a list. One unpaddingable space is inserted between the prefix and the mark. For example:

```
.BL 6
.LI
This is a simple bullet item.
.LI +
This replaces the bullet with a ``plus.''
.LI + 1
But this uses ``plus'' as prefix to the bullet.
.LE
```

yields:

- This is a simple bullet item.
- + This replaces the bullet with a "plus."
- + ● But this uses "plus" as prefix to the bullet.

⚠ *The mark must not contain ordinary (paddingable) spaces, because alignment of items will be lost if the right margin is justified {3.3}.*

If the *current mark* (in the *current list*) is a null string, and the first argument of `.LI` is omitted or null, the resulting effect is that of a *hanging indent*, i.e., the first line of the following text is "outdented," starting at the same place where the *mark* would have started {5.3.3.6}.

5.3.2 List End

`.LE [1]`

List End restores the state of the list back to that existing just before the most recent list-initialization macro call. If the optional argument is given, the `.LE` outputs a blank line ($\frac{1}{2}$ a vertical space). This option should generally be used only when the `.LE` is followed by running text, but not when followed by a macro that produces blank lines of its own, such as `.P`, `.H`, or `.LI`.

`.H` and `.HU` automatically clear all list information, so one may legally omit the `.LE(s)` that would normally occur just before either of these macros. Such a practice is *not* recommended, however, because errors will occur if the list text is separated from the heading at some later time (e.g., by insertion of text).

5.3.3 List Initialization Macros. The following are the various list-initialization macros. They are actually implemented as calls to the more basic `.LB` macro {5.4}.

5.3.3.1 Automatically-Numbered or Alphabetized Lists

`.AL [type] [text-indent] [1]`

The `.AL` macro is used to begin sequentially-numbered or alphabetized lists. If there are no arguments, the list is numbered, and text is indented *Li*, initially 6 (5)⁵ spaces from the indent in force when the `.AL` is called, thus leaving room for a space, two digits, a period, and two spaces before the text.

Spacing at the beginning of the list and between the items can be suppressed by setting the *Ls* (list space) register. *Ls* is set to the innermost list level for which spacing *is* done. For example:

`.nr Ls 0`

specifies that no spacing will occur around *any* list items. The default value for *Ls* is 6 (which is the *maximum* list nesting level).

The *type* argument may be given to obtain a different type of sequencing, and its value should indicate the first element in the sequence desired, i.e., it must be 1, A, a, l, or i {4.2.2.5}.⁶ If *type* is omitted or null, then "1" is assumed. If *text-indent* is non-null, it is used as the number of spaces from the current indent to the text, i.e., it is used instead of *Li* for this list only. If *text-indent* is null, then the value of *Li* will be used.

If the third argument is given, a blank line ($\frac{1}{2}$ a vertical space) will *not* separate the items in the list. A blank line ($\frac{1}{2}$ a vertical space) will occur before the first item, however.

5.3.3.2 Bullet List

`.BL [text-indent] [1]`

`.BL` begins a bullet list, in which each item is marked by a bullet (●) followed by one space. If *text-indent* is non-null, it overrides the default indentation—the amount of paragraph indentation as given in the register *Pi* {4.1}.⁷

If a second argument is specified, no blank lines will separate the items in the list.

5. Values that specify indentation must be *unscaled* and are treated as "character positions," i.e., as the number of *ens*.

6. Note that the "0001" format is *not* permitted.

7. So that, in the default case, the text of bullet and dash lists lines up with the first line of indented paragraphs.

5.3.3.3 Dash List

`.DL [text-indent] [1]`

`.DL` is identical to `.BL`, except that a dash is used instead of a bullet.

5.3.3.4 Marked List

`.ML mark [text-indent] [1]`

`.ML` is much like `.BL` and `.DL`, but expects the user to specify an arbitrary mark, which may consist of more than a single character. Text is indented *text-indent* spaces if the second argument is not null; otherwise, the text is indented one more space than the width of *mark*. If the third argument is specified, no blank lines will separate the items in the list.

⚠ The mark must not contain ordinary (paddable) spaces, because alignment of items will be lost if the right margin is justified {3.3}.

5.3.3.5 Reference List

`.RL [text-indent] [1]`

A `.RL` call begins an automatically-numbered list in which the numbers are enclosed by square brackets ([]). *Text-indent* may be supplied, as for `.AL`. If omitted or null, it is assumed to be 6, a convenient value for lists numbered up to 99. If the second argument is specified, no blank lines will separate the items in the list.

5.3.3.6 Variable-Item List

`.VL text-indent [mark-indent] [1]`

When a list begins with a `.VL`, there is effectively no *current mark*; it is expected that each `.LI` will provide its own mark. This form is typically used to display definitions of terms or phrases. *Mark-indent* gives the number of spaces from the current indent to the beginning of the *mark*, and it defaults to 0 if omitted or null. *Text-indent* gives the distance from the current indent to the beginning of the text. If the third argument is specified, no blank lines will separate the items in the list. Here is an example of `.VL` usage:

```
.tr ~
.VL 20 2
.LI mark~1
Here is a description of mark 1;
``mark 1`` of the .LI line contains a tilde translated to an unpaddable space in order
to avoid extra spaces between
``mark`` and ``1`` {3.3}.
.LI second~mark
This is the second mark, also using a tilde translated to an unpaddable space.
.LI third~mark~longer~than~indent:
This item shows the effect of a long mark; one space separates the mark
from the text.
.LI ~
This item effectively has no mark because the
tilde following the .LI is translated into a space.
.LE
```

yields:

- mark 1 Here is a description of mark 1; “mark 1” of the .LI line contains a tilde translated to an unpaddable space in order to avoid extra spaces between “mark” and “1” {3.3}.
- second mark This is the second mark, also using a tilde translated to an unpaddable space.
- third mark longer than indent: This item shows the effect of a long mark; one space separates the mark from the text.
- This item effectively has no mark because the tilde following the .LI is translated into a space.

The tilde argument on the last .LI above is required; otherwise a *hanging indent* would have been produced. A *hanging indent* is produced by using .VL and calling .LI with no arguments or with a null first argument. For example:

```
.VL 10
.LI
Here is some text to show a hanging indent.
The first line of text is at the left margin.
The second is indented 10 spaces.
.LE
```

yields:

Here is some text to show a hanging indent. The first line of text is at the left margin. The second is indented 10 spaces.

~~✗~~ *The mark must not contain ordinary (paddable) spaces, because alignment of items will be lost if the right margin is justified {3.3}.*

5.4 List-Begin Macro and Customized Lists •

```
.LB text-indent mark-indent pad type [mark] [LI-space] [LB-space]
```

The list-initialization macros described above suffice for almost all cases. However, if necessary, one may obtain more control over the layout of lists by using the basic list-begin macro .LB, which is also used by all the other list-initialization macros. Its arguments are as follows:

Text-indent gives the number of spaces that the text is to be indented from the current indent. Normally, this value is taken from the register *Li* for automatic lists and from the register *Pi* for bullet and dash lists.

The combination of *mark-indent* and *pad* determines the placement of the mark. The mark is placed within an area (called *mark area*) that starts *mark-indent* spaces to the right of the current indent, and ends where the text begins (i.e., ends *text-indent* spaces to the right of the current indent).⁸ Within the mark area, the mark is *left-justified* if *pad* is 0. If *pad* is greater than 0, say *n*, then *n* blanks are appended to the mark; the *mark-indent* value is ignored. The resulting string immediately precedes the text. That is, the mark is effectively *right-justified pad* spaces immediately to the left of the text.

Type and *mark* interact to control the type of marking used. If *type* is 0, simple marking is performed using the mark character(s) found in the *mark* argument. If *type* is greater than 0, automatic numbering or alphabetizing is done, and *mark* is then interpreted as the first item in the sequence to be used for numbering or alphabetizing, i.e., it is chosen from the set (1, A, a, I, i) as in {5.3.3.1}. That is:

8. The *mark-indent* argument is typically 0.

<i>Type</i>	<i>Mark</i>	<i>Result</i>
0	omitted	hanging indent
0	<i>string</i>	<i>string</i> is the mark
>0	omitted	arabic numbering
>0	one of: 1, A, a, I, i	automatic numbering or alphabetic sequencing

Each non-zero value of *type* from 1 to 6 selects a different way of displaying the marks. The following table shows the output appearance for each value of *type*:

<i>Type</i>	<i>Appearance</i>
1	x.
2	x)
3	(x)
4	[x]
5	<x>
6	{x}

where *x* is the generated number or letter.

~~✗~~ The mark must not contain ordinary (paddable) spaces, because alignment of items will be lost if the right margin is justified {3.3}.

LI-space gives the number of blank lines (halves of a vertical space) that should be output by each *.LI* macro in the list. If omitted, *LI-space* defaults to 1; the value 0 can be used to obtain compact lists. If *LI-space* is greater than 0, the *.LI* macro issues a *.ne* request for two lines just before printing the mark.

LB-space, the number of blank lines ($\frac{1}{2}$ a vertical space) to be output by *.LB* itself, defaults to 0 if omitted.

There are three reasonable combinations of *LI-space* and *LB-space*. The normal case is to set *LI-space* to 1 and *LB-space* to 0, yielding one blank line *before* each item in the list; such a list is usually terminated with a *“.LE 1”* to end the list with a blank line. In the second case, for a more compact list, set *LI-space* to 0 and *LB-space* to 1, and, again, use *“.LE 1”* at the end of the list. The result is a list with one blank line before and after it. If you set both *LI-space* and *LB-space* to 0, and use *“.LE”* to end the list, a list without *any* blank lines will result.

Appendix A shows how one can build upon the supplied list macros to obtain other kinds of lists.

6. MEMORANDUM AND RELEASED PAPER STYLES

One use of MM is for the preparation of memoranda and released papers, which have special requirements for the first page and for the cover sheet. The information needed for the memorandum or released paper (title, author, date, case numbers, etc.) is entered in the same way for *both* styles; an argument to one macro indicates which style is being used. The following sections describe the macros used to provide this data. The required order is shown in {6.9}.

If neither the memorandum nor released-paper style is desired, the macros described below should be omitted from the input text. If these macros are omitted, the first page will simply have the page header {9} followed by the body of the document.

6.1 Title

.TL [charging-case] [filing-case]
one or more lines of title text

The arguments to the .TL macro are the charging case number(s) and filing case number(s).⁹ The title of the memorandum or paper follows the .TL macro and is processed in fill mode {3.1}. Multiple charging case numbers are entered as "sub-arguments" by separating each from the previous with a comma and a space, and enclosing the *entire* argument within double quotes. Multiple filing case numbers are entered similarly. For example:

```
.TL "12345, 67890" 987654321
On the Construction of a Table of All Even Prime Numbers
```

The .br request may be used to break the title on output into several lines as follows:

```
.TL 12345
First Title Line
.br
\!.br
Second Title Line
```

On output, the title appears after the word "subject" in the memorandum style. In the released-paper style, the title is centered and bold.

If only a charging case number or only a filing case number is given, then it will be separated from the title in the memorandum style by a dash and will appear on the same line as the title. If both case numbers are given and are the same, then "Charging and Filing Case" followed by the number will appear on a line following the title. If the two case numbers are different, then separate lines for "Charging Case" and "File Case" will appear after the title.

6.2 Author(s)

```
.AU name [initials] [loc] [dept] [ext] [room] [arg] [arg] [arg]
.AT [title] ...
```

The .AU macro receives as arguments information that describes an author. If any argument contains blanks, it must be enclosed within double quotes. The first six arguments must appear in the order given (a separate .AU macro is required for each author).

The .AT macro is used to specify the author's title. Up to nine arguments may be given. Each will appear in the Signature Block for memorandum style {6.11.1} on a separate line following the signer's name. The .AT must immediately follow the .AU for the given author. For example:

```
.AU "J. J. Jones" JJJ PY 9876 5432 1Z-234
.AT Director "Materials Research Laboratory"
```

In the "from" portion in the memorandum style, the author's name is followed by location and department number on one line and by room number and extension number on the next. The "x" for the extension is added automatically. The printing of the location, department number, extension number, and room number may be suppressed on the first page of a memorandum by setting the register *Au* to 0; the default value for *Au* is 1. Arguments 7 through 9 of the .AU macro, if present, will follow this "normal" author information in the "from" portion, each on a separate line. These last three arguments may be used for organizational numbering schemes, etc. For example:

```
.AU "S. P. Lename" SPL IH 9988 7766 5H-444 9876-543210.01MF
```

The name, initials, location, and department are also used in the Signature Block {6.11.1}. The author information in the "from" portion, as well as the names and initials in the Signature Block will appear in the same order as the .AU macros.

9. The "charging case" is the case number to which time was charged for the development of the project described in the memorandum. The "filing case" is a number under which the memorandum is to be filed.

The names of the authors in the released-paper style are centered below the title. After the name of the last author, "Bell Laboratories" and the location are centered. For the case of authors from different locations, see {6.8}.

6.3 TM Number(s)

.TM [number] ...

If the memorandum is a Technical Memorandum, the TM numbers are supplied via the .TM macro. Up to nine numbers may be specified. Example:

.TM 7654321 7777777

This macro call is ignored in the released-paper and external-letter styles {6.6}.

6.4 Abstract

.AS [arg] [indent]
text of the abstract
.AE

The .AS (abstract start) and .AE (abstract end) macros bracket the (optional) abstract. Abstracts are printed on page 1 of a document and/or on its cover sheet.¹⁰

In a released paper (first argument of the .MT macro is 4; see {6.6}) and in a Technical Memorandum, if the first argument of .AS is 0, the abstract will be printed on page 1 *and* on the cover sheet (if any); if the first argument of .AS is 1, the abstract will appear *only* on the cover sheet (if any).

In Memoranda for File and in all other documents (other than external letters), if the first argument of .AS is 0, the abstract will appear on page 1 and there will be no cover sheet printed; if the first argument of .AS is 2, the abstract will appear *only* on the cover sheet, which will be produced automatically in this case (i.e., *without* invoking the .CS macro). It is *not* possible to get either an abstract or a cover sheet with an external letter (first argument of the .MT macro is 5).

Notations {6.11.2} such as a "copy to" list are allowed on Memorandum for File cover sheets; the .NS and .NE macros must appear after the .AS 2 and .AE. Headings {4.2, 4.3} and displays {7} are *not* permitted within an abstract.

The abstract is printed with ordinary text margins; an indentation to be used for both margins can be specified as the second argument of .AS.¹¹

6.5 Other Keywords

.OK [keyword] ...

Topical keywords should be specified on a Technical Memorandum cover sheet. Up to nine such keywords or keyword phrases may be specified as arguments to the .OK macro; if any keyword contains spaces, it must be enclosed within double quotes.

6.6 Memorandum Types

.MT [type] [addressee]

The .MT macro controls the format of the top part of the first page of a memorandum or of a released paper, as well as the format of the cover sheets. Legal codes for *type* and the corresponding values are:

10. There are three styles of cover sheet: released paper, Technical Memorandum, and Memorandum for File {10.2}; the last one of these is also used for Engineer's Notes, Memoranda for Record, etc. Cover sheets for released papers and Technical Memoranda are obtained by invoking the .CS macro {10.2}.

11. Values that specify indentation must be *unscaled* and are treated as "character positions," i.e., as the number of *ens*.

<i>Code</i>	<i>Value</i>
" "	no memorandum type printed
0	no memorandum type printed
<i>none</i>	MEMORANDUM FOR FILE
1	MEMORANDUM FOR FILE
2	PROGRAMMER'S NOTES
3	ENGINEER'S NOTES
4	released-paper style
5	external-letter style
" <i>string</i> "	<i>string</i>

If *type* indicates a memorandum style, then *value* will be printed after the last line of author information. If *type* is longer than one character, then the string, itself, will be printed. For example:

```
.MT "Technical Note #5"
```

A simple letter is produced by calling .MT with a null (but *not* omitted!) or zero argument.

The second argument to .MT is the name of the addressee of a letter; if present, that name and the page number replace the normal page header on the second and following pages of a letter:

```
.MT 1 "John Jones"
```

produces

```
John Jones - 2
```

This second argument may *not* be used for this purpose if the first argument is 4 (i.e., for the released-paper style) as explained in {6.8}.

In the external-letter style (.MT 5), only the title (without the word "subject:") and the date are printed in the upper left and right corners, respectively, on the first page. It is expected that preprinted stationery will be used, providing the author's company logo and address.

6.7 Date and Format Changes

6.7.1 Changing the Date. By default, the current date appears in the "date" part of a memorandum. This can be overridden by using:

```
.ND new-date
```

The .ND macro alters the value of the string *DT*, which is initially set to the current date.

6.7.2 Alternate First-Page Format. One can specify that the words "subject," "date," and "from" (in the memorandum style) be omitted and that an alternate company name be used:

```
.AF [company-name]
```

If an argument is given, it replaces "Bell Laboratories", without affecting the other headings. If the argument is *null*, "Bell Laboratories" is suppressed; in this case, extra blank lines are inserted to allow room for stamping the document with a Bell System logo or a Bell Laboratories stamp. .AF with *no* argument suppresses "Bell Laboratories" and the "Subject/Date/From" headings, thus allowing output on preprinted stationery. The use of .AF with no arguments is equivalent to the use of -rA1 {2.4}, except that the latter *must* be used if it is necessary to change the line length and/or page offset (which default to 5.8i and li, respectively, for preprinted forms). The command line options -rOk and -rWk {2.4} are *not* effective with .AF. The only .AF use appropriate for *troff* is to specify a replacement for "Bell Laboratories".

The command line option -rEn {2.4} controls the font of the "Subject/Date/From" block.

6.8 Released-Paper Style

The released-paper style is obtained by specifying:

```
.MT 4 [1]
```

This results in a centered, bold title followed by centered names of authors. The location of the last author is used as the location following "Bell Laboratories" (unless .AF {6.7.2} specifies a different company). If the optional second argument to .MT 4 is given, then the name of each author is followed by the respective company name and location.

Information necessary for the memorandum style but not for the released-paper style is ignored.

If the released-paper style is utilized, most BTL location codes¹² are defined as strings that are the addresses of the corresponding BTL locations. These codes are needed only until the .MT macro is invoked. Thus, *following* the .MT macro, the user may re-use these string names. In addition, the macros described in {6.11} and their associated lines of input are ignored when the released-paper style is specified.

Authors from non-BTL locations may include their affiliations in the released-paper style by specifying the appropriate .AF and defining a string (with a 2 character name such as XX) for the address *before* each .AU. For example:

```
.TL
A Learned Treatise
.AF "Getem Inc."
.ds XX "22 Maple Avenue, Somctown 09999"
.AU "F. Swatter" "" XX
.AF "Bell Laboratories"
.AU "Sam P. Lename" "" CB
.MT 4 1
```

6.9 Order of Invocation of "Beginning" Macros

The macros described in {6.1-6.7}, *if present*, must be given in the following order:

```
.ND new-date
.TL [charging-case] [filing-case]
one or more lines of text
.AF [company-name]
.AU name [initials] [loc] [dept] [ext] [room] [arg] [arg] [arg]
.AT [title] ...
.TM [number] ...
.AS [arg] [indent]
one or more lines of text
.AE
.NS [arg]
one or more lines of text
.NE
.OK [keyword] ...
.MT [type] [addressee]
```

The only *required* macros for a memorandum or a released paper are .TL, .AU, and .MT; all the others (and their associated input lines) may be omitted if the features they provide are not needed. Once

12. Currently, the complete list is: AK, AL, ALF, CB, CH, CP, DR, FJ, HL, HO, HOH, HP, IH, IN, INH, IW, MH, MV, PY, RD, RR, WB, WH, and WV.

.MT has been invoked, *none* of the above macros (except .NS and .NE) can be re-invoked because they are removed from the table of defined macros to save space.

6.10 Example

The input text for this manual begins as follows:

```
.TL
MM\*(EMMemorandum Macros
.AU "D. W. Smith" DWS PY
.AU "J. R. Mashey" JRM PY
.AU "E. C. Pariser (January 1980 Revision)" ECP PY
.AU "N. W. Smith (June 1980 Revision)" NWS PY
.MT 4
```

Appendix C shows the input and both *nroff* and *troff* outputs for a simple letter.

6.11 Macros for the End of a Memorandum

At the end of a memorandum (but *not* of a released paper), the signatures of the authors and a list of notations¹³ can be requested. The following macros and their input are ignored if the released-paper style is selected.

6.11.1 Signature Block

```
.FC [closing]
.SG [arg] [1]
```

.FC prints "Yours very truly," as a formal closing. It must be given before the .SG which prints the signer's name. A different closing may be specified as an argument to .FC.

.SG prints the author name(s) after the formal closing, if any. Each name begins at the center of the page. Three blank lines are left above each name for the actual signature. If no arguments are given, the line of reference data¹⁴ will *not* appear. A non-null first argument is treated as the typist's initials, and is appended to the reference data. Supply a null first argument to print the reference data with neither the typist's initials nor the preceding hyphen.

If there are several authors and if the second argument is given, then the reference data is placed on the same line as the name of the first, rather than last, author.

The reference data contains only the location and department number of the first author. Thus, if there are authors from different departments and/or from different locations, the reference data should be supplied manually after the invocation (without arguments) of the .SG macro. For example:

```
.SG
.rs
.sp -1v
PY/MH-9876/5432-JJJ/SPL-cen
```

6.11.2 "Copy to" and Other Notations

```
.NS [arg]
zero or more lines of the notation
.NE
```

13. See the *BTL Office Guide*^[9], pp. 1.12-16.

14. The following information is known as reference data: location code, department number, author's initials, and typist's initials, all separated by hyphens. See the *BTL Office Guide*^[9], page 1.11.

After the signature and reference data, many types of notations may follow, such as a list of attachments or "copy to" lists. The various notations are obtained through the .NS macro, which provides for the proper spacing and for breaking the notations across pages, if necessary.

The codes for *arg* and the corresponding notations are:

<i>Code</i>	<i>Notations</i>
<i>none</i>	Copy to
" "	Copy to
0	Copy to
1	Copy (with att.) to
2	Copy (without att.) to
3	Att.
4	Atts.
5	Enc.
6	Encs.
7	Under Separate Cover
8	Letter to
9	Memorandum to
" <i>string</i> "	Copy (<i>string</i>) to

If *arg* consists of more than one character, it is placed within parentheses between the words "Copy" and "to." For example:

```
.NS "with att. 1 only"
```

will generate "Copy (with att. 1 only) to" as the notation. More than one notation may be specified before the .NE occurs, because a .NS macro terminates the preceding notation, if any. For example:

```
.NS 4
Attachment 1-List of register names
Attachment 2-List of string and macro names
.NS 1
J. J. Jones
.NS 2
S. P. Lename
G. H. Hurtz
.NE
```

would be formatted as:

```
Atts.
Attachment 1-List of register names
Attachment 2-List of string and macro names

Copy (with att.) to
J. J. Jones

Copy (without att.) to
S. P. Lename
G. H. Hurtz
```

The .NS and .NE macros may also be used at the beginning following .AS 2 and .AE to place the notation list on the Memorandum for File cover sheet {6.4}. If notations are given at the beginning without .AS 2, they will be saved and output at the end of the document.

6.11.3 Approval Signature Line

```
.AV approver's-name
```


The .AV macro may be used after the last notation block to automatically generate a line with spaces for the approval signature and date. For example,

```
.AV "Jane Doe"
```

produces:

APPROVED:

Jane Doe

Date

6.12 Forcing a One-Page Letter

At times, one would like just a bit more space on the page, forcing the signature or items within notations onto the bottom of the page, so that the letter or memo is just one page in length. This can be accomplished by increasing the page length through the `-rLn` option, e.g. `-rL90`. This has the effect of making the formatter believe that the page is 90 lines long and therefore giving it more room than usual to place the signature or the notations. This will *only* work for a *single-page* letter or memo.

7. DISPLAYS

Displays are blocks of text that are to be kept together—not split across pages. MM provides two styles of displays:¹⁵ a *static* (.DS) style and a *floating* (.DF) style. In the *static* style, the display appears in the same relative position in the output text as it does in the input text; this may result in extra white space at the bottom of the page if the display is too big to fit there. In the *floating* style, the display “floats” through the input text to the top of the next page if there is not enough room for it on the current page; thus the input text that *follows* a floating display may *precede* it in the output text. A queue of floating displays is maintained so that their relative order is not disturbed.

By default, a display is processed in no-fill mode, with single-spacing, and is *not* indented from the existing margins. The user can specify indentation or centering, as well as fill-mode processing.

Displays and footnotes {8} may *never* be nested, in any combination whatsoever. Although lists {5} and paragraphs {4.1} are permitted, no headings (.H or .HU) {4.2, 4.3} can occur within displays or footnotes.

7.1 Static Displays

```
.DS [format] [fill] [rindent]
one or more lines of text
.DE
```

A static display is started by the .DS macro and terminated by the .DE macro. With no arguments, .DS will accept the lines of text exactly as they are typed (no-fill mode) and will *not* indent them from the prevailing left margin indentation or from the right margin. The *rindent* argument is the number of characters¹⁶ that the line length should be decreased, i.e., an indentation from the right margin.

The *format* argument to .DS is an integer or letter used to control the left margin indentation and centering with the following meanings:

15. Displays are processed in an environment that is different from that of the body of the text (see the .ev request^[2]).

16. This number must be unscaled in *troff* and is treated as *ems*. It may be scaled in *troff* or else defaults to *ems*.

<i>Format</i>	<i>Meaning</i>
"	no indent
0 or L	no indent
1 or I	indent by standard amount
2 or C	center each line
3 or CB	center as a block

The *fill* argument is also an integer or letter and can have the following meanings:

<i>Fill</i>	<i>Meaning</i>
"	no-fill mode
0 or N	no-fill mode
1 or F	fill mode

Omitted arguments are taken to be zero.

The standard amount of indentation is taken from the register *Si*, which is initially 5. Thus, by default, the text of an indented display aligns with the first line of indented paragraphs, whose indent is contained in the *Pi* register {4.1}. Even though their initial values are the same, these two registers are independent of one another.

The display format value 3 (CB) centers the *entire display* as a block (as opposed to .DS 2 and .DF 2, which center each line *individually*). That is, all the collected lines are left-justified, and then the display is centered, based on the width of the longest line. This format *must* be used in order for the *eqn/eqn* “mark” and “lineup” feature to work with centered equations (see section 7.4 below).

By default, a blank line (½ a vertical space) is placed before and after static and floating displays. These blank lines before and after *static* displays can be inhibited by setting the register *Ds* to 0.

The following example shows the usage of all three arguments for displays. This block of text will be filled and indented 5 spaces from both the left and the right margins (i.e., centered).

```
.DS I F 5
```

```
“We the people of the United States, in order to form a more perfect union,
establish justice, ensure domestic tranquility, provide for the common defense,
and secure the blessings of liberty to ourselves and our posterity,
do ordain and establish this Constitution to the
United States of America.”
```

```
.DE
```

7.2 Floating Displays

```
.DF [format] [fill] [rindent]
```

```
one or more lines of text
```

```
.DE
```

A floating display is started by the .DF macro and terminated by the .DE macro. The arguments have the same meanings as for .DS {7.1}, except that, for floating displays, indent, no indent, and centering are always calculated with respect to the initial left margin, because the prevailing indent may change between the time when the formatter first reads the floating display and the time that the display is printed. One blank line (½ a vertical space) *always* occurs both before and after a floating display.

The user may exercise great control over the output positioning of floating displays through the use of two number registers, *De* and *Df*. When a floating display is encountered by *nroff* or *troff*, it is processed and placed onto a queue of displays waiting to be output. Displays are always removed from the queue and printed in the order that they were entered on the queue, which is the order that they appeared in the input file. If a new floating display is encountered and the queue of displays is empty, then the new display is a candidate for immediate output on the current page. Immediate output is governed by the size of the display and the setting of the *Df* register (see below). The *De* register (see

below) controls whether or not text will appear on the current page after a floating display has been produced.

As long as the queue contains one or more displays, new displays will be automatically entered there, rather than being output. When a new page is started (or the top of the second column when in two-column mode) the next display from the queue becomes a candidate for output if the *Df* register has specified "top-of-page" output. When a display is output it is also removed from the queue.

When the end of a section (when using section-page numbering) or the end of a document is reached, all displays are automatically removed from the queue and output. This will occur before a .SG, .CS, or .TC is processed.

A display is said to "fit on the current page" if there is enough room to contain the entire display on the page, or if the display is longer than one page in length and less than half of the current page has been used. Also note that a wide (full page width) display will never fit in the second column of a two-column document.

The registers, their settings, and their effects are as follows:

<i>De</i>	<i>Action</i>
0	DEFAULT: No special action occurs.
1	A page eject will <i>always</i> follow the output of each floating display, so only one floating display will appear on a page and no text will follow it.
	NOTE: For any other value, the action performed is the same as for the value 1.

<i>Df</i>	<i>Action</i>
0	Floating displays will not be output until end of section (when section-page numbering) or end of document.
1	Output the new floating display on the current page if there is room, otherwise hold it until the end of the section or document.
2	Output exactly one floating display from the queue at the top of a new page or column (when in two-column mode).
3	Output one floating display on current page if there is room. Output exactly one floating display at the top of a new page or column.
4	Output as many displays as will fit (at least one), starting at the top of a new page or column. Note that if register <i>De</i> is set to 1, each display will be followed by a page eject, causing a new top of page to be reached where at least one more display will be output. (This also applies to value 5, below.)
5	DEFAULT: Output a new floating display on the current page if there is room. Output at least one, but as many displays as will fit starting at the top of a new page or column.
	NOTE: For any value greater than 5, the action performed is the same as for the value 5.

The .WC macro {12.4} may also be used to control handling of displays in double-column mode and to control the break in the text before floating displays.

7.3 Tables

```
.TS [H]
global options;
column descriptors.
title lines
[.TH [N]]
data within the table.
.TE
```

The `.TS` (table start) and `.TE` (table end) macros make possible the use of the `tbl(1)` processor^[7]. They are used to delimit the text to be examined by `tbl(1)` as well as to set proper spacing around the table. The display function and the `tbl(1)` delimiting function are independent of one another, however, so in order to permit one to keep together blocks that contain any mixture of tables, equations, filled and unfilled text, and caption lines the `.TS/.TE` block should be enclosed within a display (`.DS/.DE`). Floating tables may be enclosed inside floating displays (`.DF/.DE`).

The macros `.TS` and `.TE` also permit the processing of tables that extend over several pages. If a table heading is needed for each page of a multi-page table, specify the argument "H" to the `.TS` macro as above. Following the options and format information, the table heading is typed on as many lines as required and followed by the `.TH` macro. The `.TH` macro *must* occur when "`.TS H`" is used. Note that this is *not* a feature of `tbl(1)`, but of the macro definitions provided by MM.

The table header macro `.TH` may take as an argument the letter N. This argument causes the table header to be printed only if it is the first table header on the page. This option is used when it is necessary to build long tables from smaller `.TS H/.TE` segments. For example:

```
.TS H
global options;
column descriptors.
Title lines
.TH
data
.TE
.TS H
global options;
column descriptors.
Title lines
.TH N
data
.TE
```

will cause the table heading to appear at the top of the first table segment, and no heading to appear at the top of the second segment when both appear on the same page. However, the heading will still appear at the top of each page that the table continues onto. This feature is used when a single table must be broken into segments because of table complexity (for example, too many blocks of filled text). If each segment had its own `.TS H/.TH` sequence, each segment would have its own header. However, if each table segment after the first uses `.TS H/.TH N` then the table header will only appear at the beginning of the table and the top of each new page or column that the table continues onto.

For `nroff`, the `-e` option (`-E` for `mm(1)` {2.1}) may be used for terminals, such as the 450, that are capable of finer printing resolution. This will cause better alignment of features such as the lines forming the corner of a box. Note that `-e` is not effective with `col(1)`.

7.4 Equations

```
.DS
.EQ [label]
equation(s)
.EN
.DE
```

The equation setters *eqn*(1) and *neqn*^[6] expect to use the .EQ (equation start) and .EN (equation end) macros as delimiters in the same way that *tbl*(1) uses .TS and .TE; however, .EQ and .EN must occur inside a .DS/.DE pair.

There is an exception to this rule: if .EQ and .EN are used only to specify the delimiters for in-line equations or to specify eqn/neqn “defines,” .DS and .DE must not be used; otherwise extra blank lines will appear in the output.

The .EQ macro takes an argument that will be used as a label for the equation. By default, the label will appear at the right margin in the “vertical center” of the general equation. The *Eq* register may be set to 1 to change the labeling to the left margin.

The equation will be centered for centered displays; otherwise the equation will be adjusted to the opposite margin from the label.

7.5 Figure, Table, Equation, and Exhibit Captions

```
.FG [title] [override] [flag]
.TB [title] [override] [flag]
.EC [title] [override] [flag]
.EX [title] [override] [flag]
```

The .FG (Figure Title), .TB (Table Title), .EC (Equation Caption) and .EX (Exhibit Caption) macros are normally used inside .DS/.DE pairs to automatically number and title figures, tables, and equations. They use registers *Fg*, *Tb*, *Ec*, and *Ex*, respectively (see {2.4} on *-rN5* to reset counters in sections). As an example, the call:

```
.FG "This is an illustration"
```

yields:

Figure 1. This is an illustration

.TB replaces “Figure” by “TABLE”; .EC replaces “Figure” by “Equation”, and .EX replaces “Figure” by “Exhibit”. Output is centered if it can fit on a single line; otherwise, all lines but the first are indented to line up with the first character of the title. The format of the numbers may be changed using the .af request of the formatter. The format of the caption may be changed from “Figure 1. Title” to “Figure 1 - Title” by setting the *Of* register to 1.

The *override* string *k* used to modify the normal numbering. If *flag* is omitted or 0, *override* is used as a prefix to the number; if *flag* is 1, *override* is used as a suffix; and if *flag* is 2, *override* replaces the number. If *-rN5* {2.4} is given, “section-figure” numbering is set automatically and user-specified *override* string is ignored.

As a matter of style, table headings are usually placed ahead of the text of the tables, while figure, equation, and exhibit captions usually occur after the corresponding figures and equations.

7.6 List of Figures, Tables, Equations, and Exhibits

A List of Figures, List of Tables, List of Exhibits, and List of Equations may be obtained. They will be printed after the Table of Contents is printed if the number registers *Lf*, *Lt*, *Lx*, and *Le* (respectively) are set to 1. *Lf*, *Lt*, and *Lx* are 1 by default; *Le* is 0 by default.

The titles of these Lists may be changed by redefining the following strings which are shown here with their default values:

```
.ds Lf LIST OF FIGURES
.ds Lt LIST OF TABLES
.ds Lx LIST OF EXHIBITS
.ds Le LIST OF EQUATIONS
```

8. FOOTNOTES

There are two macros that delimit the text of footnotes,¹⁷ a string used to automatically number the footnotes, and a macro that specifies the style of the footnote text.

8.1 Automatic Numbering of Footnotes

Footnotes may be automatically numbered by typing the three characters “*F” (i.e., invoking the string *F*) immediately after the text to be footnoted, without any intervening spaces. This will place the next sequential footnote number (in a smaller point size) a half-line above the text to be footnoted.

8.2 Delimiting Footnote Text

There are two macros that delimit the text of each footnote:

```
.FS [label]
  one or more lines of footnote text
.FE
```

The .FS (footnote start) marks the beginning of the text of the footnote, and the .FE marks its end. The *label* on the .FS, if present, will be used to mark the footnote text. Otherwise, the number retrieved from the string *F* will be used. Note that automatically-numbered and user-labeled footnotes may be intermixed. If a footnote is labeled (.FS *label*), the text to be footnoted *must* be followed by *label*, rather than by “*F”. The text between .FS and .FE is processed in fill mode. Another .FS, a .DS, or a .DF are *not* permitted between the .FS and .FE macros. Automatically numbered footnotes may not be used for information, such as the title and abstract, to be placed on the cover sheet, but labeled footnotes are allowed. Similarly, only labeled footnotes may be used with tables {7.3}. Examples:

1. Automatically-numbered footnote:

```
This is the line containing the word\F
.FS
This is the text of the footnote.
.FE
to be footnoted.
```

2. Labelled footnote:

```
This is a labeled*
.FS *
The footnote is labeled with an asterisk.
.FE
footnote.
```

The text of the footnote (enclosed within the .FS/.FE pair) should *immediately* follow the word to be footnoted in the input text, so that “*F” or *label* occurs at the end of a line of input and the next line

¹⁷ Footnotes are processed in an environment that is different from that of the body of the text (see the .ev request^[2]).

is the .FS macro call. It is also good practice to append a unpadding space {3.3} to “*F” or *label* when they follow an end-of-sentence punctuation mark (i.e., period, question mark, exclamation point).

Appendix B illustrates the various available footnote styles as well as numbered and labeled footnotes.

8.3 Format of Footnote Text •

.FD [arg] [1]

Within the footnote text, the user can control the formatting style by specifying text hyphenation, right margin justification, and text indentation, as well as left- or right-justification of the label when text indenting is used. The .FD macro is invoked to select the appropriate style. The first argument is a number from the left column of the following table. The formatting style for each number is given by the remaining four columns. For further explanation of the first two of these columns, see the definitions of the .ad, .hy, .na, and .nh requests^[2].

<i>arg</i>	<i>Hyphenation</i>	<i>Adjust</i>	<i>Text Indent</i>	<i>Label Justification</i>
0	.nh	.ad	text indent	label left justified
1	.hy	.ad	"	"
2	.nh	.na	"	"
3	.hy	.na	"	"
4	.nh	.ad	no text indent	"
5	.hy	.ad	"	"
6	.nh	.na	"	"
7	.hy	.na	"	"
8	.nh	.ad	text indent	label right justified
9	.hy	.ad	"	"
10	.nh	.na	"	"
11	.hy	.na	"	"

If the first argument to .FD is out of range, the effect is as if .FD 0 were specified. If the first argument is omitted or null, the effect is equivalent to .FD 10 in *nroff* and to .FD 0 in *troff*; these are also the respective initial defaults.

If a second argument is specified, then whenever a first-level heading is encountered, automatically-numbered footnotes begin again with 1. This is most useful with the “section-page” page numbering scheme. As an example, the input line:

.FD " 1

maintains the default formatting style and causes footnotes to be numbered afresh after each first-level heading.

For long footnotes that continue onto the following page, it is possible that, if hyphenation is permitted, the last line of the footnote on the current page will be hyphenated. Except for this case (over which the user has control by specifying an *even* argument to .FD), hyphenation across pages is inhibited by MM.

Footnotes are separated from the body of the text by a short rule. Footnotes that continue to the next page are separated from the body of the text by a full-width rule. In *troff*, footnotes are set in type that is two points smaller than the point size used in the body of the text.

8.4 Spacing between Footnote Entries

Normally, one blank line (a three-point vertical space) separates the footnotes when more than one occurs on a page. To change this spacing, set the register *Fs* to the desired value. For example:

.nr Fs 2

will cause two blank lines (a six-point vertical space) to occur between footnotes.

9. PAGE HEADERS AND FOOTERS

Text that occurs at the top of each page is known as the *page header*. Text printed at the bottom of each page is called the *page footer*. There can be up to three lines of text associated with the header: every page, even page only, and odd page only. Thus the page header may have up to two lines of text: the line that occurs at the top of every page and the line for the even- or odd-numbered page. The same is true for the page footer.

This section first describes the default appearance of page headers and page footers, and then the ways of changing them. We use the term *header* (not qualified by *even* or *odd*) to mean the line of the page header that occurs on every page, and similarly for the term *footer*.

9.1 Default Headers and Footers

By default, each page has a centered page number as the header {9.2}. There is no default footer and no even/odd default headers or footers, except as specified in {9.9}.

In a memorandum or a released paper, the page header on the first page is automatically suppressed *provided* a break does *not* occur before .MT is called. The macros and text of {6.9} and of {9} as well as .nr and .ds requests do *not* cause a break and are permitted before the .MT macro call.

9.2 Page Header

.PH [arg]

For this and for the .EH, .OH, .PF, .EF, .OF macros, the argument is of the form:

"`left-part`center-part`right-part`"

If it is inconvenient to use the apostrophe (`) as the delimiter (i.e., because it occurs within one of the parts), it may be replaced *uniformly* by *any* other character. On output, the parts are left-justified, centered, and right-justified, respectively. See {9.11} for examples.

The .PH macro specifies the header that is to appear at the top of every page. The initial value (as stated in {9.1}) is the default centered page number enclosed by hyphens. The page number contained in the *P* register is an Arabic number. The format of the number may be changed by the .af request.

If *debug mode* is set using the flag -rD1 on the command line {2.4}, additional information, printed at the top left of each page, is included in the default header. This consists of the SCCS^[10] Release and Level of MM (thus identifying the current version {12.3}), followed by the current line number within the current input file.

9.3 Even-Page Header

.EH [arg]

The .EH macro supplies a line to be printed at the top of each even-numbered page, immediately *following* the header. The initial value is a blank line.

9.4 Odd-Page Header

.OH [arg]

This macro is the same as .EH, except that it applies to odd-numbered pages.

9.5 Page Footer

.PF [arg]

The .PF macro specifies the line that is to appear at the bottom of each page. Its initial value is a blank line. If the -rC*n* flag is specified on the command line {2.4}, the type of copy *follows* the footer on a separate line. In particular, if -rC3 or -rC4 (DRAFT) is specified, then, in addition, the footer is initialized to contain the date {6.7.1}, instead of being a blank line.

9.6 Even-Page Footer

`.EF [arg]`

The `.EF` macro supplies a line to be printed at the bottom of each even-numbered page, immediately *preceding* the footer. The initial value is a blank line.

9.7 Odd-Page Footer

`.OF [arg]`

This macro is the same as `.EF`, except that it applies to odd-numbered pages.

9.8 Footer on the First Page

By default, the footer is a blank line. If, in the input text, one specifies `.PF` and/or `.OF` before the end of the first page of the document, then these lines will appear at the bottom of the first page.

The header (whatever its contents) *replaces* the footer *on the first page only* if the `-rN1` flag is specified on the command line {2.4}.

9.9 Default Header and Footer with “Section-Page” Numbering

Pages can be numbered sequentially within sections {4.5}. To obtain this numbering style, specify `-rN3` or `-rN5` on the command line. In this case, the default *footer* is a centered “section-page” number, e.g. 7-2, and the default page header is blank.

9.10 Use of Strings and Registers in Header and Footer Macros •

String and register names may be placed in the arguments to the header and footer macros. If the value of the string or register is to be computed *when the respective header or footer is printed*, the invocation must be escaped by four (4) backslashes. This is because the string or register invocation will be processed three times:

- as the argument to the header or footer macro;
- in a formatting request within the header or footer macro;
- in a `.tl` request during header or footer processing.

For example, the page number register `P` must be escaped with four backslashes in order to specify a header in which the page number is to be printed at the right margin, e.g.:

```
.PH "^^^Page \\\nP"
```

creates a right-justified header containing the word “Page” followed by the page number. Similarly, to specify a footer with the “section-page” style, one specifies (see {4.2.2.5} for meaning of `H1`):

```
.PF "^^^- \\\n(H1-\\nP -"
```

As another example, suppose that the user arranges for the string `a/` to contain the current section heading which is to be printed at the bottom of each page. The `.PF` macro call would then be:

```
.PF "^^\\*(a)^^"
```

If only one or two backslashes were used, the footer would print a constant value for `a/`, namely, its value when the `.PF` appeared in the input text.

9.11 Header and Footer Example •

The following sequence specifies blank lines for the header and footer lines, page numbers on the outside edge of each page (i.e., top left margin of even pages and top right margin of odd pages), and “Revision 3” on the top inside margin of each page:

```
.PH ""
.PF ""
.EH "^\nP^ Revision 3^"
.OH "^\nP^ Revision 3^"^\nP^"
```

9.12 Generalized Top-of-Page Processing •

This section is intended only for users accustomed to writing formatter macros.

During header processing, MM invokes two user-definable macros. One, the .TP macro, is invoked in the environment (see .ev request^[2]) of the header; the other, .PX, is a user-exit macro that is invoked (without arguments) when the normal environment has been restored, and with “no-space” mode already in effect.

The effective initial definition of .TP (after the first page of a document) is:

```
.de TP
^sp 3
.tl \*(}t
.if e ^tl \*(}e
.if o ^tl \*(}o
^sp 2
..
```

The string }t contains the header, the string }e contains the even-page header, and the string }o contains the odd-page header, as defined by the .PH, .EH, and .OH macros, respectively. To obtain more specialized page titles, the user may redefine the .TP macro to cause any desired header processing {12.5}. Note that formatting done within the .TP macro is processed in an environment different from that of the body.

For example, to obtain a page header that includes three centered lines of data, say, a document's number, issue date, and revision date, one could define .TP as follows:

```
.de TP
.sp
.ce 3
777-888-999
Iss. 2, AUG 1977
Rev. 7, SEP 1977
.sp
..
```

The .PX macro may be used to provide text that is to appear at the top of each page after the normal header and that may have tab stops to align it with columns of text in the body of the document.

9.13 Generalized Bottom-of-Page Processing

```
.BS
zero or more lines of text
.BE
```

Lines of text that are specified between the .BS (bottom-block start) and .BE (bottom-block end) macros will be printed at the bottom of each page,¹⁸ after the footnotes (if any), but before the page footer. This block of text is removed by specifying an empty block, i.e.:

18. The bottom block will appear on the table of contents pages and the cover sheet for Memorandum for File, but *not* on the Technical Memorandum or released-paper cover sheets.

.BS
.BE

9.14 Top and Bottom Margins

.VM [top] [bottom]

.VM (Vertical Margin) allows the user to specify extra space at the top and bottom of the page. This space precedes the page header and follows the page footer. .VM takes two unscaled arguments that are treated as v's. For example:

.VM 10 15

adds 10 blank lines to the default top of page margin, and 15 blank lines to the default bottom of page margin. Both arguments must be positive (default spacing at the top of the page may be decreased by redefining .TP).

9.15 Proprietary Markings

.PM [code]

.PM, for Proprietary Marking, appends to the page footer a PRIVATE, NOTICE, BELL LABORATORIES PROPRIETARY, or BELL LABORATORIES RESTRICTED disclaimer. The *code* may be:

<i>Code</i>	<i>Meaning</i>
<i>none</i>	turn off previous disclaimer, if any
P	PRIVATE
N	NOTICE
BP	BELL LABORATORIES PROPRIETARY
BR	BELL LABORATORIES RESTRICTED

The disclaimers are in a form approved for use by the Bell System.

9.16 Private Documents

.nr Pv value

The word "PRIVATE" may be printed centered and underlined on the second line of a document (preceding the page header). This is done by setting the *Pv* register:

<i>Value</i>	<i>Meaning</i>
0	do not print PRIVATE (default)
1	PRIVATE on first page only
2	PRIVATE on all pages

If *Pv* is 2, the user definable .TP may not be used because .TP is used by MM to print PRIVATE on all pages except the first page of a memorandum on which .TP is not invoked.

10. TABLE OF CONTENTS AND COVER SHEET

The table of contents and the cover sheet for a document are produced by invoking the .TC and .CS macros, respectively.

This section will refer to cover sheets for Technical Memoranda and released papers only. The mechanism for producing a Memorandum for File cover sheet was discussed earlier {6.4}.

These macros should normally appear only once at the *end* of the document, after the Signature Block {6.11.1} and Notations {6.11.2} macros. They may occur in either order.

The table of contents is produced at the end of the document because the entire document must be processed before the table of contents can be generated. Similarly, the cover sheet is often not needed, and is therefore produced at the end.

10.1 Table of Contents

```
.TC [slevel] [spacing] [tlevel] [tab] [head1] [head2] [head3] [head4] [head5]
```

The .TC macro generates a table of contents containing the headings that were saved for the table of contents as determined by the value of the *CI* register {4.4}. The arguments to .TC control the spacing before each entry, the placement of the associated page number, and additional text on the first page of the table of contents before the word "CONTENTS."

Spacing before each entry is controlled by the first two arguments; headings whose level is less than or equal to *slevel* will have *spacing* blank lines (halves of a vertical space) before them. Both *slevel* and *spacing* default to 1. This means that first-level headings are preceded by one blank line ($\frac{1}{2}$ a vertical space). Note that *slevel* does *not* control what levels of heading have been saved; the saving of headings is the function of the *CI* register {4.4}.

The third and fourth arguments control the placement of the page number for each heading. The page numbers can be justified at the right margin with either blanks or dots ("leaders") separating the heading text from the page number, or the page numbers can follow the heading text. For headings whose level is less than or equal to *tlevel* (default 2), the page numbers are justified at the right margin. In this case, the value of *tab* determines the character used to separate the heading text from the page number. If *tab* is 0 (the default value), dots (i.e., leaders) are used; if *tab* is greater than 0, spaces are used. For headings whose level is greater than *tlevel*, the page numbers are separated from the heading text by two spaces (i.e., they are "ragged right").

All additional arguments (e.g., *head1*, *head2*, etc.), if any, are horizontally centered on the page, and precede the actual table of contents itself.

If the .TC macro is invoked with at most four arguments, then the user-exit macro .TX is invoked (without arguments) before the word "CONTENTS" is printed; or the user-exit macro .TY is invoked and the word "CONTENTS" is *not* printed. By defining .TX or .TY and invoking .TC with at most four arguments, the user can specify what needs to be done at the top of the (first) page of the table of contents. For example, the following input:

```
.de TX
.ce 2
Special Application
Message Transmission
.sp 2
.in +10n
Approved: \l'3i'
.in
.sp
.:
.TC
```

yields:

Special Application
Message Transmission

Approved: _____

CONTENTS

⋮

If this macro were defined as .TY rather than .TX, the word "CONTENTS" would not appear. Defining .TY as an empty macro will suppress "CONTENTS" with no replacement:

```
.de TY
```

```
..
```

By default, the first level headings will appear in the table of contents at the left margin. Subsequent levels will be aligned with the text of headings at the preceding level. These indentations may be changed by defining the *Ci* string which takes a maximum of seven arguments corresponding to the heading levels. It must be given at least as many arguments as are set by the *Cl* register {4.4}. The arguments must be scaled. For example, with *Cl*=5,

```
.ds Ci .25i .5i .75i 1i 1i
```

or

```
.ds Ci 0 2n 4n 6n 8n
```

Two other registers are available to modify the format of the table of contents, *Oc* and *Cp*. By default, table of contents pages will have lower-case Roman numeral page numbering. If the *Oc* register is set to 1, the *.TC* macro will not print any page number but will instead reset the *P* register to 1. It is the user's responsibility to give an appropriate page footer to place the page number. Ordinarily the same *.PF* used in the body of the document will be adequate.

The List of Figures, Tables, etc. pages will be produced separately unless *Cp* is set to 1 which causes these lists to appear on the same page as the table of contents.

10.2 Cover Sheet

```
.CS [pages] [other] [total] [figs] [tbls] [refs]
```

The *.CS* macro generates a cover sheet in either the released paper or Technical Memorandum style (see {6.4} for details of the Memorandum for File cover sheet). All the other information for the cover sheet is obtained from the data given before the *.MT* macro call {6.9}. If the Technical Memorandum style is used, the *.CS* macro generates the "Cover Sheet for Technical Memorandum." The data that appear in the lower left corner of the Technical Memorandum cover sheet^[9] (the counts of: pages of text, other pages, total pages, figures, tables, and references) are generated automatically (0 is used for the count of "other pages"). These values may be changed by supplying the corresponding arguments to the *.CS* macro. If the released-paper style is used, all arguments to *.CS* are ignored.

11. REFERENCES

There are two macros that delimit the text of references, a string used to automatically number the references, and an optional macro to produce reference pages within the document.

11.1 Automatic Numbering of References

Automatically numbered references may be obtained by typing **(Rf* (i.e., invoking the string *Rf*) immediately after the text to be referenced. This places the next sequential reference number (in a smaller point size) enclosed in brackets a half-line above the text to be referenced, as illustrated throughout this document. The reference count is kept in the number register *Rf*.

11.2 Delimiting Reference Text

The *.RS* and *.RF* macros are used to delimit the text of each reference.

```
A line of text to be referenced\*(Rf
.RS [string-name]
reference text
.RF
```

11.3 Subsequent References

.RS takes one argument, a *string-name*. For example:

```
.RS aA
reference text
.RF
```

The string *aA* is assigned the current reference number. It may be used later in the document, as the string call, *(*aA*), to reference text which must be labeled with a prior reference number. The reference is output enclosed in brackets a half-line above the text to be referenced. No .RS/.RF pair is needed for subsequent references.

11.4 Reference Page

A reference page, entitled by default "References", will be generated *automatically* at the end of the document (before the Table of Contents and the Cover Sheet) and will be listed in the Table of Contents. This page contains the reference items (i.e., text enclosed within .RS/.RF pairs). Reference items will be separated by a space (1/2 space) unless the *Ls* register is set to 0 to suppress this spacing. The user may change the reference page title by defining the *Rp* string:

```
.ds Rp "New Title"
```

The .RP (Reference Page) macro may be used to produce reference pages anywhere else within a document (i.e., after each major section); .RP is *not* needed to produce a separate reference page with default spacings at the end of the document.

```
.RP [arg1] [arg2]
```

The two arguments allow the user to control resetting of reference numbering and page skipping.

<i>arg1</i>	<i>Meaning</i>
0	reset reference counter (default)
1	do not reset reference counter

<i>arg2</i>	<i>Meaning</i>
0	put on separate page (default)
1	do not cause a following .SK
2	do not cause a preceding .SK
3	no .SK before or after

If no .SK is issued by .RP, then a single blank line will separate the References from the following (preceding) text. The user may wish to adjust the spacing. For example, to produce references at the end of each major section:

```
.sp 3
.RP 1 2
.H 1 "Next Section"
```

12. MISCELLANEOUS FEATURES

12.1 Bold, Italic, and Roman Fonts

```
.B [bold-arg] [previous-font-arg] ...
.I [italic-arg] [previous-font-arg] ...
.R
```

When called without arguments, .B changes the font to bold and .I changes to underlining (italic). This condition continues until the occurrence of a .R, when the (regular) roman font is restored. Thus:

```
.I
here is some text.
.R
```

yields:

here is some text.

If **.B** or **.I** is called with one argument, that argument is printed in the appropriate font (underlined in *nroff* for **.I**). Then the *previous* font is restored (underlining is turned off in *nroff*). If two or more arguments (maximum 6) are given to a **.B** or **.I**, the second argument is then concatenated to the first with no intervening space (1/12 space if the first font is italic), but is printed in the previous font; and the remaining pairs of arguments are similarly alternated. For example:

```
.I italic " text " right -justified
```

produces:

italic text right-justified

These macros alternate with the prevailing font at the time they are invoked. To alternate specific pairs of fonts, the following macros are available:

```
.IB .BI .IR .RI .RB .BR
```

Each takes a maximum of 6 arguments and alternates the arguments between the specified fonts.

Note that font changes in headings are handled separately {4.2.2.4.1}.

Anyone using a terminal that cannot underline might wish to insert:

```
.rm ul
.rm cu
```

at the beginning of the document to eliminate *all* underlining.

12.2 Justification of Right Margin

```
.SA [arg]
```

The **.SA** macro is used to set right-margin justification for the main body of text. Two justification flags are used: *current* and *default*. **.SA 0** sets both flags to no justification, i.e., it acts like the **.na** request. **.SA 1** is the inverse: it sets both flags to cause justification, just like the **.ad** request. However, calling **.SA** *without* an argument causes the *current* flag to be copied from the *default* flag, thus performing either a **.na** or **.ad**, depending on what the *default* is. Initially, both flags are set for no justification in *nroff* and for justification in *troff*.

In general, the request **.na** can be used to ensure that justification is turned off, but **.SA** should be used to restore justification, rather than the **.ad** request. In this way, justification or lack thereof for the remainder of the text is specified by inserting **.SA 0** or **.SA 1** *once* at the beginning of the document.

12.3 SCCS Release Identification

The string *RE* contains the SCCS^[10] Release and Level of the current version of MM. For example, typing:

```
This is version \*(RE of the macros.
```

produces:

This is version 15.128 of the macros.

This information is useful in analyzing suspected bugs in MM. The easiest way to have this number appear in your output is to specify **-rD1 {2.4}** on the command line, which causes the string *RE* to be output as part of the page header {9.2}.

12.4 Two-Column Output

MM can print two columns on a page:

```
.2C
text and formatting requests (except another .2C)
.1C
```

The .2C macro begins two-column processing which continues until a .1C macro is encountered. In two-column processing, each physical page is thought of as containing two columnar “pages” of equal (but smaller) “page” width. Page headers and footers are *not* affected by two-column processing. The .2C macro does *not* “balance” two-column output.

It is possible to have full-page width footnotes and displays when in two column mode, although the default action is for footnotes and displays to be narrow in two column mode and wide in one column mode. Footnote and display width is controlled by a macro, .WC (Width Control), which takes the following arguments:

<i>arg</i>	<i>Meaning</i>
N	Normal default mode (–WF, –FF, –WD, FB)
WF	Wide Footnotes always (even in two column mode)
–WF	DEFAULT: turn off WF (footnotes follow column mode, wide in 1C mode, narrow in 2C mode, unless FF is set)
FF	First Footnote; all footnotes have the same width as the first footnote encountered for that page
–FF	DEFAULT: turn off FF (footnote style follows the settings of WF or –WF)
WD	Wide Displays always (even in two column mode)
–WD	DEFAULT: Displays follow whichever column mode is in effect when the display is encountered
FB	DEFAULT: Floating displays cause a break when output on the current page
–FB	Floating displays on current page do not cause a break

For example: “.WC WD FF” will cause all displays to be wide, and all footnotes on a page to be the same width, while “.WC N” will reinstate the default actions. If conflicting settings are given to .WC the last one is used. That is, “.WC WF –WF” has the effect of “.WC –WF”.

12.5 Column Headings for Two-Column Output •

✎ This section is intended only for users accustomed to writing formatter macros.

In two-column output, it is sometimes necessary to have headers over each column, as well as headers over the entire page {9}. This is accomplished by redefining the .TP macro {9.12} to provide header lines both for the entire page and for each of the columns. For example:


```

.de TP
.sp 2
.tl `Page \\nP`OVERALL`
.tl ``TITLE``
.sp
.nf
.ta 16C 31R 34 50C 65R
left→center→right→left→center→right      (where → stands for the tab character)
→first column→→→second column
.fi
.sp 2
..

```

The above example will produce two lines of page header text plus two lines of headers over each column. The tab stops are for a 65-en overall line length.

12.6 Vertical Spacing

`.SP [lines]`

There exist several ways of obtaining vertical spacing, all with different effects: the `.sp` request spaces the number of lines specified, *unless* “no space” (`.ns`) mode is on, in which case the request is ignored. This mode is set at the end of a page header to eliminate spacing by a `.sp` or `.bp` request that happens to occur at the top of a page. This mode can be turned *off* by the `.rs` (“restore spacing”) request.

The `.SP` macro is used to avoid the accumulation of vertical space by successive macro calls. Several `.SP` calls in a row produce *not* the sum of their arguments, but their maximum; i.e., the following produces only 3 blank lines:

```

.SP 2
.SP 3
.SP

```

Many MM macros utilize `.SP` for spacing. For example, “.LE 1” {5.3.2} immediately followed by “.P” {4.1} produces only a single blank line (½ a vertical space) between the end of the list and the following paragraph. An omitted argument defaults to one blank line (*one* vertical space). Negative arguments are not permitted. The argument must be unscaled but fractional amounts are permitted. Like `.sp`, `.SP` is also inhibited by the `.ns` request.

12.7 Skipping Pages

`.SK [pages]`

The `.SK` macro skips pages, but retains the usual header and footer processing. If *pages* is omitted, null, or 0, `.SK` skips to the top of the next page *unless* it is currently at the top of a page, in which case it does nothing. `.SK n` skips *n* pages. That is, `.SK` always positions the text that follows it at the top of a page, while `.SK 1` always leaves one page that is blank except for the header and footer.

12.8 Forcing an Odd Page

`.OP`

This macro is used to ensure that the text following it begins at the top of an odd-numbered page. If currently at the top of an odd page, no motion takes place. If currently on an even page, text resumes printing at the top of the next page. If currently on an odd page (but not at the top of the page) one blank page is produced, and printing resumes on the page after that.

12.9 Setting the Point Size and Vertical Spacing

In *troff*, the default point size (obtained from the MM register *S* {2.4}) is 10 points, and the vertical spacing is 12 points (i.e., 6 lines per inch). The prevailing point size and vertical spacing may be changed by invoking the *.S* macro:

```
.S [point size] [vertical spacing]
```

The mnemonics D (default value), C (current value), and P (previous value) may be used for both arguments. If an argument is negative, the current value is decremented by the specified amount; if an argument is positive, the current value is incremented by the specified amount; if an argument is unsigned, it is used as the new value; *.S* without arguments defaults to P. If the first argument is specified but the second is not, then D is used for the vertical spacing; the default value for vertical spacing is always 2 points greater than the *current* point size.¹⁹ A null ("") value for either argument defaults to C. Thus, if *n* is a numeric value:

```
.S          = .S P P
.S "" n     = .S C n
.S n ""     = .S n C
.S n        = .S n D
.S ""       = .S C D
.S "" ""    = .S C C
.S n n      = .S n n
```

If the first argument is greater than 99, the default *point size* (10 points) is restored. If the second argument is greater than 99, the default *vertical spacing* (current point size plus 2 points) is used. For example:

```
.S 100      = .S 10 12
.S 14 111   = .S 14 16
```

The *.SM* macro allows one to reduce by 1 point the size of a string:

```
.SM string1 [string2] [string3]
```

If the third argument is omitted, the *first* argument is made smaller and is concatenated with the second argument, if the latter is specified. If all three arguments are present (even if any are null), the *second* argument is made smaller and all three arguments are concatenated. For example:

```
.SM X          gives X
.SM X Y        gives XY
.SM Y X Y      gives YXY
.SM UNIX       gives UNIX
.SM UNIX )     gives UNIX)
.SM ( UNIX )   gives (UNIX)
.SM U NIX ""   gives UNIX
```

19. Footnotes {8} are two points *smaller* than the body, with an additional three-point space between footnotes.

12.10 Producing Accents

The following strings may be used to produce accents for letters:

	<i>Input</i>	<i>Output</i>
Grave accent	e\ <i>*`</i>	è
Acute accent	e\ <i>*^</i>	é
Circumflex	o\ <i>*^</i>	ô
Tilde	n\ <i>*~</i>	ñ
Cedilla	c\ <i>*,</i>	ç
Lower-case umlaut	u\ <i>*:</i>	ü
Upper-case umlaut	U\ <i>*;</i>	Û

12.11 Inserting Text Interactively •

`.RD [prompt] [diversion] [string]`

`.RD` (ReaD insertion) allows a user to stop the standard output of a document and to read text from the standard input until two consecutive new-lines are found. When the new-lines are encountered, normal output is resumed.

`.RD` follows the formatting conventions in effect. Thus, the examples below assume that the `.RD` is invoked in no fill mode (`.nf`).

The first argument is a *prompt* which will be printed at the terminal. If no prompt is given, `.RD` signals the user with a BEL on terminal output.

The second argument, the name of a *diversion*, allows the user to save all the text typed in after the prompt in a macro whose name is that of the diversion. The third argument, the name of a *string*, allows the user to save for later reference the first line following the prompt in the named string. For example:

```
.RD Name aA bB
```

produces:

```
Name: (user types) J. Jones
16 Elm Rd.,
Piscataway
```

The diverted macro `.aA` will contain:

```
J. Jones
16 Elm Rd.,
Piscataway
```

The string `bB` (`*(bB)`) contains "J. Jones".

A new-line followed by a control-d (EOF) also allows the user to resume normal output.

13. ERRORS AND DEBUGGING

13.1 Error Terminations

When a macro discovers an error, the following actions occur:

- A break occurs.
- To avoid confusion regarding the location of the error, the formatter output buffer (which may contain some text) is printed.

- A short message is printed giving the name of the macro that found the error, the type of error, and the approximate line number (in the current input file) of the last processed input line. (All the error messages are explained in Appendix D.)
- Processing terminates, unless the register *D* {2.4} has a positive value. In the latter case, processing continues even though the output is guaranteed to be deranged from that point on.
- ☞ *The error message is printed by writing it directly to the user's terminal. If an output filter, such as 300(1), 450(1), or hp(1) is being used to post-process nroff output, the message may be garbled by being intermixed with text held in that filter's output buffer.*
- ☞ *If any of cw(1), eqn(1)/neqn, and tbl(1) are being used, and if the -olist option of the formatter causes the last page of the document not to be printed, a harmless "broken pipe" message may result.*

13.2 Disappearance of Output

This usually occurs because of an unclosed diversion (e.g., missing .DE or .FE). Fortunately, the macros that use diversions are careful about it, and they check to make sure that illegal nestings do not occur. If any message is issued about a missing .DE or .FE, the appropriate action is to search backwards from the termination point looking for the corresponding .DF, .DS, or .FS.

The following command:

```
grep -n "\.[EDFRT][EFNQS]" files ...
```

prints all the .DF, .DS, .DE, .EQ, .EN, .FS, .FE, .RS, .RF, .TS, and .TE macros found in *files ...*, each preceded by its file name and the line number in that file. This listing can be used to check for illegal nesting and/or omission of these macros.

14. EXTENDING AND MODIFYING THE MACROS •

14.1 Naming Conventions

In this section, the following conventions are used to describe names:

- n: digit
- a: lower-case letter
- A: upper-case letter
- x: n, a, or A: i.e., any letter or digit (any alphanumeric character)
- s: special character (any non-alphanumeric character)

All other characters are literals (i.e., stand for themselves).

Note that *request*, *macro*, and *string* names are kept by the formatters in a single internal table, so that there must be no duplication among such names. *Number register* names are kept in a separate table.

14.1.1 Names Used by Formatters

- requests:
 - aa (most common)
 - an (only one, currently: c2)
- registers:
 - aa (normal)
 - .x (normal)
 - .s (only one, currently: .s)
 - a. (only one, currently: c.)
 - % (page number)

14.1.2 Names Used by MM

macros and strings: A, AA, Aa (accessible to users; e.g., macros P and HU, strings F, BU, and Lt)
 nA (accessible to users; only two, currently: 1C and 2C)
 aA (accessible to users; only one, currently: nP)
 s (accessible to users; only the seven accents, currently {12.10})
)x, }x,]x, >x, ?x (internal)

registers: An, Aa (accessible to users; e.g., H1, Fg)
 A (accessible to users; meant to be set on the command line; e.g., C)
 :x, ;x, #x, ?x, !x (internal)

14.1.3 Names Used by CW, EQN/NEQN, and TBL. Cw(1), the constant-width font preprocessor for *troff*, uses the following five macro names: .CD, .CN, .CP, .CW, and .PC; it also uses the number register names cE and cW. The equation preprocessors, eqn(1) and neqn use registers and string names of the form nn. The table preprocessor, tbl(1), uses T&, T#, and TW, and names of the form:

a- a+ a| nn na ^a #a #s

14.1.4 User-Definable Names. Given the above, what is left for user extensions? To avoid “collisions” with already used names, use names that consist either of a single lower-case letter, or of a lower-case letter followed by a character *other than* a lower-case letter (remembering, however, that the names .c2 and .nP are already used). The following is a possible user naming convention:

macros: aA (e.g., bG, kW)
 strings: as (e.g., c), f], p}
 registers: a (e.g., f, t)

14.2 Sample Extensions

14.2.1 Appendix Headings. The following is a way of generating and numbering appendix headings:

```
.nr Hu 1
.nr a 0
.de aH
.nr a +1
.nr P 0
.PH "^^^ Appendix \\na-\\\\\\\\\\\\nP"
.SK
.HU "\\$1"
..
```

After the above initialization and definition, each call of the form “.aH “title”” begins a new page (with the page header changed to “Appendix a-n”) and generates an unnumbered heading of *title*, which, if desired, can be saved for the table of contents. Those who wish appendix titles to be centered must, in addition, set the register Hc to 1 {4.2.2.3}.

14.2.2 Hanging Indent with Tabs. The following example illustrates the use of the hanging-indent feature of variable-item lists {5.3.3.6}. First, a user-defined macro is built to accept four arguments that make up the *mark*. In the output, each argument is to be separated from the previous one by a tab; tab settings are defined later. Since the first argument may begin with a period or apostrophe, the “&” is used so that the formatter will not interpret such a line as a formatter request or macro call.²⁰ The “\t” is translated by the formatter into a tab. The “c” is used to concatenate the input *text* that follows the macro call to the line built by the macro. The macro and an example of its use are:

20. The two-character sequence “&” is understood by the formatters to be a “zero-width” space, i.e., it causes no output characters to appear, but it removes the special meaning of a leading period or apostrophe.

```
.de aX
.LI
\&\$1\t\\$2\t\\$3\t\\$4\t\c
..
:
.ta 9n 18n 27n 36n
.VL 36
.aX .nh off \- no
No hyphenation.
Automatic hyphenation is turned off.
Words containing hyphens
(e.g., mother-in-law) may still be split across lines.
.aX .hy on \- no
Hyphenate.
Automatic hyphenation is turned on.
.aX .hc\c none none no
.LE
```

(□ stands for a space)

The resulting output is:

.nh	off	—	no	No hyphenation. Automatic hyphenation is turned off. Words containing hyphens (e.g., mother-in-law) may still be split across lines.
.hy	on	—	no	Hyphenate. Automatic hyphenation is turned on.
.hc c	none	none	no	Hyphenation indicator character is set to “c” or removed. During text processing the indicator is suppressed and will not appear in the output. Prepending the indicator to a word has the effect of preventing hyphenation of that word.

15. CONCLUSION

The following are the qualities that we have tried to emphasize in MM, in approximate order of importance:

- *Robustness in the face of error*—A user need not be an *nroff/troff* expert to use these macros. When the input is incorrect, either the macros attempt to make a reasonable interpretation of the error, or a message describing the error is produced. We have tried to minimize the possibility that a user would get cryptic system messages or strange output as a result of simple errors.
- *Ease of use for simple documents*—It is not necessary to write complex sequences of commands to produce simple documents. Reasonable default values are provided, where at all possible.
- *Parameterization*—There are many different preferences in the area of document styling. Many parameters are provided so that users can adapt the output to their respective needs over a wide range of styles.
- *Extension by moderately expert users*—We have made a strong effort to use mnemonic naming conventions and consistent techniques in the construction of the macros. Naming conventions are given so that a user can add new macros or redefine existing ones, if necessary.

- *Device independence*—The most common use of MM is to print documents on hard-copy typewriter terminals, using the *nroff* formatter. The macros can be used conveniently with both 10- and 12-pitch terminals. In addition, output can be scanned with an appropriate CRT terminal. The macros have been constructed to allow compatibility with *troff*, so that output can be produced both on typewriter-like terminals and on a phototypesetter.
- *Minimization of input*—The design of the macros attempts to minimize repetitive typing. For example, if a user wants to have a blank line after all first- or second-level headings, he or she need only set a specific parameter *once* at the beginning of a document, rather than add a blank line after each such heading.
- *Decoupling of input format from output style*—There is but one way to prepare the input text, although the user may obtain a number of output styles by setting a few global flags. For example, the .H macro is used for all numbered headings, yet the actual output style of these headings may be made to vary from document to document or, for that matter, within a single document.

Acknowledgements. We are indebted to T. A. Dolotta for his continuing guidance during the development of MM. We also thank our many users who have provided much valuable feedback, both about the macros and about this manual. Many of the features of MM are patterned after similar features in a number of earlier macro packages, and, in particular, after one implemented by M. E. Lesk. Finally, because MM often approaches the limits of what is possible with the text formatters, during the implementation of MM we have generated atypical requirements and encountered unusual problems; we thank the late J. F. Ossanna for his willingness to add new features to the formatters and to invent ways of having the formatters perform unusual but desired actions.

REFERENCES

1. Dolotta, T. A., Olsson, S. B., and Petruccelli, A. G. (eds.). *UNIX User's Manual*—Release 3.0, Bell Laboratories (June 1980).
2. Ossanna, J. F. *NROFF/TROFF User's Manual*. Bell Laboratories, October 1976.
3. Kernighan, B. W. *UNIX for Beginners*. Bell Laboratories, October 1974.
4. Kernighan, B. W. *A Tutorial Introduction to the UNIX Text Editor*. Bell Laboratories, October 1974.
5. Kernighan, B. W. *A TROFF Tutorial*. Bell Laboratories, August 1976.
6. Kernighan, B. W. and Cherry, L. L. *Typesetting Mathematics—User's Guide (Second Edition)*. Bell Laboratories, August 1978.
7. Lesk, M. *TBL—A Program to Format Tables*. Bell Laboratories, September 1977.
8. Smith, D. W. and Piskorik, E. M. *Typing Documents with MM*. Bell Laboratories, April 1980.
9. Bell Laboratories Methods and Systems Department. *Office Guide*. Unpublished Memorandum, Bell Laboratories, April 1972 (as revised).
10. Bonanni, L. E. and Salemi, C. A. *The Source Code Control System User's Guide*. Bell Laboratories, April 1979.

Appendix A: USER-DEFINED LIST STRUCTURES •

This appendix is intended only for users accustomed to writing formatter macros.

If a large document requires complex list structures, it is useful to be able to define the appearance for each list level only once, instead of having to define it at the beginning of each list. This permits consistency of style in a large document. For example, a generalized list-initialization macro might be defined in such a way that what it does depends on the list-nesting level in effect at the time the macro is called. Suppose that levels 1 through 5 of lists are to have the following appearance:

```
A.
  [1]
    •
    a)
      +
```

The following code defines a macro (.aL) that always begins a new list and determines the type of list according to the current list level. To understand it, you should know that the number register :g is used by the MM list macros to determine the current list level; it is 0 if there is no currently active list. Each call to a list-initialization macro increments :g, and each .LE call decrements it.

```
.de aL
  "\      register g is used as a local temporary to save :g before it is changed below
.nr g \n(:g
.if \ng=0 .AL A \" give me an A.
.if \ng=1 .LB \n(Li 0 1 4 \" give me a [1]
.if \ng=2 .BL \" give me a bullet
.if \ng=3 .LB \n(Li 0 2 2 a \" give me an a)
.if \ng=4 .ML + \" give me a +
..
```

This macro can be used (in conjunction with .LI and .LE) instead of .AL, .RL, .BL, .LB, and .ML. For example, the following input:

```
.aL
.LI
first line.
.aL
.LI
second line.
.LE
.LI
third line.
.LE
```

will yield:

```
A. first line.
  [1] second line.
B. third line.
```

There is another approach to lists that is similar to the .H mechanism. The list-initialization, as well as the .LI and the .LE macros are all included in a single macro. That macro (called .bL below) requires

an argument to tell it what level of item is required; it adjusts the list level by either beginning a new list or setting the list level back to a previous value, and then issues a `.LI` macro call to produce the item:

```
.de bL
.ie \n(. $ .nr g \\\$1 \" if there is an argument, that is the level
.el .nr g \n(:g \" if no argument, use current level
.if \ng-\n(:g>1 .)D \"**ILLEGAL SKIPPING OF LEVEL\" \" increasing level by more than 1
.if \ng>\n(:g \{.aL \ng-1 \" if g > :g, begin new list
.   nr g \n(:g) \" and reset g to current level (.aL changes g)
.if \n(:g>\ng .LC \ng \" if :g > g, prune back to correct level
.\"   if :g = g, stay within current list
.LI \" in all cases, get out an item
..
```

For `.bL` to work, the previous definition of the `.aL` macro must be changed to obtain the value of `g` from its argument, rather than from `:g`. Invoking `.bL` without arguments causes it to stay at the current list level. The MM `.LC` macro (List Clear) removes list descriptions until the level is less than or equal to that of its argument. For example, the `.H` macro includes the call `“.LC 0”`. If text is to be resumed at the end of a list, insert the call `“.LC 0”` to clear out the lists completely. The example below illustrates the relatively small amount of input needed by this approach. The input text:

```
The quick brown fox jumped over the lazy dog's back.
.bL 1
first line.
.bL 2
second line.
.bL 1
third line.
.bL
fourth line.
.LC 0
fifth line.
```

yields:

The quick brown fox jumped over the lazy dog's back.

A. first line.

[1] second line.

B. third line.

C. fourth line.

fifth line.

Appendix B: SAMPLE FOOTNOTES

The following example illustrates several footnote styles and both labeled and automatically-numbered footnotes. The actual input for the immediately following text and for the footnotes at the bottom of this page is shown on the following page:

With the footnote style set to the *nroff* default, we process a footnote¹ followed by another one.***** Using the .FD macro, we changed the footnote style to hyphenate, right margin justification, indent, and left justify the label. Here is a footnote,² and another.† The footnote style is now set, again via the .FD macro, to no hyphenation, no right margin justification, no indentation, and with the label left-justified. Here comes the final one.³

1. This is the first footnote text example (.FD 10). This is the default style for *nroff*. The right margin is *not* justified. Hyphenation is *not* permitted. The text is indented, and the automatically generated label is *right*-justified in the text-indent space.

***** This is the second footnote text example (.FD 10). This is also the default *nroff* style but with a long footnote label provided by the user.

2. This is the third footnote example (.FD 1). The right margin is justified, the footnote text is indented, the label is *left*-justified in the text-indent space. Although not necessarily illustrated by this example, hyphenation is permitted. The quick brown fox jumped over the lazy dog's back.

† This is the fourth footnote example (.FD 1). The style is the same as the third footnote.

3. This is the fifth footnote example (.FD 6). The right margin is *not* justified, hyphenation is *not* permitted, the footnote text is *not* indented, and the label is placed at the beginning of the first line. The quick brown fox jumped over the lazy dog's back. Now is the time for all good men to come to the aid of their country.

```
.FD 10
With the footnote style set to the
.I nroff
default, we process a footnote\*F
.FS
This is the first footnote text example (.FD 10).
This is the default style for
.I nroff.
The right margin is
.I not
justified.
Hyphenation is
.I not
permitted.
The text is indented, and the automatically generated label is
.I right -justified
in the text-indent space.
.FE
followed by another one.*****\□ (□ stands for a space)
.FS *****
This is the second footnote text example (.FD 10).
This is also the default
.I nroff
style but with a long footnote label provided by the user.
.FE
.FD 1
Using the .FD macro, we changed the footnote style to
hyphenate, right margin justification, indent, and left justify the label.
Here is a footnote.\*F
.FS
This is the third footnote example (.FD 1).
The right margin is justified,
the footnote text is indented, the label is
.I left -justified
in the text-indent space.
Although not necessarily illustrated by this example, hyphenation is permitted.
The quick brown fox jumped over the lazy dog's back.
.FE
and another.\(dg\□
.FS \(dg
This is the fourth footnote example (.FD 1).
The style is the same as the third footnote.
.FE
.FD 6
The footnote style is now set, again via the .FD macro, to no hyphenation, no right margin justification,
no indentation, and with the label left-justified.
Here comes the final one.\*F\□
.FS
This is the fifth footnote example (.FD 6).
The right margin is
.I not
justified, hyphenation is
.I not
permitted, the footnote text is
.I not
indented, and the label is placed at the beginning of the first line.
The quick brown fox jumped over the lazy dog's back.
Now is the time for all good men to come to the aid of their country.
.FE
```

Appendix C: SAMPLE LETTER

~~var~~ The nroff and troff outputs corresponding to the input text below are shown on the following pages.

.ND "May 31, 1979"
 .TL 334455
 Out-of-Hours Course Description
 .AU "D. W. Stevenson" DWS PY 9876 5432 1X-123
 .MT 0
 .DS
 J. M. Jones:
 .DE
 .P
 Please use the following description for the Out-of-Hours course
 .I
 Document Preparation on the UNIX*
 .R
 .FS *
 UNIX is a trademark of Bell Laboratories.
 .FE
 .I "Time-Sharing System:"
 .P
 The course is intended for clerks, typists, and others
 who intend to use the UNIX system for preparing documentation.
 The course will cover such topics as:
 .VL 18
 .LI Environment:
 utilizing a time-sharing computer system;
 accessing the system; using appropriate output terminals.
 .LI Files:
 how text is stored on the system;
 directories; manipulating files.
 .LI "Text editing:"
 how to enter text so that subsequent revisions are easier to make;
 how to use the editing system to add, delete, and move lines of text;
 how to make corrections.
 .LI "Text processing:"
 basic concepts;
 use of general-purpose formatting packages.
 .LI "Other facilities:"
 additional capabilities useful to the typist such as the
 .I "spell, diff,"
 and
 .I grep
 commands, and a desk-calculator package.
 .LE
 .SG jrm
 .NS
 S. P. Lename
 H. O. Del
 M. Hill
 .NE

Bell Laboratories

subject: Out-of-Hours Course Description -
Case 334455

date: May 31, 1979

from: D. W. Stevenson
PY 9876
1X-123 x5432

J. M. Jones:

Please use the following description for the Out-of-Hours course
Document Preparation on the UNIX* Time-Sharing System:

The course is intended for clerks, typists, and others who intend to use the UNIX system for preparing documentation. The course will cover such topics as:

Environment: utilizing a time-sharing computer system; accessing the system; using appropriate output terminals.

Files: how text is stored on the system; directories; manipulating files.

Text editing: how to enter text so that subsequent revisions are easier to make; how to use the editing system to add, delete, and move lines of text; how to make corrections.

Text processing: basic concepts; use of general-purpose formatting packages.

Other facilities: additional capabilities useful to the typist such as the spell, diff, and grep commands, and a desk-calculator package.

PY-9876-DWS-jrm

D. W. Stevenson

Copy to
S. P. Lename
H. O. Del
M. Hill

* UNIX is a trademark of Bell Laboratories.

**Bell Laboratories**

subject: **Out-of-Hours Course Description - Case 334455**

date: **May 31, 1979**

from: **D. W. Stevenson**
PY 9876
1X-123 x5432

J. M. Jones:

Please use the following description for the Out-of-Hours course *Document Preparation on the UNIX* Time-Sharing System*:

The course is intended for clerks, typists, and others who intend to use the UNIX system for preparing documentation. The course will cover such topics as:

Environment: utilizing a time-sharing computer system; accessing the system; using appropriate output terminals.

Files: how text is stored on the system; directories; manipulating files.

Text editing: how to enter text so that subsequent revisions are easier to make; how to use the editing system to add, delete, and move lines of text; how to make corrections.

Text processing: basic concepts; use of general-purpose formatting packages.

Other facilities: additional capabilities useful to the typist such as the *spell*, *diff*, and *grep* commands, and a desk-calculator package.

PY-9876-DWS-jrm

D. W. Stevenson

Copy to
S. P. Lename
H. O. Del
M. Hill

* UNIX is a trademark of Bell Laboratories.

Appendix D: ERROR MESSAGES

I. MM Error Messages

An MM error message has a standard part followed by a variable part. The standard part has the form:

ERROR:(*filename*)input line *n*:

The variable part consists of a descriptive message, usually beginning with a macro name. The variable parts are listed below in alphabetical order by macro name, each with a more complete explanation.²¹

Check TL, AU, AS, AE, MT sequence	The correct order of macros at the start of a memorandum is shown in {6.9}. Something has disturbed this order. If .AS 2 was used, then the error message will be "Check TL, AU, AS, AE, NS, NE, MT sequence".
CS:cover sheet too long	The text of the cover sheet is too long to fit on one page. The abstract should be reduced or the indent of the abstract should be decreased {6.4}.
DE:no DS or DF active	.DE has been encountered but there has not been a previous .DS or .DF to match it.
DF:illegal inside TL or AS	Displays are not allowed in the title or abstract.
DF:missing DE	.DF occurs within a display, i.e., a .DE has been omitted or mistyped.
DF:missing FE	A display starts inside a footnote. The likely cause is the omission (or misspelling) of a .FE to end a previous footnote.
DF:too many displays	More than 26 floating displays are active at once, i.e., have been accumulated but not yet output.
DS:illegal inside TL or AS	Displays are not allowed in the title or abstract.
DS:missing DE	.DS occurs within a display, i.e., a .DE has been omitted or mistyped.
DS:missing FE	A display starts inside a footnote. The likely cause is the omission (or misspelling) of a .FE to end a previous footnote.
FE:no FS active	.FE has been encountered with no previous .FS to match it.
FS:missing DE	A footnote starts inside a display, i.e., a .DS or .DF occurs without a matching .DE.
FS:missing FE	A previous .FS was not matched by a closing .FE, i.e., an attempt is being made to begin a footnote inside another one.
H:bad arg:value	The first argument to .H must be a single digit from 1 to 7, but <i>value</i> has been supplied instead.
H:missing arg	.H needs at least 1 argument.
H:missing DE	A heading macro (.H or .HU) occurs inside a display.
H:missing FE	A heading macro (.H or .HU) occurs inside a footnote.

21. This list is set up by ".LB 37 0 2 0" {5.4}.

HU:missing arg	.HU needs 1 argument.
LB:missing arg(s)	.LB requires at least 4 arguments.
LB:too many nested lists	Another list was started when there were already 6 active lists.
LE:mismatched	.LE has occurred without a previous .LB or other list-initialization macro {5.3.3}. Although this is not a fatal error, the message is issued because there almost certainly exists some problem in the preceding text.
LI:no lists active	.LI occurs without a preceding list-initialization macro. The latter has probably been omitted, or has been separated from the .LI by an intervening .H or .HU.
ML:missing arg	.ML requires at least 1 argument.
ND:missing arg	.ND requires 1 argument.
RF:no RS active	.RF has been encountered with no previous .RS to match it.
RP:missing RF	A previous .RS was not matched by a closing .RF.
RS:missing RF	A previous .RS was not matched by a closing .RF.
S:bad arg:value	The incorrect argument <i>value</i> has been given for .S, see {12.9}.
SA:bad arg:value	The argument to .SA (if any) must be either 0 or 1. The incorrect argument is shown as <i>value</i> .
SG:missing DE	.SG occurs inside a display.
SG:missing FE	.SG occurs inside a footnote.
SG:no authors	.SG occurs without any previous .AU macro(s).
VL:missing arg	.VL requires at least 1 argument.
WC:unknown option	An incorrect argument has been given to .WC, see {12.4}.

II. Formatter Error Messages

Most messages issued by the formatter are self-explanatory. Those error messages over which the *user* has (some) control are listed below. Any other error messages should be reported to the local system-support group.

- “Cannot do ev” is caused by (a) setting a page width that is negative or extremely short; (b) setting a page length that is negative or extremely short; (c) reprocessing a macro package (e.g., performing a .so formatter request on a macro package that was already requested on the command line); and (d) requesting the *troff* -s1 option on a document that is longer than ten pages.
- “Cannot execute *filename*” is given by the .! request if it cannot find the *filename*.
- “Cannot open *filename*” is issued if one of the files in the list of files to be processed cannot be opened.
- “Exception word list full” indicates that too many words have been specified in the hyphenation exception list (via .hw requests).
- “Line overflow” means that the output line being generated was too long for the formatter’s line buffer. The excess was discarded. See the “Word overflow” message below.
- “Non-existent font type” means that a request has been made to mount an unknown font.
- “Non-existent macro file” means that the requested macro package does not exist.
- “Non-existent terminal type” means that the terminal options refers to an unknown terminal type.
- “Out of temp file space” means that additional temporary space for macro definitions, diversions, etc. cannot be allocated. This message often occurs because of unclosed diversions (missing .FE or .DE), unclosed macro definitions (e.g., missing “.”), or a huge table of contents.
- “Too many page numbers” is issued when the list of pages specified to the formatter -o option is too long.
- “Too many number registers” means that the pool of number register names is full. Unneeded registers can be deleted by using the .rr request.
- “Too many string/macro names” is issued when the pool of string and macro names is full. Unneeded strings and macros can be deleted using the .rm request.
- “Word overflow” means that a word being generated exceeded the formatter’s word buffer. The excess characters were discarded. A likely cause for this and for the “Line overflow” message above are very long lines or words generated through the misuse of \c or of the .cu request, or very long equations produced by *eqn*(1)/*neqn*.

Appendix E: SUMMARY OF MACROS, STRINGS, AND NUMBER REGISTERS

I. Macros

The following is an alphabetical list of macro names used by MM. The first line of each item gives the name of the macro, a brief description, and a reference to the section in which the macro is described. The second line gives a prototype call of the macro.

Macros marked with an asterisk are *not*, in general, invoked directly by the user. Rather, they are “user exits” *defined* by the user and called by the MM macros from inside header, footer, or other macros.

1C	One-column processing {12.4} .1C
2C	Two-column processing {12.4} .2C
AE	Abstract end {6.4} .AE
AF	Alternate format of “Subject/Date/From” block {6.7.2} .AF [company-name]
AL	Automatically-incremented list start {5.3.3.1} .AL [type] [text-indent] [1]
AS	Abstract start {6.4} .AS [arg] [indent]
AT	Author’s title {6.2} .AT [title] ...
AU	Author information {6.2} .AU name [initials] [loc] [dept] [ext] [room] [arg] [arg] [arg]
AV	Approval signature {6.11.3} .AV [name]
B	Bold {12.1} .B [bold-arg] [previous-font-arg] [bold] [prev] [bold] [prev]
BE	Bottom Block End {9.13} .BE
BI	Bold/Italic {12.1} .BI [bold-arg] [italic-arg] [bold] [italic] [bold] [italic]
BL	Bullet list start {5.3.3.2} .BL [text-indent] [1]
BR	Bold/Roman {12.1} .BR [bold-arg] [Roman-arg] [bold] [Roman] [bold] [Roman]
BS	Bottom Block Start {9.13} .BS
CS	Cover sheet {10.2} .CS [pages] [other] [total] [figs] [tbls] [refs]
DE	Display end {7.1} .DE

DF	Display floating start {7.2} .DF [format] [fill] [right-indent]
DL	Dash list start {5.3.3.3} .DL [text-indent] [1]
DS	Display static start {7.1} .DS [format] [fill] [right-indent]
EC	Equation caption {7.5} .EC [title] [override] [flag]
EF	Even-page footer {9.6} .EF [arg]
EH	Even-page header {9.3} .EH [arg]
EN	End equation display {7.4} .EN
EQ	Equation display start {7.4} .EQ [label]
EX	Exhibit caption {7.5} .EX [title] [override] [flag]
FC	Formal closing {6.11} .FC [closing]
FD	Footnote default format {8.3} .FD [arg] [1]
FE	Footnote end {8.2} .FE
FG	Figure title {7.5} .FG [title] [override] [flag]
FS	Footnote start {8.2} .FS [label]
H	Heading—numbered {4.2} .H level [heading-text] [heading-suffix]
HC	Hyphenation character {3.4} .HC [hyphenation-indicator]
HM	Heading mark style (Arabic or Roman numerals, or letters) {4.2.2.5} .HM [arg1] ... [arg7]
HU	Heading—unnumbered {4.3} .HU heading-text
HX *	Heading user exit X (before printing heading) {4.6} .HX dlevel rlevel heading-text
HY *	Heading user exit Y (before printing heading) {4.6} .HY dlevel rlevel heading-text
HZ *	Heading user exit Z (after printing heading) {4.6} .HZ dlevel rlevel heading-text

I	Italic (underline in <i>nroff</i>) {12.1} .I [italic-arg] [previous-font-arg] [italic] [prev] [italic] [prev]
IB	Italic/Bold {12.1} .IB [italic-arg] [bold-arg] [italic] [bold] [italic] [bold]
IR	Italic/Roman {12.1} .IR [italic-arg] [Roman-arg] [italic] [Roman] [italic] [Roman]
LB	List begin {5.4} .LB text-indent mark-indent pad type [mark] [LI-space] [LB-space]
LC	List-status clear {Appendix A} .LC [list-level]
LE	List end {5.3.2} .LE [1]
LI	List item {5.3.1} .LI [mark] [1]
ML	Marked list start {5.3.3.4} .ML mark [text-indent] [1]
MT	Memorandum type {6.6} .MT [type] [addressee] or .MT [4] [1]
ND	New date {6.7.1} .ND new-date
NE	Notation end {6.11.2} .NE
NS	Notation start {6.11.2} .NS [arg]
nP	Double-line indented paragraphs {4.1} .nP
OF	Odd-page footer {9.7} .OF [arg]
OH	Odd-page header {9.4} .OH [arg]
OK	Other keywords for the Technical Memorandum cover sheet {6.5} .OK [keyword] ...
OP	Odd page {12.8} .OP
P	Paragraph {4.1} .P [type]
PF	Page footer {9.5} .PF [arg]
PH	Page header {9.2} .PH [arg]
PM	Proprietary Marking {9.15} .PM [code]

PX * Page-header user exit {9.12}
.PX

R Return to regular (roman) font (end underlining in *nroff*) {12.1}
.R

RB Roman/Bold {12.1}
.RB [Roman-arg] [bold-arg] [Roman] [bold] [Roman] [bold]

RD Read insertion from terminal {12.11}
.RD [prompt] [diversion] [string]

RF Reference end {11.2}
.RF

RI Roman/Italic {12.1}
.RI [Roman-arg] [italic-arg] [Roman] [italic] [Roman] [italic]

RL Reference list start {5.3.3.5}
.RL [text-indent] [1]

RP Produce Reference Page {11.4}
.RP [arg] [arg]

RS Reference start {11.2}
.RS [string-name]

S Set *troff* point size and vertical spacing {12.9}
.S [size] [spacing]

SA Set adjustment (right-margin justification) default {12.2}
.SA [arg]

SG Signature line {6.11.1}
.SG [arg] [1]

SK Skip pages {12.7}
.SK [pages]

SM Make a string smaller {12.9}
.SM string1 [string2] [string3]

SP Space vertically {12.6}
.SP [lines]

TB Table title {7.5}
.TB [title] [override] [flag]

TC Table of contents {10.1}
.TC [slevel] [spacing] [tlevel] [tab] [head1] [head2] [head3] [head4] [head5]

TE Table end {7.3}
.TE

TH Table header {7.3}
.TH [N]

TL Title of memorandum {6.1}
.TL [charging-case] [filing-case]

TM Technical Memorandum number(s) {6.3}
.TM [number] ...

TP *	Top-of-page macro {9.12} .TP
TS	Table start {7.3} .TS [H]
TX *	Table-of-contents user exit {10.1} .TX
TY *	Table-of-contents user exit (suppresses "CONTENTS") {10.1} .TY
VL	Variable-item list start {5.3.3.6} .VL text-indent [mark-indent] [1]
VM	Vertical margins {9.14} .VM [top] [bottom]
WC	Width Control {12.4} .WC [format]

II. Strings

The following is an alphabetical list of string names used by MM, giving for each a brief description, section reference, and initial (default) value(s). See {1.4} for notes on setting and referencing strings.

BU	Bullet {3.7} <i>nroff</i> : ⊕ <i>troff</i> : ●
Ci	Table-of-contents indent list, up to seven args for heading levels (must be scaled) {10.1}
DT	Date (current date, unless overridden) {6.7.1} Month day, year (e.g., March 27, 1981)
EM	Em dash string, produces an em dash in <i>troff</i> and a double hyphen in <i>nroff</i> {3.8}.
F	Footnote numberer {8.1} <i>nroff</i> : \u\\n+(:p\d <i>troff</i> : \v~-.4m^s-3\\n+(:p\s0\v~.4m^
HF	Heading font list, up to seven codes for heading levels 1 through 7 {4.2.2.4.1} 3 3 2 2 2 2 2 (levels 1 and 2 bold, 3-7 underlined in <i>nroff</i> , and italic in <i>troff</i>)
HP	Heading point size list, up to seven codes for heading levels 1 through 7 {4.2.2.4.3}
Le	Title for LIST OF EQUATIONS {7.6}
Lf	Title for LIST OF FIGURES {7.6}
Lt	Title for LIST OF TABLES {7.6}
Lx	Title for LIST OF EXHIBITS {7.6}
RE	SCCS Release and Level of MM {12.3} Release.Level (e.g., 15.128)
Rf	Reference numberer {11.1}
Rp	Title for References {11.4}
Tm	Trademark string; places the letters "TM" one half-line above the text that it follows {3.9}.

Seven accent strings are also available {12.10}.

If the released-paper style is used, then, in addition to the above strings, certain BTL location codes are defined as strings; these location strings are needed only until the .MT macro is called {6.8}. Currently, the following are recognized: AK, AL, ALF, CB, CH, CP, DR, FJ, HL, HO, HOH, HP, IH, IN, INH, IW, MH, MV, PY, RD, RR, WB, WH, and WV.

III. Number Registers

This section provides an alphabetical list of register names, giving for each a brief description, section reference, initial (default) value, and the legal range of values (where [m:n] means values from m to n inclusive).

Any register having a single-character name can be set from the command line. An asterisk attached to a register name indicates that that register can be set *only* from the command line or *before* the MM macro definitions are read by the formatter {2.4, 2.5}. See {1.4} for notes on setting and referencing registers.

- A * Handles preprinted forms and the Bell System logo {2.4}
0, [0:2]
- Au Inhibits printing of author's location, department, room, and extension in the "from" portion of a memorandum {6.2}
1, [0:1]
- C * Copy type (Original, DRAFT, etc.) {2.4}
0 (Original), [0:4]
- Cl Contents level (i.e., level of headings saved for table of contents) {4.4}
2, [0:7]
- Cp Placement of List of Figures, etc. {10.1}
1 (on separate pages), [0:1]
- D * Debug flag {2.4}
0, [0:1]
- De Display eject register for floating displays {7.2}
0, [0:1]
- Df Display format register for floating displays {7.2}
5, [0:5]
- Ds Static display pre- and post-space {7.1}
1, [0:1]
- E * Controls font of the Subject/Date/From fields {2.4}
0 (*nroff*) 1 (*troff*), [0:1]
- Ec Equation counter, used by .EC macro {7.5}
0, [0:?], incremented by 1 for each .EC call.
- Ej Page-ejection flag for headings {4.2.2.1}
0 (no eject), [0:7]
- Eq Equation label placement {7.4}
0 (right-adjusted), [0:1]
- Ex Exhibit counter, used by .EX macro {7.5}
0, [0:?], incremented by 1 for each .EX call.
- Fg Figure counter, used by .FG macro {7.5}
0, [0:?], incremented by 1 for each .FG call.

- Fs Footnote space (i.e., spacing between footnotes) {8.4}
1, [0:?]
- H1-H7 Heading counters for levels 1-7 {4.2.2.5}
0, [0:?], incremented by .H of corresponding level or .HU if at level given by register *Hu*.
H2-H7 are reset to 0 by any heading at a lower-numbered level.
- Hb Heading break level (after .H and .HU) {4.2.2.2}
2, [0:7]
- Hc Heading centering level for .H and .HU {4.2.2.3}
0 (no centered headings), [0:7]
- Hi Heading temporary indent (after .H and .HU) {4.2.2.2}
1 (indent as paragraph), [0:2]
- Hs Heading space level (after .H and .HU) {4.2.2.2}
2 (space only after .H 1 and .H 2), [0:7]
- Ht Heading type (for .H: single or concatenated numbers) {4.2.2.5}
0 (concatenated numbers: 1.1.1, etc.), [0:1]
- Hu Heading level for unnumbered heading (.HU) {4.3}
2 (.HU at the same level as .H 2), [0:7]
- Hy Hyphenation control for body of document {3.4}
0 (automatic hyphenation off), [0:1]
- L * Length of page {2.4}
66, [20:?] (11i, [2i:?] in *troff*)²²
- Lc List of Equations {7.6}
0 (list not produced) [0:1]
- Lf List of Figures {7.6}
1 (list produced) [0:1]
- Li List indent {5.3.3.1}
6 (*nroff*) 5 (*troff*), [0:?]
- Ls List spacing between items by level {5.3.3.1}
6 (spacing between all levels) [0:6]
- Lt List of Tables {7.6}
1 (list produced) [0:1]
- Lx List of Exhibits {7.6}
1 (list produced) [0:1]
- N * Numbering style {2.4}
0, [0:5]
- Np Numbering style for paragraphs {4.1}
0 (unnumbered) [0:1]
- O * Offset of page {2.4}
.75i, [0:?] (0.5i, [0i:?] in *troff*)²²

22. For *nroff*, these values are *unscaled* numbers representing lines or character positions; for *troff*, these values must be *scaled*.

- Oc Table of Contents page numbering style {10.1}
0 (lower-case Roman), [0:1]
- Of Figure caption style {7.5}
0 (period separator), [0:1]
- P Page number, managed by MM {2.4}
0, [0:?]
- Pi Paragraph indent {4.1}
5 (*nroff*) 3 (*troff*), [0:?]
- Ps Paragraph spacing {4.1}
1 (one blank space between paragraphs), [0:?]
- Pt Paragraph type {4.1}
0 (paragraphs always left-justified), [0:2]
- Pv "PRIVATE" header {9.16}
0 (not printed), [0:2]
- Rf Reference counter, used by .RS macro {11.1}
0, [0:?], incremented by 1 for each .RS call.
- S * *Troff* default point size {2.4}
10, [6:36]
- Si Standard indent for displays {7.1}
5 (*nroff*) 3 (*troff*), [0:?]
- T * Type of *nroff* output device {2.4}
0, [0:2]
- Tb Table counter, used by .TB macro {7.5}
0, [0:?], incremented by 1 for each .TB call.
- U * Underlining style (*nroff*) for .H and .HU {2.4}
0 (continuous underline when possible), [0:1]
- W * Width of page (line and title length) {2.4}
6i, [10:1365] (6i, [2i:7.54i] in *troff*)²³

January 1981

23. For *nroff*, these values are *unscaled* numbers representing lines or character positions; for *troff*, these values must be *scaled*.

MM—Memorandum Macros

TABLE OF CONTENTS

1. INTRODUCTION	1
1.1 Purpose 1	
1.2 Conventions 1	
1.3 Overall Structure of a Document 2	
1.4 Definitions 2	
1.5 Prerequisites and Further Reading 3	
2. INVOKING THE MACROS	3
2.1 The mm Command 3	
2.2 The -cm or -mm Flag 4	
2.3 Typical Command Lines 4	
2.4 Parameters that Can Be Set from the Command Line 5	
2.5 Omission of -cm or -mm 7	
3. FORMATTING CONCEPTS	7
3.1 Basic Terms 7	
3.2 Arguments and Double Quotes 7	
3.3 Unpaddable Spaces 8	
3.4 Hyphenation 8	
3.5 Tabs 9	
3.6 Special Use of the BEL Character 9	
3.7 Bullets 9	
3.8 Dashes, Minus Signs, and Hyphens 9	
3.9 Trademark String 10	
3.10 Use of Formatter Requests • 10	
4. PARAGRAPHS AND HEADINGS	10
4.1 Paragraphs 10	
4.2 Numbered Headings 11	
4.3 Unnumbered Headings 14	
4.4 Headings and the Table of Contents 14	
4.5 First-Level Headings and Page Numbering Style 14	
4.6 User Exit Macros • 15	
4.7 Hints for Large Documents 16	
5. LISTS	16
5.1 Basic Approach 16	
5.2 Sample Nested Lists 17	
5.3 Basic List Macros 18	
5.4 List-Begin Macro and Customized Lists • 21	
6. MEMORANDUM AND RELEASED PAPER STYLES	22
6.1 Title 22	
6.2 Author(s) 23	
6.3 TM Number(s) 24	
6.4 Abstract 24	
6.5 Other Keywords 24	
6.6 Memorandum Types 24	
6.7 Date and Format Changes 25	
6.8 Released-Paper Style 26	
6.9 Order of Invocation of “Beginning” Macros 26	
6.10 Example 27	

6.11	Macros for the End of a Memorandum	27
6.12	Forcing a One-Page Letter	29
7.	DISPLAYS	29
7.1	Static Displays	29
7.2	Floating Displays	30
7.3	Tables	32
7.4	Equations	33
7.5	Figure, Table, Equation, and Exhibit Captions	33
7.6	List of Figures, Tables, Equations, and Exhibits	33
8.	FOOTNOTES	34
8.1	Automatic Numbering of Footnotes	34
8.2	Delimiting Footnote Text	34
8.3	Format of Footnote Text	35
8.4	Spacing between Footnote Entries	35
9.	PAGE HEADERS AND FOOTERS	36
9.1	Default Headers and Footers	36
9.2	Page Header	36
9.3	Even-Page Header	36
9.4	Odd-Page Header	36
9.5	Page Footer	36
9.6	Even-Page Footer	37
9.7	Odd-Page Footer	37
9.8	Footer on the First Page	37
9.9	Default Header and Footer with "Section-Page" Numbering	37
9.10	Use of Strings and Registers in Header and Footer Macros	37
9.11	Header and Footer Example	37
9.12	Generalized Top-of-Page Processing	38
9.13	Generalized Bottom-of-Page Processing	38
9.14	Top and Bottom Margins	39
9.15	Proprietary Markings	39
9.16	Private Documents	39
10.	TABLE OF CONTENTS AND COVER SHEET	39
10.1	Table of Contents	40
10.2	Cover Sheet	41
11.	REFERENCES	41
11.1	Automatic Numbering of References	41
11.2	Delimiting Reference Text	41
11.3	Subsequent References	42
11.4	Reference Page	42
12.	MISCELLANEOUS FEATURES	42
12.1	Bold, Italic, and Roman Fonts	42
12.2	Justification of Right Margin	43
12.3	SCCS Release Identification	43
12.4	Two-Column Output	44
12.5	Column Headings for Two-Column Output	44
12.6	Vertical Spacing	45
12.7	Skipping Pages	45
12.8	Forcing an Odd Page	45
12.9	Setting the Point Size and Vertical Spacing	46
12.10	Producing Accents	47
12.11	Inserting Text Interactively	47

13. ERRORS AND DEBUGGING	47
13.1 Error Terminations	47
13.2 Disappearance of Output	48
14. EXTENDING AND MODIFYING THE MACROS ●	48
14.1 Naming Conventions	48
14.2 Sample Extensions	49
15. CONCLUSION	50
REFERENCES	51
Appendix A: USER-DEFINED LIST STRUCTURES ●	52
Appendix B: SAMPLE FOOTNOTES	54
Appendix C: SAMPLE LETTER	56
Appendix D: ERROR MESSAGES	59
Appendix E: SUMMARY OF MACROS, STRINGS, AND NUMBER REGISTERS	62

LIST OF FIGURES

Figure 1. This is an illustration 33



Typing Documents with MM

D. W. Smith and E. M. Piskorik

Bell Laboratories

Piscataway, New Jersey 08854

This guide shows several examples of documents prepared with MM, a set of general-purpose formatting macros used with the UNIX† text formatters *nroff* and *troff* (as well as with the *eqn/neqn* and *tbl* programs) to produce memoranda, letters, books, manuals, etc. References to manuals for these programs are given on p. 16.

In the examples, input is shown in this

Helvetica Medium type.

The resulting output is shown (boxed) in this

Times Roman type.

Substitutable arguments are shown in this

Times Italic type.

Square brackets ([...]) indicate that the enclosed substitutable argument is optional.

All output shown in the examples was done with *troff*; *nroff* output would look somewhat different.*

Contents

Paragraphs and Headings	2
Paragraph and Heading Parameters	2
Lists and List Types	4
Nested Lists	5
Italic, Bold, and Underlining	5
Displays	6
Footnotes	6
Simple Letter—Example	7
Technical Memorandum—Example	9
Memorandum-Style Macros	11
Two-Column Output	13
Equations	14
Tables	15
How to Get Output	16
References	16

† UNIX is a trademark of Bell Laboratories.

* For example, what is called here a “blank line” is a blank line in *nroff*, but is ½ of a vertical space in *troff*, while text that is underlined in *nroff* is *italic* in *troff*.

Paragraphs and Headings

Output for the following is shown on p. 3.

.H 1 "PARAGRAPHS AND HEADINGS"

.P

This section describes the types of paragraphs and the kinds of headings that are available.

.H 2 Paragraphs

.P

Paragraphs are specified by the **.P** macro.

Usually, they are flush left.

The number register **Pt** is used

to change the paragraph style.

.H 2 Headings

.H 3 "Numbered Headings."

There are seven levels of numbered headings.

Level 1 is the most major or highest;

level 7, the lowest.

.P

Headings are specified with the **.H** macro,

whose first argument is the level of heading

(1 through 7).

.P

On output, level-1 headings are preceded by two blank lines; all others, by one blank line.

Level-1 and level-2 headings

are normally bold and stand-alone;

levels 3 through 7 are normally run-in

and underlined (italic).

.H 3 "Unnumbered Headings."

The macro **.HU** is a special case

of **.H**, in that no heading number is printed.

Each **.HU** heading has the level given by

the register **Hu**, whose initial value is 2.

Usually, the value of that

register is set to make unnumbered headings

occur at the lowest heading level in a document.

Paragraph and Heading Parameters

Below are some of the many parameters that can change the appearance of headings and paragraphs, together with their *default* values and their meanings (level 1 is the *most major* or *highest*, while level 7 is the *lowest*):

.nr Pi 5 paragraph indent in characters (or *ens*).

.nr Pt 0 never indent paragraphs (*default*).

.nr Pt 1 always indent paragraphs.

.nr Pt 2 indent paragraphs *except* after headings, lists, and displays.

.ds HF 3 3 2 2 2 2 2

font specification for each of the 7 heading levels:

1 indicates roman,

2 indicates italic,

3 indicates bold.

1. PARAGRAPHS AND HEADINGS

This section describes the types of paragraphs and the kinds of headings that are available.

1.1 Paragraphs

Paragraphs are specified by the `.P` macro. Usually, they are flush left. The number register `Pt` is used to change the paragraph style.

1.2 Headings

1.2.1 Numbered Headings. There are seven levels of numbered headings. Level 1 is the most major or highest; level 7, the lowest.

Headings are specified with the `.H` macro, whose first argument is the level of heading (1 through 7).

On output, level-1 headings are preceded by two blank lines; all others, by one blank line. Level-1 and level-2 headings are normally bold and stand-alone; levels 3 through 7 are normally run-in and underlined (italic).

1.2.2 Unnumbered Headings. The macro `.HU` is a special case of `.H`, in that no heading number is printed. Each `.HU` heading has the level given by the register `Hu`, whose initial value is 2. Usually, the value of that register is set to make unnumbered headings occur at the lowest heading level in a document.

.HM 1 1 1 1 1 1 1

“marking” style for each heading level; the above yields an all-numeric marking style. Available styles are: **1, 0001, A, a, l, and i.**

.nr Hb 2 lowest heading level that is stand-alone (i.e., *not* run-in with the following text).

.nr Hc 0 lowest heading level that is centered.

.nr Hs 2 lowest heading level after which there is a blank line.

.nr Ht 0 heading “marks” will be concatenated.

.nr Hu 2 unnumbered headings (`.HU`) are equivalent to numbered headings at this level for spacing, font, and counting.

.nr Cl 2 lowest heading level to be saved for the table of contents.

.nr Ej 0 lowest heading level that forces the start of a new page.

Default Heading Style	
<i>To get:</i>	<i>Type:</i>
n. HEADING	.H 1 "HEADING" .P
Text ...	Text ...
n.n Heading	.H 2 "Heading" .P
Text ...	Text ...
n.n.n Heading. TextH 3 "Heading." Text ...

Lists and List Types

All lists have a *list-begin* macro, one or more *list items*—each consisting of a `.LI` macro followed by the *list item text*—and the *list-end* macro `.LE`:

list-begin macro

```
.LI
text of first list item ...
.LI
text of second list item ...
:
:
.LE
```

where the *list-begin macro* is one of the following:

```
.AL [type] [indent]    automatic list (type is:
                        1, A, a, I, or i; default is 1).
.BL [indent]           bullet list.
.DL [indent]           dash list.
.ML mark [indent]     marked list (mark is the
                        desired mark).
.RL [indent]           reference list.
.VL indent            variable list.
```

indent is the number of characters of indentation (from the current indent) at which the list is to start; if it is optional and omitted, the default indentation for the given list style is used; *mark* will appear to the left of the indentation.

Output for the following is shown on p. 5.

```
.AL
.LI
Pencilpusher, I., and Hardwired, X.
A New Kind of Set Screw.
.I "Proc. IEEE"
.B 75
(1976), 235-41.
.LI
Nalls, H., and Irons, R.
Fasteners for Printed Circuit Boards.
.I "Proc. ASME"
.B 123
(1974), 23-24.
.LE
```

1. Pencilpusher, I., and Hardwired, X. A New Kind of Set Screw. *Proc. IEEE* 75 (1976), 235-41.
2. Nails, II., and Irons, R. Fasteners for Printed Circuit Boards. *Proc. ASME* 123 (1974), 23-24.

Nested Lists

This is ordinary text to show the margins of the page.

.AL 1

.LI

First-level item.

.AL a

.LI

Second-level item.

.LI

Another second-level item, but a longer one.

.LE

.LI

Return to previous list

(and to previous value of indentation) at this point.

.LI

Third (and last) first-level item.

.LE

.P

Now we're out of the lists and at the left margin that existed at the beginning of this example.

This is ordinary text to show the margins of the page.

1. First-level item.
 - a. Second-level item.
 - b. Another second-level item, but a longer one.
2. Return to previous list (and to previous value of indentation) at this point.
3. Third (and last) first-level-item.

Now we're out of the lists and at the left margin that existed at the beginning of this example.

Italic, Bold, and Underlining

The macros **.I**, **.B**, and **.R** are used to change to, respectively, the italic, bold, and roman fonts (see examples on pp. 4 and 7). A single argument given to either **.I** or **.B** results in that argument being printed in the indicated font. If two or more arguments are given (maximum of 6), the first is printed in the indicated font, the second in the prevailing font, etc., without any space between them. The macro **.IB** prints its successive arguments alternately in italic and bold; there are also **.BI**, **.IR**, **.RI**, **.RB**, and **.BR** macros.

Displays

Displays are blocks of text that are to be kept together—not split across pages. A static display (**.DS**) appears in the same relative position in the output text as it does in the input text; this may result in extra white space at the bottom of a page if a static display is too big to fit there. A floating display (**.DF**), on the other hand, will “float” through the input text to the top of the next page if there is not enough room for it on the current page; thus, the text that *follows* a floating display in the input may *precede* it in the output. Displays can be positioned at the left margin, indented, or centered.

.DS [*format*] [*fill*] [*ind*] **.DF** [*format*] [*fill*] [*ind*]
text ... *text* ...
.DE **.DE**

where *ind* is the amount of indentation from the *right*, and where *format* and *fill* have the following meanings:

<i>format:</i>	<i>Means:</i>	<i>fill:</i>	<i>Means:</i>
L	flush left*	N	no fill*
I	indented	F	fill
C	centered		
CB	centered block		* default

**Highland Avenue, Mountain Station,
 South Orange, Maplewood, Millburn, Short Hills;**

.DS I

and now

for something

completely different

.DE

Summit, Chatham, Madison,

Convent Station, Morristown, New Providence,

Murray Hill, Berkeley Heights.

Highland Avenue, Mountain Station, South
 Orange, Maplewood, Millburn, Short Hills;

and now

for something

completely different

Summit, Chatham, Madison, Convent Station,
 Morristown, New Providence, Murray Hill,
 Berkeley Heights.

Footnotes

Two styles of footnote marking are shown on p. 7. In the first, the asterisk is the mark placed on the footnote and the following **.FS** macro, while in the second, a number is *automatically* generated to mark the footnote. The macros **.FS** and **.FE** are used to delimit the footnote text that is to appear at the bottom of the page.

Among the most important occupants
of the workbench are the long-nosed pliers.

Without this basic tool,*

.FS *

As first shown by Tiger & Leopard (1975).

.FE

few assemblies could be completed.

They may lack the popular\F

.FS

According to Panther & Lion (1979).

.FE

appeal of the sledgehammer . . .

Among the most important occupants of the
workbench are the long-nosed pliers. Without
this basic tool,* few assemblies could be
completed. They may lack the popular¹ appeal of
the sledgehammer . . .

* As first shown by Tiger & Leopard (1975).

1. According to Panther & Lion (1979).

Simple Letter—Example

Output for the following is shown on p. 8.

.ND "May 1, 1979"

.TL

MM Class

.AU "J. J. Jones" JJJ PY 9999 5001 1Q-100

.AT "Education Center"

.MT 0

.DS

To All Students:

.DE

.P

There will be a class on the document preparation
facilities of MM on May 15-18.

This class lasts for 4 half-day (morning) sessions,
each consisting of a lecture
and practice exercises on the system.

.P

The meeting rooms for the class are:

.DS I

.ta 15n

(n represents character positions)

Monday→4D-502

(→ indicates an input tab)

Tuesday→4D-502

Wednesday→2B-639

Thursday→2C-641.

.DE

.P

Please read the following before attending class:

.DL

.LI

.I "UNIX for Beginners,"

Sections I and II.

.LI

.I

A Tutorial Introduction to the UNIX Text Editor.

.R

.LE

Input example continued on next page.



Bell Laboratories

subject: **MM Class**

date: **May 1, 1979**

from: **J. J. Jones**

PY 9999

1Q-100 x5001

To All Students:

There will be a class on the document preparation facilities of MM on May 15-18. This class lasts for 4 half-day (morning) sessions, each consisting of a lecture and practice exercises on the system.

The meeting rooms for the class are:

Monday	4D-502
Tuesday	4D-502
Wednesday	2B-639
Thursday	2C-641.

Please read the following before attending class:

- *UNIX for Beginners*, Sections I and II.
- *A Tutorial Introduction to the UNIX Text Editor*.

These can be obtained from the Computing Information Library.

PY-9999-JJJ-ae

J. J. Jones
Education Center

Copy to
G. H. Hurtz
S. P. LeName

Input example continued from previous page.

.P

These can be obtained from the Computing Information Library.

.SG ae

.NS

G. H. Hurtz

S. P. LeName

.NE

Technical Memorandum—Example

Output for the following is shown on pp. 10-12.

.ND "June 29, 1979"

.TL 12345 666666

**On Constructing a Table of All
Even Prime Numbers**

.AU "S. P. LeName" SPL PY 9999 4000 1Z-123

.AU "G. H. Hurtz" GHH PY 9999 4001 1Z-121

.TM 76543210

.AS 1

.P

**This is an abstract for a technical memorandum
that will appear only on the
cover sheet.**

.P

**The TM number appears on the cover sheet
and on the first page.**

Other Keywords appear only on the cover sheet.

.P

**The abstract may consist of one or more
paragraphs; it must fit on the cover sheet.**

.AE

.OK "Prime Numbers" Even

.MT

.H 1 "INTRODUCTORY MATERIAL"

.P

**The body of the memorandum immediately
follows the .MT macro; the body
may contain headings, paragraphs, lists, etc.
A brief example of lists follows:**

.AL A

.LI

**This is the first item in an alphabetical
list in the body of this memorandum.**

.LI

This is the second item in that list.

.AL

.LI

This is the first item in a (numbered) sub-list.

.LI

This is the second item in that sub-list.

.LE

.LE

.P

**In addition to alphabetized and numbered lists,
there exist bullet lists, variable lists, etc.**

.H 2 "First Second-Level Heading"

.P

**This is the first paragraph under the second-level
heading; notice how that heading is numbered and
where the heading and text are printed.**

.H 1 "SECOND FIRST-LEVEL HEADING"

.P

**This is the first paragraph under the
second first-level heading of the memorandum.**

Input example continued on next page.



Bell Laboratories

subject: **On Constructing a
Table of All Even
Prime Numbers
Charge Case 12345
File Case 666666**

date: **June 29, 1979**

from: **S. P. LeName
PY 9999
1Z-123 x4000
G. H. Hurtz
PY 9999
1Z-121 x4001**

TM 76543210

MEMORANDUM FOR FILE**1. INTRODUCTORY MATERIAL**

The body of the memorandum immediately follows the .MI macro; the body may contain headings, paragraphs, lists, etc. A brief example of lists follows:

- A. This is the first item in an alphabetical list in the body of this memorandum.
- B. This is the second item in that list.
 - 1. This is the first item in a (numbered) sub-list.
 - 2. This is the second item in that sub-list.

In addition to alphabetized and numbered lists, there exist bullet lists, variable lists, etc.

1.1 First Second-Level Heading

This is the first paragraph under the second-level heading; notice how that heading is numbered and where the heading and text are printed.

Input example continued from previous page.

.HU REFERENCES**.RL****.LI****Pencilpusher, I., and Hardwired, X.****A New Kind of Set Screw.****.I "Proc. IEEE"****.B 75****(1976), 235-41.****.LI****Nails, H., and Irons, R.****Fasteners for Printed Circuit Boards.****.I "Proc. ASME"****.B 123****(1974), 23-24.****.LE****.SG tad****.NS 3****List of Even Primes****.NS 2****G. B. Brown****C. P. Jones****.NE****.CS**

2. SECOND FIRST-LEVEL HEADING

This is the first paragraph under the second first-level heading of the memorandum.

REFERENCES

- [1] Pencilpusher, I., and Hardwired, X. A New Kind of Set Screw. *Proc. IEEE* 75 (1976), 235-41.
- [2] Nails, H., and Irons, R. Fasteners for Printed Circuit Boards. *Proc. ASME* 123 (1974), 23-24.

S. P. LeName

PY-9999-SPL/G1111-tad G. H. Hurtz

Att.

List of Even Primes

Copy (without att.) to

G. B. Brown

C. P. Jones

Memorandum-Style Macros

Macros for a memorandum-style document must be invoked in the order shown on pp. 9-10. Once the "memorandum type" (**.MT**) macro has been invoked, none of the macros that precede it can be used. The **.MT** macro controls the format of the "subject, date, from" portion of the first page of the memorandum. Different arguments to the **.MT** macro will produce different kinds of memoranda:

<i>Code:</i>	<i>Means:</i>
.MT 0	no memorandum type is printed
.MT	MEMORANDUM FOR FILE
.MT 1	MEMORANDUM FOR FILE
.MT 2	PROGRAMMER'S NOTES
.MT 3	ENGINEER'S NOTES
.MT 4	Released-paper style
.MT 5	External letter



Bell Laboratories

Cover Sheet for TM

The information contained herein . . . not for publication . . .

Title: **On Constructing a Table of All Even Prime Numbers** Date: **June 29, 1979**

TM: **76543210**

Other Keywords: **Prime Numbers
Even**

Author(s)	Location	Ext.	Charging Case: 12345
S. P. LeName	PY 1Z-123	4000	Filing Case: 666666
G. H. Hertz	PY 1Z-121	4001	

ABSTRACT

This is an abstract for a technical memorandum that will appear only on the cover sheet.

The TM number appears on the cover sheet and on the first page. "Other Keywords" appear only on the cover sheet.

The abstract may consist of one or more paragraphs; it must fit on the cover sheet.

Pages Text: 2 Other: 1 Total: 3

No. Figures: 0 No. Tables: 0 No. Refs.: 2

Z-0000-X SEE REVERSE SIDE FOR DISTRIBUTION LIST

The input and the resulting output for a simple letter are shown on pp. 7-8. Note that the **.TM**, **.AS/.AE**, and **.OK** macros are *not* used there, and that the **.MT** macro has a **0** argument. Documents of the type shown on pp. 2-3 (essentially plain text) are produced by omitting, as well, all the other "memorandum-style" macros: **.ND**, **.TL**, **.AU**, **.AT**, and **.MT** at the beginning of the document, and **.SG**, **.NS/.NE**, and **.CS** at the end.

Like the **.MT** macro, the *notations* macro (**.NS**) may also take different arguments to produce a variety of notations following the signature line and/or on the Memorandum for File (MF) cover sheet:

<i>Code:</i>	<i>Means:</i>
.NS	Copy to
.NS 0	Copy to
.NS 1	Copy (with att.) to
.NS 2	Copy (without att.) to
.NS 3	Att.
.NS 4	Atts.
.NS 5	Enc.
.NS 6	Encs.
.NS 7	Under Separate Cover
.NS 8	Letter to
.NS 9	Memorandum to

If the **.CS** macro is included in the input file (see last line of p. 10), a technical memorandum cover sheet is generated (see p. 12). An MF cover sheet may be obtained by specifying **.AS 2**; in an MF, notations may appear after the **.AE**. The **.TC** macro generates a table of contents; **.CS** and **.TC** can occur only at the end of a document.

Two-Column Output

.nr Pt 1

.hy 14

.DS C

The Declaration of Independence

.DE

.2C

.P

When in the Course of human events, it becomes necessary for one people to dissolve the political bands which have connected them with another, and to assume among the powers of the earth, the separate and equal station to which the Laws of Nature and of Nature's God entitle them, a decent respect to the opinions of mankind requires that they should declare the causes which impel them to the separation.

.P

We hold these truths to be self-evident, that all men are created equal, . . .

The Declaration of Independence

When in the Course of human events, it becomes necessary for one people to dissolve the political bands which have connected them with another, and to assume among the powers of the earth, the separate and equal station to which the Laws of Nature and of Nature's

God entitle them, a decent respect to the opinions of mankind requires that they should declare the causes which impel them to the separation.

We hold these truths to be self-evident, that all men are created equal, . . .

Equations

A stand-alone equation is built within a display.

.DS C

.EQ

x sup 2 over a sup 2 = sqrt { pz sup 2 + qz + r }

.EN

.DE

$$\frac{x^2}{a^2} = \sqrt{pz^2 + qz + r}$$

.DS I

.EQ

bold V bar sub nu left [pile { a above b above c } right] + left [matrix { col { A(11) above . above . } col { . above . above . } col { . above . above A(33) } } right] times left [pile { alpha above beta above gamma } right]

.EN

.DE

$$\bar{\mathbf{v}}_v = \begin{bmatrix} a \\ b \\ c \end{bmatrix} + \begin{bmatrix} A(11) & . & . \\ . & . & . \\ . & . & A(33) \end{bmatrix} \times \begin{bmatrix} \alpha \\ \beta \\ \gamma \end{bmatrix}$$

In-line equations may appear in running text if a character has been defined to mark the left and right ends of the equation. Normally, \$ is used as that character and is so defined by typing the following three lines at the beginning of the document:

.EQ

delim \$\$

.EN

The quantities \$a dot\$, \$b dotdot\$, \$xi tilde times y vec\$ are the values that ...

The quantities \dot{a} , \ddot{b} , $\tilde{\xi} \times \vec{y}$ are the values that ...

This facility can be used for preparing text that contains subscripts and superscripts:

The quantity \$ a sub j sup 3 \$ is ...

The quantity a_j^3 is ...

For more examples, see p. 15 and Reference 4.

Tables

Global table options are **center**, **expand**, **box**, **allbox**, **doublebox**, and **tab(x)**.

The meanings of the key-letters describing the alignment of each entry are:

c	center	n	numerical
r	right-adjust	a	alphabetic subcolumn
l	left-adjust	s	spanned

In the input below, → indicates a tab.

.DF

.TS

allbox ;

cB s s

c c c

n n n .

AT&T Common Stock

Year→Price→Dividend

1973→46-55→2.87

4→40-53→3.24

5→45-52→3.40

6→51-59→.95*

.TE

.DE

*** First quarter only.**

AT&T Common Stock		
Year	Price	Dividend
1973	46-55	2.87
4	40-53	3.24
5	45-52	3.40
6	51-59	.95*

* First quarter only.

.EQ

delim \$\$

.EN

.DS L

.TS

box ;

ll ci

ll .

Name→Definition

.sp 0.5v

Sine→\$sin (x) = 1 over 2j

(e sup jx - e sup -jx) \$

.sp 0.5v

Zeta→\$zeta (s) =

sum from k=1 to inf k sup -s (Re s > 1) \$

.TE

.DE

Name	Definition
Sine	$\sin(x) = \frac{1}{2j}(e^{jx} - e^{-jx})$
Zeta	$\zeta(s) = \sum_{k=1}^{\infty} k^{-s} \quad (\text{Re } s > 1)$

For more examples, see Reference 3.

How to Get Output

Documents with text only:

nroff: mm [options] files or
 nroff [options] -cm files
troff: mmt [options] files or
 troff [options] -cm files

Text and tables:

nroff: mm -t [options] files or
 tbl files | nroff -cm [options]
troff: mmt -t [options] files or
 tbl files | troff -cm [options]

Text, tables, and equations:

nroff: mm -t -e [options] files or
 tbl files | neqn | nroff [options] -cm
troff: mmt -t -e [options] files or
 tbl files | eqn | troff [options] -cm

Some of the *options* that may be specified on the above command lines are:

- ok,m-n print only pages *k*, and *m* through *n*.
- rC1 OFFICIAL FILE COPY in footer.
- rC2 DATE FILE COPY in footer.
- rC3 DRAFT in footer, single spaced.
- rC4 DRAFT in footer, double spaced.
- rLn set page length to *n* lines.*
- rN1 page header at *bottom* of first page *only*.
- rN2 no page number on first page.
- rN3 section-page numbering.
- rOn set page offset to *n* characters.*
- rWn set line width to *n* characters.*
- Tx terminal is type *x*.

The *mm* command recognizes the *nroff* options -T450, -T300, -T300s, etc., to indicate terminal type; if such an option is *not* given, *mm* tries to find the \$TERM variable in the environment. If no \$TERM variable is found, *mm* uses 450 as the value of TERM. The -12 option tells *mm* to use 12-pitch, if possible. See Reference 6 for details.

References

1. *MM—Memorandum Macros* by D. W. Smith and J. R. Mashey.
2. *A Tutorial Introduction to the UNIX Text Editor* by B. W. Kernighan.
3. *TBL—A Program to Format Tables* by M. E. Lesk.
4. *Typesetting Mathematics—User's Guide* (Second Edition) by B. W. Kernighan and L. L. Cherry.
5. *NROFF/TROFF User's Manual* by J. F. Ossanna.
6. *UNIX User's Manual—Release 3.0* by T. A. Dolotta, S. B. Olsson, and A. G. Petrucci, eds.

* For *nroff*, *n* must be an *unscaled* number representing lines or character positions. For *troff*, *n* must be *scaled*.

A Macro Package for View Graphs and Slides

T. A. Dolotta
D. W. Smith

Bell Laboratories
Murray Hill, New Jersey 07974

1. INTRODUCTION

This manual describes a package of UNIX† *troff*(1)¹ macros called MV, designed for typesetting view graphs and slides. This manual assumes that the reader has a basic knowledge of the UNIX system, the UNIX editor *ed*(1), and *troff* [3,8].

With these macros, one can easily prepare view graphs in a variety of dimensions (see Table I below), as well as 35mm slides and 2×2 “super-slides.” These transparencies can be made in a variety of styles, in different fonts, with oversize titles, and with highlighted subordination levels. Because the text from which the foils are typeset is stored on UNIX, the contents of a foil can be readily changed to include new data, or can be incorporated into a new presentation; the text of the foils can be passed through *spell*(1), preprocessed by *eqn*(1), *tbl*(1), *cw*(1), etc.

It is not possible to include artwork, graphics, or multicolored text in foils made with this package except by manual cut-and-paste.

2. SIMPLE EXAMPLES

Before explaining the macros in detail, we illustrate the formatting process with two examples.

2.1 Trivial Example

If you are familiar enough with the UNIX editor *ed*(1) to create the following text file, naming it *trivial*:

```
.Sw
Six stages of a project:
.B
wild enthusiasm
.B
disillusionment
.B
total confusion
.B
search for the guilty
.B
punishment of the innocent
.B
promotion of the non-participants
```

and if you then utter the following UNIX command:

```
mvt trivial
```

you will be rewarded with the first view graph in the Appendix. The *.Sw* is a *foil-start macro*; by looking at that view graph, you should be able to figure out what the *.B* macro does.

† UNIX is a trademark of Bell Laboratories.

1. The notation *name*(*N*) indicates entry *name* in Section *N* of the *UNIX User's Manual* [2].

2.2 Less Trivial Example

The foil that results from typesetting the following input is the second view graph in the Appendix.²

```
.Vw 2 "Less Trivial" "June 29, 1980"
.T "What the Walrus Said"
``The time has come,`` the Walrus said,
.BR
``To talk of many things:
.I .5
.B
Of shoes\(\em and ships\(\em and sealing wax\(\em
.B
Of cabbages\(\em and kings\(\em
.B
And why the sea is boiling hot\(\em
.B
And whether pigs have wings.``
```

The `.Vw` is *another* foil-start macro. We will see exactly what it does in the next section. A bit later on, we will also find out what the other macros in this example (`.T`, `.BR`, and `.I`) are all about.

3. THE MACROS THEMSELVES

The time has come to explain all the MV macros in detail.³

3.1 Foil-Start Macros

Each foil *must* start with a foil-start macro. There are nine foil-start macros for generating nine different-sized foils; the names (and the corresponding mounting-frame sizes) of these macros are shown in Table I.

TABLE I
Foil-Start Macros

Macro Name	Size* and Type	BTL Frame Number†
<code>.VS</code>	7×7 view graph <i>or</i> 2×2 super-slide	E-7351 <i>or</i> E-7351-R
<code>.Vw</code>	7×5 view graph	E-7351-B
<code>.Vh</code>	5×7 view graph	E-7351-A
<code>.VW</code>	9×7 view graph	E-8814 <i>or</i> E-9148
<code>.VH</code>	7×9 view graph	E-8814 <i>or</i> E-9148
<code>.Sw</code>	7×5 35mm slide	E-7351-B
<code>.Sh</code>	5×7 35mm slide	E-7351-A
<code>.SW</code>	9×7 35mm slide	E-8814 <i>or</i> E-9148
<code>.SH</code>	7×9 35mm slide	E-8814 <i>or</i> E-9148

* Size of the mounting frame opening (width times height) in inches.

† BTL stock item number.

The naming convention for these nine macros is that the first character of the name (V or S) distinguishes between view graphs and slides, while the second character indicates whether the foil is square (S), small wide (w), small high (h), big wide (W), or big high (H). Slides are

2. The input string `\(\em` is the *troff* name for the "em dash" (long dash).

3. The MV macros are summarized in *mv(7)*.

“skinnier” than the corresponding view graphs: the ratio of the longer dimension to the shorter one is larger for slides than for view graphs. As a result, slide foils can be used for view graphs, but not vice versa; on the other hand, view graphs can accommodate a bit more text.

Note that `.VW` and `.SW` produce foils that are 7×5.4 inches because commonly available typesetter paper is less than 9 inches wide; these foils must be enlarged by a factor of 9/7 before they can be used as 9-inch-wide by 7-inch-high view graphs.

Each foil-start macro causes the previous foil (if any) to be terminated, foil separators to be produced, and certain heading information to be generated. The *default* heading information consists of three lines of *right*-justified data:

- The current date in the form *mo/dy/yr*
- BTL
- $\text{\textcircled{A}}$ FOIL *n*

where *n* is the sequence number in the current “run”; as explained below, this heading information is replaced by the three arguments of the foil-start macro, if those arguments are given.

The actual projection area is marked by “cross-hairs” (plus signs) that fit into the corners of the view graph mount, helping one to position the foil for mounting.

All foils other than the square (`.VS`) foil also have a set of (horizontal or vertical) “crop marks”; these indicate how much of the foil will be seen if it is made into a slide, rather than into a view graph.

The default heading information can be changed by specifying three optional arguments to the foil-start macro (we use the square brackets `[]` to indicate that the argument they enclose is optional):

```
.XX [ n ] [ id ] [ date ]
```

where *XX* stands for one of the nine foil-start macros, *n* is the foil identifier (typically a number), *id* is other identifying information (typically the initials of the person creating the foil), and *date* is usually the date. The resulting heading information consists of three lines of right-justified text: *id*, *date*, and FOIL *n*. If *date* and *id* are omitted on a foil-start macro, then the corresponding values (if any) from the previous foil-start macro are used.

See the Appendix for examples of all this.

3.2 Level Macros

The MV macros provide four levels of indentation, called `.A`, `.B`, `.C`, and `.D`. Each of these *level macros* causes the text that follows it to be placed at the corresponding level of indentation.

3.2.1 The `.A` Level

The leftmost (left margin) level is obtained by:

```
.A [ x ]
```

The `.A` level is automatically invoked by each of the foil-start macros. Each `.A` macro spaces a half-line from the preceding text, unless the *x* argument is specified (*x* can be any character or string of characters); *x* suppresses the spacing.

The `.A` macro can also be invoked through the `.I` macro (see §3.4).

3.2.2 The `.B` Level

```
.B [ mark [ size ] ]
```

The `.B` level items are marked by a bullet (in a slightly reduced point size). The text that follows the `.B` macro is spaced one half-line from the preceding text.

The `.B` level mark may be changed by specifying the desired mark (which may be any character string⁴) as the first argument (*mark*). Without the second argument, the point size of the mark is not reduced. Thus, one can produce a numbered list as follows:

```
.VS
This is a list of things:
.B 1.
This is thing number 1.
.B 2.
This is thing number 2.
.B 3.
This is the third and last thing on this foil.
```

It is possible to change the point size of the mark: the second argument (*size*), if given, specifies the desired point-size change. An unsigned or positive (+) argument is taken as an increment; a negative (-) argument is a decrement; an argument greater than 99 causes the mark to be reduced in size just as if it were the default mark, namely the bullet. After the mark is printed, the previous point size is restored. All these point-size changes are *completely* invisible to the user.

3.2.3 The `.C` Level

```
.C [ mark [ size ] ]
```

The `.C` level is just like the `.B` level *except* that it is indented farther to the right than the `.B` level and the default mark is a long (*em*) dash (—), also in a slightly reduced point size.

3.2.4 The `.D` Level

```
.D [ mark [ size ] ]
```

The `.D` level's default mark is a bullet (smaller than that used for the `.B` level); the `.D` level is indented farther to the right than the `.C` level and it does *not* space from the previous text; it just causes the following text to start on a new line (in other words, it causes a *break*—see §3.10). Otherwise, it behaves just like the `.B` and `.C` levels.

3.2.5 More about Levels

- The `.A` macro never generates a mark of any sort; it is the “left-margin” macro.
- Repeated `.A` calls are ignored, but each successive call of any of the other three level macros generates the corresponding mark.
- The amount of vertical pre-spacing done by each level macro can be changed with the `.DV` macro (see §3.7).
- Example 3 in the Appendix is devoted to the level macros.

3.3 Titles

The title macro `.T` creates a centered title from its argument:

```
.T string
```

The size of the title is four points larger than the prevailing point size. Remember that the argument must be enclosed within double quotes (“...”) if it contains blanks. Any indentation established by the `.I` macro (see §3.4) has no effect on titles; they are always centered within the foil's horizontal dimension.

See Examples 2, 3, and 4 in the Appendix.

4. All character-string arguments that contain blanks must be quoted (“...”).

3.4 Global Indents

The entire text (except titles) of the foil may be shifted right or left by the `.I` macro:

```
.I [ indent ] [ a [ x ] ]
```

The first argument is the amount of indentation that is to be used to establish a new left margin. This argument may be signed positive or negative, indicating right or left movement from the current margin. If unsigned, the argument specifies the new margin, relative to the *initial* default margin. If the argument is not dimensioned, it is assumed to be in inches (see [3,8] for legal *troff* units). If the argument is null or omitted, 0i is assumed, causing the margin to revert to the initial default margin.

If a second argument is specified, the `.I` macro calls the `.A` macro (see §3.2.1) before exiting. The third argument, if present, is passed to the `.A` macro.

See Examples 2, 4, 5, and 7 in the Appendix.

3.5 Point Sizes and Line Lengths

Each foil-start macro begins the foil with an appropriate default point size⁵ and line length. The prevailing point size and the line length may be changed by invoking the `.S` macro:

```
.S [ ps ] [ ll ]
```

If *ps* is null, the *previous* point size is restored. If *ps* is signed negative, the point size is decremented by the specified amount. If *ps* is signed positive, it is used as an increment, and if *ps* is unsigned, it is used as the new point size. If *ps* is greater than 99, the *initial default* point size is restored (see Table II). Vertical spacing is always 1.25 times the current point size.

The second argument, if given, specifies the line length. It may be dimensioned. If it is *not* dimensioned and less than 10, it is taken as inches; if it is *not* dimensioned and greater than or equal to 10, it is taken as *troff units* (1/432nds of an inch); see also §7.3.

See Examples 4, 5, and 6 in the Appendix.

3.6 Default Fonts

The macros assume that the Helvetica (also known as Geneva) Regular font, mounted in position 1, is the default font. Additional fonts can be mounted and the default font can be changed:

```
.DF n font [ n font ... ]
```

The `.DF` macro informs *troff* that *font* is in position *n*. The first-named font is the default font. Up to four pairs of arguments may be specified.

The `.DF` macro must immediately precede a foil-start macro; the initial setting is equivalent to:

```
.DF 1 H 2 I 3 B 4 S
```

See Examples 4 and 5 in the Appendix.

3.7 Default Vertical Space

The vertical space macro allows one to change the vertical pre-spacing done by each of the four level macros (see §3.2):

```
.DV [ a ] [ b ] [ c ] [ d ]
```

5. The default point sizes for each type of foil are given in Table II.

The first argument (*a*) is the spacing for `.A`, *b* is for `.B`, *c* is for `.C`, and *d* is for `.D`; all non-null arguments must be dimensioned; null arguments leave the corresponding spacing unaffected; the initial setting is equivalent to:

```
.DV .5v .5v .5v 0v
```

3.8 Underlining

The underline macro `.U` takes one or two arguments:

```
.U string1 [ string2 ]
```

The first argument is the string of characters to be underlined. The second argument, if present, is not underlined, but concatenated to the first argument.

Example:

```
.U phototypesetter
```

produces:

```
phototypesetter
```

while:

```
.U under line
```

produces:

```
underline
```

See also Example 4 in the Appendix.

3.9 Synonyms

The MV macro package also recognizes the following upper-case synonyms for the corresponding lower-case *troff* requests:

```
.AD .BR .CE .FI .HY .NA .NF .NH .NX .SO .SP .TA .TI
```

See [8] for definitions of the corresponding *troff* requests.

3.10 Breaks

The `.S`, `.DF`, `.DV`, and `.U` macros do *not* cause a break; the `.I` macro causes a break *only* if it is invoked with more than one argument; all the other MV macros *always* cause a break. The *troff* synonyms (see §3.9) `.AD`, `.BR`, `.CE`, `.FI`, `.NA`, `.NF`, `.SP`, and `.TI` also cause a break.

3.11 Text Filling, Adjusting, and Hyphenation

By default, the MV macros fill, but neither adjust nor hyphenate text. This is an aesthetic judgement that seems correct for foils. These defaults can, of course, be changed by using the `.AD`, `.FI`, `.HY`, `.NA`, `.NF`, and `.NH` macros (see §3.9).

4. THE TROFF PREPROCESSORS

It is possible to use the various *troff* preprocessors to typeset foils that require more powerful formatting capabilities.

4.1 TBL for tables

The `tbl(1)` program can be used to set up columns of data within a view graph or slide. The `.TS` and `.TE` macros are *not* defined in the MV macro package, but are merely flags to `tbl(1)`; see [5], as well as Examples 4 and 7 in the Appendix.

4.2 EQN for Mathematical Expressions

The *eqn(1)* program can be used to typeset mathematical expressions and formulas on foils, *provided* one is careful to specify the proper fonts and point sizes; see [4], as well as Examples 5 and 7 in the Appendix. The *.EQ* and *.EN* macros are *not* defined in the MV macro package:

4.3 CW for Constant-Width Program Examples

The constant-width font simulates computer-terminal and line-printer output, and can be sometimes effective in presenting computer-related topics; see *cw(1)*, as well as Examples 5 and 6 in the Appendix.

5. THE FINISHED PRODUCT

5.1 Phototypesetter Output

The typeset output is obtained via the command:

```
mvt [ options ] file_name ...
```

where *file_name* contains the text and the macro invocations for the foils, and *options* can be one or more of the following:

- a preview output on a terminal (other than a Tektronix 4014—see §5.2)
- e invoke *eqn(1)*
- t invoke *tbl(1)*
- Term* direct output to *term*, where *term* can be one of the following:

st	STARE
40 14	Tektronix 4014
vp	Versatec printer

Using a hyphen (-) in place of *file_name* causes the *mvt* command to read the standard input (rather than a file), as in the following example using the *cw(1)* preprocessor (see §4.3):

```
cw [ options ] file_name ... | mvt [ options ] -
```

5.2 Output Approximation on a Terminal

One can obtain an approximation of the typeset output by entering the command:

```
mvt -a file_name ...
```

The resulting output shows the formatted foils except that:

- Point-size changes are not visible.
- Font changes cannot be seen.
- Titles that are too long appear proper.
- All horizontal motions are reduced to one horizontal space to the right.
- All vertical motions are reduced to one vertical space down.

Thus, for example, it appears that the lines of text following a *.B*, *.C*, or *.D* macro do not align properly (even though, in fact, they do).

Although alignment cannot be determined from this approximation, one can observe line breaks and the amount of vertical space used by the text. If the foil is not full, the macro package prints the number of blank lines (in the then current point size) that remain on the foil; if the foil is full, a warning is printed. If the text *did* overflow the foil, text will be printed after the “cross-hairs.”

5.3 Making Actual View Graphs and Slides

The output of the typesetter is so-called “mechanical paper,” which is white, opaque photographic paper with black letters. There are several very simple processes (e.g., Thermofax, Bruning) for making transparent foils from opaque paper. Because some of these processes involve heat, and because mechanical paper is heat sensitive, one should first make copies of the typesetter output on a good-quality office copier, and then use these copies for making the transparencies.

Getting slides made is a much more complicated photographic process that is best left to professionals. It is possible to make both positive (opaque letters on transparent background) and negative (transparent letters on opaque background) slides, as well as colored-background slides, etc.

6. OPINIONS AND SUGGESTIONS

The following suggestions and authors’ opinions have been derived from experience, from the examination of several other macro packages for making foils [6,7,10,11,12], and from some publications that discuss good and bad foil-making practices [1,9]:

☞ *Foils 3, 4, and 7 in the Appendix violate some or most of the “rules” given below.*

- The most useful foil sizes are .VS and .Vw (or .Sw). This is because most projection screens are either square, or “wide” (i.e., are wider than they are tall), and also because the resulting foils are smaller, easier to carry, and require no enlargement before use.
- The *default* point size for each type of foil (see Table II) is the *smallest* point size that will result in a foil that is legible by an audience of more than a dozen people. Reducing the point size below the default should be avoided. If you have more text than fits onto a foil, *don’t* reduce the point size; use two or more foils instead.
- Don’t abuse font changes. Most novice foil-makers tend to use too many typefaces, resulting in foils that look cluttered and distract the viewer. Be *sure* you need to use a different typeface (e.g., for special emphasis); when in doubt, stick to a single typeface. You should almost never use more than two different typefaces on a single foil.
- Even though this package contains a macro for underlining, don’t use it: underlined typeset text almost always looks awful; instead, if necessary, use a different typeface.
- The Helvetica sans-serif font (which is the default font used by this package) is “fatter” and is, *in foils*, considered easier to read than, for instance, the Times Roman serif font used for typesetting normal, “running” text (such as the text in this paragraph). On the other hand, the Times Roman font will allow you to “squeeze” a bit more text onto a foil. If you intend to use italic and/or bold typefaces in your foils, you probably want to mount (via the .DF macro—see §3.6) either the Helvetica Regular, Italic, and Medium:⁶

.DF 1 H 2 HI 3 HM

or the Times Roman regular, italic, and bold:

.DF 1 R 2 I 3 B

Bold typefaces tend to be a bit overwhelming. On the whole, the choice of fonts is primarily a matter of personal aesthetics. The examples in the Appendix use the following fonts:

6. Helvetica Medium is really a bold typeface.

1, 2, and 3: H (default)
 4 and 7: R and I
 5 and 6: R and CW

- If possible, use the `.SP` macro to insert a bit of additional white space (say, `.5v` or `1v`, where `v` means “vertical space”) at the top of each foil (i.e., increase the top margin).
- Some people believe that foils should not contain any lower-case alphabetic characters to maximize legibility; to the best of our knowledge, this is exactly the opposite of the truth: “normal” upper-and-lower-case text is far more legible than upper-case-only text.⁷ Upper-and-lower-case alphabets have evolved and been refined over millenia precisely because they result in more legible text. Furthermore, such text is less “bulky” than upper-case-only text, so one can get more information onto a foil without crowding it.
- Make all the foils for a presentation as consistent as possible; changing fonts, typefaces, point sizes, etc., from foil to foil tends to jar and distract the viewer. While it is possible to introduce emphasis and draw the viewer’s attention to particular items with such changes, this only works if you do it very purposefully and very sparingly; overuse of these techniques is almost always counter-productive.

In summary, the dictum that “the medium is the message” doesn’t apply to foil making; so when in doubt:

- Don’t change point sizes.
- Don’t change fonts or typefaces.
- Don’t underline!
- Use many “sparse” foils, rather than few “dense” ones.
- Use fewer words, rather than more.
- Use larger point sizes, rather than smaller.
- Use larger top and bottom margins, rather than smaller.
- Use normal upper-and-lower-case text, rather than upper-case only.

7. WARNINGS

7.1 Use of Troff Requests

In general, it is not advisable to intermix arbitrary *troff* requests with the MV macros, because this often leads to undesirable (and sometimes downright astonishing) results. The “safe” requests are the ones for which upper-case synonyms have been defined in the MV package (see §3.9). Other *troff* requests should be used sparingly (if at all), and with care and discipline. Particularly dangerous are the requests that affect point size, indentation, page offset, line and title lengths, and vertical spacing between lines. Use the `.S` and `.I` macros instead (see §3.5 and §3.4).

7.2 Reserved Names

Certain names are used internally by this macro package. In particular, all two-character names starting with either `)` or `]` are reserved. Names that are the same as names of the MV macros and strings described in this manual, or the same as any *troff* names [8], cannot be used either. Furthermore, if any of the preprocessors (see §4) are used, their reserved names must also be avoided.

7. The only exception to this rule are foils set in a point size so small that lower-case characters simply can’t be read; this is usually the case for foils produced on a normal typewriter.

7.3 Miscellaneous

The `.S` macro changes the point size and vertical spacing immediately, but a line-length change requested with that macro does not take effect until the next level-macro call.

Specifying a *third* argument to the `.S` macro usually results in a disaster.

The string `Tm` (invoked as `*(Tm)` generates a trademark symbol.

The tilde (`-`) is defined by the `MV` macros as a "non-paddable" space; that is, the tilde may be used wherever a fixed-size (non-adjustable) space is desired. To override this definition, include the following line in your input file:

```
.tr --
```

8. DIMENSIONAL DETAILS

Table II shows, for each style of view graph, the default point size, the maximum number of lines of text (at the default point size), and the height, width, and "aspect ratio," both nominal and actual.

TABLE II
Default Point Size, Dimensions, and Aspect Ratios

Macro Name	Point Size	Max. Lines	Nominal				Actual (Text)			
			Width (inches)	Height (inches)	Aspect Ratio	$\frac{1}{AR}$	Width (inches)	Height (inches)	Aspect Ratio	$\frac{1}{AR}$
<code>.VS</code>	18	21	7	7	1	1	6	6.8	1.13	.88
<code>.Vw</code>	14	19	7	5	.71	1.4	6	4.8	.8	1.25
<code>.Vh</code>	14	27	5	7	1.4	.71	4.2	6.8	1.6	.62
<code>.VW</code>	14	21	7	5.4	.77	1.3	6	5.2	.87	1.15
<code>.VH</code>	18	28	7	9	1.3	.77	6	8.8	1.5	.68
<code>.Sw</code>	14	18	7	4.6	.67	1.5	6	4.4	.73	1.4
<code>.Sh</code>	14	27	4.6	7	1.5	.67	3.8	6.8	1.8	.56
<code>.SW</code>	14	18	7	4.6	.67	1.5	6	4.4	.73	1.4
<code>.SH</code>	18	28	6	9	1.5	.67	5	8.8	1.76	.57

NOTES: "Max. Lines" is the maximum number of lines of text at the *default* point size.

"Aspect Ratio" (AR) is the ratio of height over width.

Remember that, normally, each `.A`, `.B`, and `.C` macro generates a ½-line space.

The `.SW` (if used as a view graph) and `.VW` foils must be enlarged by a factor of 9/7.

ACKNOWLEDGEMENTS

We thank the many users of `MV` who provided the feedback necessary to refine the various features of this package in the early stages of its development and who were willing to use it despite the fact that, during the first several years of its existence, the only available user "documentation" was by word of mouth.

REFERENCES

- [1] Bell Laboratories. *Visual Aid Standards* (1973—out of print).
- [2] Dolotta, T. A., Olsson, S. B., and Petrucelli, A. G. (eds.). *UNIX User's Manual—Release 3.0*, Bell Laboratories (June 1980).
- [3] Kernighan, B. W. *A TROFF Tutorial*, Bell Laboratories.
- [4] Kernighan, B. W., and Cherry, L. L. *Typesetting Mathematics—User's Guide* (Second Edition), Bell Laboratories.
- [5] Lesk, M. *TBL—A Program to Format Tables*, Bell Laboratories.
- [6] McGill, R. *VMAC—Commands for Preparing Vu-graphs or Posters*, Bell Laboratories (1976).
- [7] Noll, J. C. *ATS Bulletin 77-5*, Bell Laboratories (1977).
- [8] Ossanna, J. F. *NROFF/TROFF User's Manual*, Bell Laboratories.
- [9] Perry, R. E. *Audience Requirements for Technical Speakers*, American Federation of Information Processing Societies (1971).
- [10] Renkel, W. H. *VGEL—View Graph Extended Language*, Bell Laboratories (1978).
- [11] Sturman, J. N. *MVIEW—A Set of Macrocommands for the Generation of View Graphs*, Bell Laboratories (1978).
- [12] Vogel, G. C. *Easy Phototypeset View Graphs*, Bell Laboratories (1977).

APPENDIX

This Appendix contains seven examples. The input for Examples 1 and 2 is given in §2 above. The input for each of the other examples precedes the corresponding view graph. Note that the *output* of Example 6 is, essentially, the *input* for Example 7.

EXAMPLE 1:

9/15/81
BTL
⊗ FOIL 1

Six stages of a project:

- wild enthusiasm
- disillusionment
- total confusion
- search for the guilty
- punishment of the innocent
- promotion of the non-participants

EXAMPLE 2:

June 29, 1980
Less Trivial
⊗ FOIL 2

What the Walrus Said

“The time has come,” the Walrus said,
“To talk of many things:

- Of shoes—and ships—and sealing wax—
- Of cabbages—and kings—
- And why the sea is boiling hot—
- And whether pigs have wings.”

EXAMPLE 3:

```

.Vh 3 "Levels & Marks"
.T "Foil Levels & Level Marks"
This is the .A (left margin) level;
.B
this is the .B level,
.B
as is this;
.C
this is the .C level,
.C
as is this;
.D
and this is the .D level,
.D
as is this.
.A
The large bullet, the dash, and the small
bullet are the default ``marks`` for
levels .B, .C, and .D, respectively.
However, these three levels can also
be marked arbitrarily:
.B B.
Like this (this is the .B level);
.C 3.
like this (this is the .C level);
.D d.
like this (this is the .D level), or
.D iv.
like this, or even
.D \(\rh-\(bu +4
like this.
.A
The .A level cannot be marked.
.B
An arbitrary number of lines of text
can be included in any item at any level;
the text will be filled, but neither adjusted
nor hyphenated, just like this .B level item.

```

Foil Levels & Level Marks

This is the .A (left margin) level;

- this is the .B level,
- as is this;
 - this is the .C level,
 - as is this;
 - and this is the .D level,
 - as is this.

The large bullet, the dash, and the small bullet are the default “marks” for levels .B, .C, and .D, respectively. However, these three levels can also be marked arbitrarily:

B. Like this (this is the .B level);

3. like this (this is the .C level);

d. like this (this is the .D level), or

iv. like this, or even

 ● like this.

The .A level cannot be marked.

- An arbitrary number of lines of text can be included in any item at any level; the text will be filled, but neither adjusted nor hyphenated, just like this .B level item.

EXAMPLE 4:

```

.DF 1 R
.VS 4 Complex
.T "Of Bits & Bytes & Words"
.S -4
.I 3 A x
.ft I
But let your communication be, Yea, yea;
Nay, nay: for whatsoever is more than these
cometh of evil.*
.ft
.I +1 a nospace
Matthew 5:37
.BR
.S
.I 0 .A
Binary notation has been around for a
.S +6
long
.S
time.
.B
The above verse tells us to use:
.C 1)
binary notation,
.ft I
and
.ft
.C 2)
redundancy
.D \(rh
(in communicating)
.B
Binary notation is
.U not
suited for human use, above verse to
the contrary notwithstanding.
.SP
.S -2
.TS
box ;
c | c | c | c
l | c | c | c .
System Bits/Byte      Bytes/Word      Bits/Word


IBM-7090/94      6      6      36
IBM-360/370      8      4      32
PDP-11/70        8      2      16
.TE
.S
.S -4
.U -----
.BR
* The use of this verse in this context
is plagiarized from C. Shannon.
.S

```

Of Bits & Bytes & Words

*But let your communication be,
Yea, yea; Nay, nay: for whatsoever
is more than these cometh of evil.*
Matthew 5:37*

Binary notation has been around for a long time.

- The above verse tells us to use:
 - 1) binary notation, *and*
 - 2) redundancy
 (in communicating)
- Binary notation is not suited for human use, above verse to the contrary notwithstanding.

System	Bits/Byte	Bytes/Word	Bits/Word
IBM 7090/94	6	6	36
IBM 360/370	8	4	32
PDP 11/70	8	2	16

* The use of this verse in this context is plagiarized from C. Shannon.

EXAMPLE 5:

```

.de CW
.I .5 a
.NF
..
.de CN
.FI
.I 0 a
..
.DF 1 R 2 I 3 CW
.VS 5 "CW & EQN"
.EQ
gsize 18
.EN
.S 100 5.5
Input:
.CW
.EQ
sum from k=1 to inf m sup k-1
~-- 1 over 1-m
.EN
.CN
Output:
.I 2 a
.EQ
sum from k=1 to inf m sup k-1
-- 1 over 1-m
.EN
.I 0 a
Input:
.CW
The equation $ f(t) --- 2 pi
int sin ( omega t ) dt $
is used here in running text,
rather than being displayed.
.CN
Output:
.I .5 a
.EQ
delim $$
.EN
.AD
The equation $ f(t) --- 2 pi
int sin ( omega t ) dt $
is used here in running text,
rather than being displayed.
.EQ
delim off
gsize 10
.EN

```

Input:

```
.EQ
sum from k=1 to inf m sup k-1
  ~ = ~ 1 over 1-m
.EN
```

Output:

$$\sum_{k=1}^{\infty} m^{k-1} = \frac{1}{1-m}$$

Input:

The equation \$ f(t) \sim 2 \pi \int \sin(\omega t) dt \$
is used here in running text,
rather than being displayed.

Output:

The equation $f(t) = 2\pi \int \sin(\omega t) dt$ is
used here in running text, rather than
being displayed.

EXAMPLE 6:

```

.VS 6 "The Works: Input"
Input:
.S -4
.CW
.TS
center doublebox ;
Cip+4 | Cip+4 S S
^ | L L L
^ | C | C | C
^ | C | C | C
Li | C | C | N .
Users\(->Hardware
\(-> \(-> \(->
\(->UNIX\*(Tm\(->Model\(->Serial
\(->System\(->\^(->Number
=
OS Dev.\(->A\(->VAX\(->54
SGS Dev.\(->B\(->11/70\(->3275
Low-End\(->C\(->11/23\(->221

And now ...\(->T{
.NA
Some filled text and an equation:
T)\(->T{
$ zeta (s) = prod
from k=1 to inf k sup -s $
.AD
T)\(->1.2
.TE
.CN

```

(\(-> = tab)

+

+

Input:

```

.TS                                (→ = tab)
center doublebox ;
Cip+4 | Cip+4 S S
^ | L L L
^ | C | C | C
^ | C | C | C
Li | C | C | N .
Users→Hardware
→_→_→_
→UNIX\*(Tm→Model→Serial
→System→\^→Number
=
OS Dev.→A→VAX→54
SGS Dev.→B→11/70→3275
Low-End→C→11/23→221

-
And now ...→T{
.NA
Some filled text and an equation:
T}→T{
$ zeta (s) = prod
from k=1 to inf k sup -s $
.AD
T}→1.2
.TE

```

+

+

EXAMPLE 7:

```

.VS 7 "The Works: Output"
.EQ
delim $$
gsize 14
.EN
Output:
.I 0 a
.SP
.TS
center doublebox ;
Cip+4 | Cip+4 S S
^ | L L L
^ | C | C | C
^ | C | C | C
Li | C | C | N .
Users Hardware

      UNIX\*(Tm      Model  Serial
      System \ ^     Number

=
OS Dev.      A      VAX      54
SGS Dev.     B      11/70   3275
Low-End      C      11/23   221

And now ...  T{
.NA
Some filled text and an equation:
T}          T{
$ zeta (s) = prod
from k=1 to inf k sup -s $
.AD
T}          1.2
.TE
.EQ
delim off
gsize 10
.EN

```

+

+

Output:

<i>Users</i>	<i>Hardware</i>		
	UNIX TM System	Model	Serial Number
<i>OS Dev.</i>	A	VAX	54
<i>SGS Dev.</i>	B	11/70	3275
<i>Low-End</i>	C	11/23	221
<i>And now ...</i>	Some filled text and an equation:	$\zeta(s) = \prod_{k=1}^{\infty} k^{-s}$	1.2

+

+

TBL—A Program to Format Tables*M. E. Lesk*Bell Laboratories
Murray Hill, New Jersey 07974**ABSTRACT**

Tbl is a document formatting preprocessor for *troff* or *nroff* which makes even fairly complex tables easy to specify and enter. It is available on the PDP-11 UNIX† system, and on Honeywell 6000 GCOS. Tables are made up of columns which may be independently centered, right-adjusted, left-adjusted, or aligned by decimal points. Headings may be placed over single columns or groups of columns. A table entry may contain equations, or may consist of several rows of text. Horizontal or vertical lines may be drawn as desired in the table, and any table or element may be enclosed in a box. For example:

1970 Federal Budget Transfers (in billions of dollars)			
State	Taxes collected	Money spent	Net
New York	22.91	21.35	-1.56
New Jersey	8.33	6.96	-1.37
Connecticut	4.12	3.10	-1.02
Maine	0.74	0.67	-0.07
California	22.29	22.42	+0.13
New Mexico	0.70	1.49	+0.79
Georgia	3.30	4.28	+0.98
Mississippi	1.15	2.32	+1.17
Texas	9.33	11.13	+1.80

INTRODUCTION

Tbl turns a simple description of a table into a *troff* or *nroff* [1] program (list of requests) that prints the table. *Tbl* may be used on the PDP-11 UNIX [2] system and on the Honeywell 6000 GCOS system. It attempts to isolate a portion of a job that it can successfully handle and leave the remainder for other programs. Thus *tbl* may be used with the equation formatting program *eqn* [3] and/or various *nroff/troff* layout macro packages [4,5,6], but does not duplicate their functions.

This memorandum is divided into two parts. First we give the rules for preparing *tbl* input; then some examples are shown. The description of rules is precise but technical, and the beginning user may prefer to read the examples first, as they show some common table arrangements. A section explaining how to invoke *tbl* precedes the examples. To avoid repetition, henceforth read "*troff*" as "*troff* or *nroff*."

† UNIX is a trademark of Bell Laboratories.

The input to *tbl* is text for a document, with tables preceded by a “.TS” (table start) command and followed by a “.TE” (table end) command. *Tbl* processes the tables, generating *troff* formatting requests, and leaves the remainder of the text unchanged. The “.TS” and “.TE” lines are copied, too, so that *troff* layout macros (such as the memorandum formatting macros [4,6]) can use these lines to delimit and place tables as they see fit. In particular, any arguments on the “.TS” or “.TE” lines are copied but otherwise ignored, and may be used by document layout macro requests. The format of the input is as follows:

```
text
.TS
table
.TE
text
.TS
table
.TE
text
...
```

where the format of each table is as follows:

```
.TS
options ;
format .
data
.TE
```

Each table is independent, and must contain formatting information followed by the data to be entered in the table. The formatting information, which describes the individual columns and rows of the table, may be preceded by a few options that affect the entire table. A detailed description of tables is given in the next section.

INPUT COMMANDS

As indicated above, a table contains, first, global options, then a format section describing the layout of the table entries, and then the data to be printed. The format and data are always required, but not the options. The various parts of the table are entered as follows:

- 1) **OPTIONS.** There may be a single line of options affecting the whole table. If present, this line must follow the .TS line immediately and must contain a list of option names separated by spaces, tabs, or commas, and must be terminated by a semicolon. The allowable options are:

```
center    — center the table (default is left-adjust);
expand    — make the table as wide as the current line length;
box       — enclose the table in a box;
allbox    — enclose each item in the table in a box;
doublebox — enclose the table in two boxes;
tab (x)   — use x instead of tab to separate data items.
linesize (n) — set lines or rules (e.g., from box) in n-point type;
delim (xy) — recognize x and y as the eqn delimiters.
```

The *tbl* program tries to keep boxed tables on one page by issuing appropriate “need” (.ne) requests. These requests are calculated from the number of lines in the tables, and if there are spacing requests embedded in the input, the .ne requests may be inaccurate; use normal *troff* procedures, such as keep-release macros, in that case. The user who

must have a multi-page boxed table should use macros designed for this purpose, as explained below under 'Usage.'

- 2) **FORMAT.** The format section of the table specifies the layout of the columns. Each line in this section corresponds to one line of the table (except that the last line corresponds to all following lines up to the next .T&, if any—see below), and each line contains a key-letter for each column of the table. It is good practice to separate the key letters for each column by spaces or tabs. Each key-letter is one of the following:

- L** or **l** to indicate a left-adjusted column entry;
- R** or **r** to indicate a right-adjusted column entry;
- C** or **c** to indicate a centered column entry;
- N** or **n** to indicate a numerical column entry, to be aligned with other numerical entries so that the units digits of numbers line up;
- A** or **a** to indicate an alphabetic subcolumn; all corresponding entries are aligned on the left, and positioned so that the widest is centered within the column (see example on page 13);
- S** or **s** to indicate a spanned heading, i.e., to indicate that the entry from the previous column continues across this column (not allowed for the first column of the table, obviously); or
- ^** to indicate a vertically spanned heading, i.e., to indicate that the entry from the previous row continues down through this row (not allowed for the first row of the table, obviously).

When numerical alignment is specified, a location for the decimal point is sought. The rightmost dot (.) adjacent to a digit is used as a decimal point; if there is no dot adjoining a digit, the rightmost digit is used as a units digit; if no alignment is indicated, the item is centered in the column. However, the special non-printing character string \& may be used to override unconditionally dots and digits, or to align alphabetic data; this string lines up where a dot normally would, and then disappears from the final output. In the example below, the items shown at the left will be aligned (in a numerical column) as shown on the right:

13	13
4.2	4.2
26.4.12	26.4.12
abc	abc
abc\&	abc
43\&3.22	433.22
749.12	749.12

Note: If numerical data are used in the same column with wider **L** or **r** type table entries, the widest *number* is centered relative to the wider **L** or **r** items (**L** is used instead of **l** for readability; they have the same meaning as key-letters). Alignment within the numerical items is preserved. This is similar to the behavior of **a** type data, as explained above. However, alphabetic subcolumns (requested by the **a** key-letter) are always slightly indented relative to **L** items; if necessary, the column width is increased to force this. This is not true for **n** type entries.

Warning: The **n** and **a** items should not be used in the same column.

For readability, the key-letters describing each column should be separated by spaces. The end of the format section is indicated by a period. The layout of the key-letters in the format section resembles the layout of the actual data in the table. Thus a simple format might appear as:

c s s
l n n .

which specifies a table of three columns. The first line of the table contains a heading centered across all three columns; each remaining line contains a left-adjusted item in the first column followed by two columns of numerical data. A sample table in this format might be:

	Overall title	
Item-a	34.22	9.1
Item-b	12.65	.02
Items: c,d,e	23	5.8
Total	69.87	14.92

There are some additional features of the key-letter system:

Horizontal lines — A key-letter may be replaced by '_' (underscore) to indicate a horizontal line in place of the corresponding column entry, or by '=' to indicate a double horizontal line. If an adjacent column contains a horizontal line, or if there are vertical lines adjoining this column, this horizontal line is extended to meet the nearby lines. If any data entry is provided for this column, it is ignored and a warning message is printed.

Vertical lines — A vertical bar may be placed between column key-letters. This will cause a vertical line between the corresponding columns of the table. A vertical bar to the left of the first key-letter or to the right of the last one produces a line at the edge of the table. If two vertical bars appear between key-letters, a double vertical line is drawn.

Space between columns — A number may follow the key-letter. This indicates the amount of separation between this column and the next column. The number normally specifies the separation in *ens* (one en is about the width of the letter 'n').* If the *expand* option is used, then these numbers are multiplied by a constant such that the table is as wide as the current line length. The default column separation number is 3. If the separation is changed the worst case (largest space requested) governs.

Vertical spanning — Normally, vertically spanned items extending over several rows of the table are centered in their vertical range. If a key-letter is followed by t or T, any corresponding vertically spanned item will begin at the top line of its range.

Font changes — A key-letter may be followed by a string containing a font name or number preceded by the letter f or F. This indicates that the corresponding column should be in a different font from the default font (usually Roman). All font names are one or two letters; a one-letter font name should be separated from whatever follows by a space or tab. The single letters B, b, I, and i are shorter synonyms for fB and fI. Font-change requests given with the table entries override these specifications.

Point size changes — A key-letter may be followed by the letter p or P and a number to indicate the point size of the corresponding table entries. The number may be a signed digit, in which case it is taken as an increment or decrement from the current point size. If both a point size and a column separation value are given, one or more blanks must separate them.

Vertical spacing changes — A key-letter may be followed by the letter v or V and a number to indicate the vertical line spacing to be used within a multi-line corresponding table entry. The number may be a signed digit, in which case it is taken as an increment or decrement from the current vertical spacing. A column

* More precisely, an en is a number of points (1 point = 1/72 inch) equal to half the current type size.

separation value must be separated by blanks or some other specification from a vertical spacing request. This request has no effect unless the corresponding table entry is a text block (see below).

Column width indication — A key-letter may be followed by the letter *w* or *W* and a width value in parentheses. This width is used as a minimum column width. If the largest element in the column is not as wide as the width value given after the *w*, the largest element is assumed to be that wide. If the largest element in the column is wider than the specified value, its width is used. The width is also used as a default line length for included text blocks. Normal *troff* units can be used to scale the width value; if none are used, the default is *ens*. If the width specification is a unitless integer the parentheses may be omitted. If the width value is changed in a column, the *last* one given controls.

Equal-width columns — A key-letter may be followed by the letter *e* or *E* to indicate equal-width columns. All columns whose key-letters are followed by *e* or *E* are made the same width. This permits the user to get a group of regularly spaced columns.

Staggered columns — A key-letter may be followed by the letter *u* or *U* to indicate that the corresponding entry is to be moved up one-half line. This makes it easy, for example, to have a column of differences between numbers in an adjoining column. The *allbox* option does not work with staggered columns.

Zero-width item — A key-letter may be followed by the letter *z* or *Z* to indicate that the corresponding data item is to be ignored in calculating column widths. This may be useful, for example, in allowing headings to run across adjacent columns where spanned headings would be inappropriate.

Note: The order of the above features is immaterial; they need not be separated by spaces, except as indicated above to avoid ambiguities involving point size and font changes. Thus a numerical column entry in italic font and 12-point type with a minimum width of 2.5 inches and separated by 6 *ens* from the next column could be specified as

```
np12w(2.5i)fI 6
```

Alternative notation — Instead of listing the format of successive lines of a table on consecutive lines of the format section, successive line formats may be given on the same line, separated by commas, so that the format for the example above might have been written:

```
c s s , l n n .
```

Default — Column descriptors missing from the end of a format line are assumed to be *L*. The longest line in the format section, however, defines the number of columns in the table; extra columns in the data are ignored silently.

- 3) **DATA.** The data for the table are typed after the format. Normally, each table line is typed as one line of data. Very long input lines can be broken: any line whose last character is ** is combined with the following line (and the ** vanishes). The data for different columns (the table entries) are separated by tabs, or by whatever character has been specified in the option *tabs* option. There are a few special cases:

Troff requests within tables — An input line beginning with a *'* followed by anything but a number is assumed to be a request to *troff* and is passed through unchanged, retaining its position in the table. So, for example, space within a table may be produced by *“.sp”* requests in the data.

Full width horizontal lines — An input line containing only the character *_* (underscore) or *=* (equal sign) is taken to be a single or double line, respectively, extending the full width of the table.

Single column horizontal lines — An input table *entry* containing only the character `_` or `=` is taken to be a single or double line extending the full width of the *column*. Such lines are extended to meet horizontal or vertical lines adjoining this column. To obtain these characters explicitly in a column, either precede them by `\&` or follow them by a space before the usual tab or new-line.

Short horizontal lines — An input table *entry* containing only the string `_` is taken to be a single line as wide as the contents of the column. It is not extended to meet adjoining lines.

Repeated characters — An input table *entry* containing only a string of the form `\Rx` where *x* is any character is replaced by repetitions of the character *x* as wide as the data in the column. The sequence of *x*'s is not extended to meet adjoining columns.

Vertically spanned items — An input table entry containing only the character string `\^` indicates that the table entry immediately above spans downward over this row. It is equivalent to a table format key-letter of `^`.

Text blocks — In order to include a block of text as a table entry, precede it by `T{` and follow it by `T}`. Thus the sequence

```
... T{
  block of
  text
T} ...
```

is the way to enter, as a single entry in the table, something that cannot conveniently be typed as a simple string between tabs. Note that the `T}` end delimiter must begin a line; additional columns of data may follow after a tab on the same line. See the example on page 11 for an illustration of included text blocks in a table. If more than thirty or so text blocks are used in a table, various limits in the *troff* program are likely to be exceeded, producing diagnostics such as 'too many string/macro names' or 'too many number registers.'

Text blocks are pulled out from the table, processed separately by *troff*, and replaced in the table as a solid block. If no line length is specified in the *block of text* itself, or in the table format, the default is to use $L \times C / (N + 1)$ where *L* is the current line length, *C* is the number of table columns spanned by the text, and *N* is the total number of columns in the table. The other parameters (point size, font, etc.) used in setting the *block of text* are those in effect at the beginning of the table (including the effect of the `“.TS”` macro) and any table format specifications of size, spacing, and font, using the `p`, `v` and `f` modifiers to the column key-letters. Requests within the text block itself are also recognized, of course. However, *troff* requests within the table data but not within the text block do not affect that block.

Warnings: Although any number of lines may be present in a table, only the first 200 lines are used in setting up the table; a multi-page table, of course, may be arranged as several single-page tables if this proves to be a problem. Other difficulties with formatting may arise because, in the calculation of column widths all table entries are assumed to be in the font and size being used when the `“.TS”` command was encountered, except for font and size changes indicated (a) in the table format section and (b) within the table data (as in the entry `\s+3\fiData\fp\s0`). Therefore, although arbitrary *troff* requests may be sprinkled in a table, care must be taken to avoid confusing the width calculations; use requests such as `‘.ps’` with care.

- 4) **ADDITIONAL COMMAND LINES.** If the format of a table must be changed after many similar lines, as with sub-headings or summarizations, the `“.T&”` (table continue) command can be used to change column parameters. The outline of such a table input is:

```

.TS
options ;
format .
data
...
.T&
format .
data
.T&
format .
data
.TE

```

as in the examples on pages 10 and 13. Using this procedure, each table line can be close to its corresponding format line.

Warning: It is not possible to change the number of columns, the space between columns, the global options such as *box*, or the selection of columns to be made equal-width. Furthermore, “.T&” is not recognized after the first 200 lines of a table.

USAGE

On UNIX, *tbl* can be run on a simple table with the command

```
tbl input-file | troff
```

but for more complicated use, where there are several input files, and they contain equations and *ms* (or *mm*) macro requests as well as tables, the normal command would be

```
tbl file-1 file-2 ... | eqn | troff -ms      (or -mm)
```

and, of course, the usual options may be used on the *troff* and *eqn* commands. The usage for *nroff* is similar to that for *troff*, but only *TELETYPE*® Model 37 and Diablo-mechanism (DASI or GSI) terminals can print boxed tables directly. If a file name is “-”, the standard input is read at that point.

For the convenience of users employing line printers without adequate driving tables or post-filters, there is a special *-TX* command-line option to *tbl* which produces output that does not have fractional line motions in it. The only other command-line options recognized by *tbl* are *-ms* and *-mm* which are turned into commands to fetch the corresponding macro files; usually it is more convenient to place these arguments on the *troff* part of the command line, but they are accepted by *tbl* as well.

Note that when *eqn* and *tbl* are used together on the same file *tbl* should be used first. If there are no equations within tables, either order works, but it is usually faster to run *tbl* first, since *eqn* normally produces a larger expansion of the input than *tbl*. However, if there are equations within tables (using the *delim* mechanism in *eqn*), *tbl* must be first or the output will be scrambled. Users must also beware of using equations in *n*-style columns; this is nearly always wrong, since *tbl* attempts to split numerical format items into two parts and this is not possible with equations. The user can defend against this by giving the *delim(xx)* table option; this prevents splitting of numerical columns within the delimiters. For example, if the *eqn* delimiters are *\$\$*, giving *delim(\$\$)* causes a numerical column such as **1245 \$+ - 16\$** to be divided after 1245, not after 16.

Tbl accepts up to about 35 columns, but the actual number that can be processed may be smaller, depending on availability of *troff* number registers. The user must avoid number register names used by *tbl*, which include two-digit numbers from 31 to 99 and strings of the form *4x*, *5x*, *#x*, *x+*, *x|*, *^x*, and *x-*, where *x* is any lower-case letter. The names *##*, *#-*, and *#^* are also used in certain circumstances. To conserve register names, the *n* and *a* formats share a register; hence the restriction above that they may not be used in the same column.

For aid in writing layout macros, *tbl* defines a number register TW which is the table width; it is defined by the time that the “.TE” macro is invoked and may be used in the expansion of that macro. More importantly, to assist in laying out multi-page boxed tables the macro T# is defined to produce the bottom lines and side lines of a boxed table, and then invoked at its end. By use of this macro in the page footer a multi-page table can be boxed. In particular, the *ms* and *mm* macros can be used to print a multi-page boxed table with a repeated heading by giving the argument H to the “.TS” macro. If the table start macro is written

```
.TS H
```

a line of the form

```
.TH
```

must be given in the table after any table heading (or at the start if none). Material up to the “.TH” is placed at the top of each page of table; the remaining lines in the table are placed on several pages as required. Note that this is *not* a feature of *tbl*, but of the *ms* and *mm* macros.

EXAMPLES

Here are some examples illustrating features of *tbl*. The symbol ⊕ in the input represents a tab character.

Input:

```
.TS
box;
c c c
l l l.
Language ⊕Authors ⊕Runs on

Fortran ⊕Many ⊕Almost anything
PL/1 ⊕IBM ⊕360/370
C ⊕BTL ⊕11/45,H6000,370
BLISS ⊕Carnegie-Mellon ⊕PDP-10,11
IDS ⊕Honeywell ⊕H6000
Pascal ⊕Stanford ⊕370
.TE
```

Output:

Language	Authors	Runs on
Fortran	Many	Almost anything
PL/1	IBM	360/370
C	BTL	11/45,H6000,370
BLISS	Carnegie-Mellon	PDP-10,11
IDS	Honeywell	H6000
Pascal	Stanford	370

Input:

```
.TS
allbox;
c s s
c c c
n n n.
AT&T Common Stock
Year ⊕Price ⊕Dividend
1971 ⊕41-54 ⊕$2.60
2 ⊕41-54 ⊕2.70
3 ⊕46-55 ⊕2.87
4 ⊕40-53 ⊕3.24
5 ⊕45-52 ⊕3.40
6 ⊕51-59 ⊕.95*
.TE
* (first quarter only)
```

Output:

AT&T Common Stock		
Year	Price	Dividend
1971	41-54	\$2.60
2	41-54	2.70
3	46-55	2.87
4	40-53	3.24
5	45-52	3.40
6	51-59	.95*

* (first quarter only)

Input:

```
.TS
box;
c s s
c|c|c
1|1|n.
Major New York Bridges
=
Bridge ⊕ Designer ⊕ Length
-
Brooklyn ⊕ J. A. Roebling ⊕ 1595
Manhattan ⊕ G. Lindenthal ⊕ 1470
Williamsburg ⊕ L. L. Buck ⊕ 1600
-
Queensborough ⊕ Palmer & ⊕ 1182
⊕ Hornbostel
-
⊕ ⊕ 1380
Triborough ⊕ O. H. Ammann ⊕ _
⊕ ⊕ 383
-
Bronx Whitestone ⊕ O. H. Ammann ⊕ 2300
Throgs Neck ⊕ O. H. Ammann ⊕ 1800
-
George Washington ⊕ O. H. Ammann ⊕ 3500
.TE
```

Output:

Major New York Bridges		
Bridge	Designer	Length
Brooklyn	J. A. Roebling	1595
Manhattan	G. Lindenthal	1470
Williamsburg	L. L. Buck	1600
Queensborough	Palmer & Hornbostel	1182
Triborough	O. H. Ammann	1380
		383
Bronx Whitestone	O. H. Ammann	2300
Throgs Neck	O. H. Ammann	1800
George Washington	O. H. Ammann	3500

Input:

```
.TS
c c
np-2|n|.
⊕ Stack
⊕ _
1 ⊕ 46
⊕ _
2 ⊕ 23
⊕ _
3 ⊕ 15
⊕ _
4 ⊕ 6.5
⊕ _
5 ⊕ 2.1
⊕ _
.TE
```

Output:

Stack	
1	46
2	23
3	15
4	6.5
5	2.1

Input:

```
.TS
box;
L L L
L L _
L L |LB
L L _
L L L.
january ⊕ february ⊕ march
april ⊕ may
june ⊕ july ⊕ Months
august ⊕ september
october ⊕ november ⊕ december
.TE
```

Output:

january	february	march
april	may	Months
june	july	
august	september	
october	november	december

Input:

```
.TS
box;
cfB s s s.
Composition of Foods
-
.T&
c | c s s
c | c s s
c | c | c | c.
Food ⊕ Percent by Weight
\ ^ ⊕ _
\ ^ ⊕ Protein ⊕ Fat ⊕ Carbo-
\ ^ ⊕ \ ^ ⊕ \ ^ ⊕ hydrate
-
.T&
l | n | n | n.
Apples ⊕ .4 ⊕ .5 ⊕ 13.0
Halibut ⊕ 18.4 ⊕ 5.2 ⊕ . . .
Lima beans ⊕ 7.5 ⊕ .8 ⊕ 22.0
Milk ⊕ 3.3 ⊕ 4.0 ⊕ 5.0
Mushrooms ⊕ 3.5 ⊕ .4 ⊕ 6.0
Rye bread ⊕ 9.0 ⊕ .6 ⊕ 52.7
.TE
```

Output:

Composition of Foods			
Food	Percent by Weight		
	Protein	Fat	Carbo- hydrate
Apples	.4	.5	13.0
Halibut	18.4	5.2	...
Lima beans	7.5	.8	22.0
Milk	3.3	4.0	5.0
Mushrooms	3.5	.4	6.0
Rye bread	9.0	.6	52.7

Input:

```
.TS
allbox;
cfl s s
c cw(1i) cw(1i)
lp9 lp9 lp9 .
New York Area Rocks
Era @Formation @Age (years)
Precambrian @Reading Prong @>1 billion
Paleozoic @Manhattan Prong @400 million
Mesozoic @T{
.na
Newark Basin, incl.
Stockton, Lockatong, and Brunswick
formations; also Watchungs
and Palisades.
.ad
T} @200 million
Cenozoic @Coastal Plain @T{
.na
On Long Island 30,000 years;
Cretaceous sediments redeposited
by recent glaciation.
.ad
T}
.TE
```

Output:

<i>New York Area Rocks</i>		
Era	Formation	Age (years)
Precambrian	Reading Prong	>1 billion
Paleozoic	Manhattan Prong	400 million
Mesozoic	Newark Basin, incl. Stockton, Lockatong, and Brunswick formations; also Watchungs and Palisades.	200 million
Cenozoic	Coastal Plain	On Long Island 30,000 years; Cretaceous sediments redeposited by recent glaciation.

Input:

```
.EQ
delim $$
.EN
...
.TS
doublebox;
c c
ll.
Name @Definition
.sp
.vs +2p
Gamma @$GAMMA (z) = int sub 0 sup inf t sup {z-1} e sup -t dt$
Sine @$sin (x) = 1 over 2i ( e sup ix - e sup -ix )$
Error @$ roman erf (z) = 2 over sqrt pi int sub 0 sup z e sup {-t sup 2} dt$
Bessel @$ J sub 0 (z) = 1 over pi int sub 0 sup pi cos ( z sin theta ) d theta $
Zeta @$ zeta (s) = sum from k=1 to inf k sup -s ~( Re`s > 1)$
.vs -2p
.sp 2p
.TE
```

Output:

Name	Definition
Gamma	$\Gamma(z) = \int_0^{\infty} t^{z-1} e^{-t} dt$
Sine	$\sin(x) = \frac{1}{2i} (e^{ix} - e^{-ix})$
Error	$\operatorname{erf}(z) = \frac{2}{\sqrt{\pi}} \int_0^z e^{-t^2} dt$
Bessel	$J_0(z) = \frac{1}{\pi} \int_0^{\pi} \cos(z \sin \theta) d\theta$
Zeta	$\zeta(s) = \sum_{k=1}^{\infty} k^{-s} \quad (\operatorname{Re} s > 1)$

Input:

```
.TS
box, tab(:);
cb s s s s
cp-2 s s s s
c|c|c|c|c
c|c|c|c|c
r2||n2|n2|n2|n.
Readability of Text
Line Width and Leading for 10-Point Type
=
```

Line : Set : 1-Point : 2-Point : 4-Point
 Width : Solid : Leading : Leading : Leading

```
9 Pica:\-9.3:\-6.0:\-5.3:\-7.1
14 Pica:\-4.5:\-0.6:\-0.3:\-1.7
19 Pica:\-5.0:\-5.1: 0.0:\-2.0
31 Pica:\-3.7:\-3.8:\-2.4:\-3.6
43 Pica:\-9.1:\-9.0:\-5.9:\-8.8
.TE
```

Output:

Readability of Text				
Line Width and Leading for 10-Point Type				
Line Width	Set Solid	1-Point Leading	2-Point Leading	4-Point Leading
9 Pica	-9.3	-6.0	-5.3	-7.1
14 Pica	-4.5	-0.6	-0.3	-1.7
19 Pica	-5.0	-5.1	0.0	-2.0
31 Pica	-3.7	-3.8	-2.4	-3.6
43 Pica	-9.1	-9.0	-5.9	-8.8

Input:

.TS
 c s
 cip-2 s
 l n
 a n.
 Some London Transport Statistics
 (Year 1964)
 Railway route miles ⊕244
 Tube ⊕66
 Sub-surface ⊕22
 Surface ⊕156
 .sp .5
 .T&
 l r
 a r.
 Passenger traffic \- railway
 Journeys ⊕674 million
 Average length ⊕4.55 miles
 Passenger miles ⊕3,066 million
 .T&
 l r
 a r.
 Passenger traffic \- road
 Journeys ⊕2,252 million
 Average length ⊕2.26 miles
 Passenger miles ⊕5,094 million
 .T&
 l n
 a n.
 .sp .5
 Vehicles ⊕12,521
 Railway motor cars ⊕2,905
 Railway trailer cars ⊕1,269
 Total railway ⊕4,174
 Omnibuses ⊕8,347
 .T&
 l n
 a n.
 .sp .5
 Staff ⊕73,739
 Administrative, etc. ⊕5,582
 Civil engineering ⊕5,134
 Electrical eng. ⊕1,714
 Mech. eng. \- railway ⊕4,310
 Mech. eng. \- road ⊕9,152
 Railway operations ⊕8,930
 Road operations ⊕35,946
 Other ⊕2,971
 .TE

Output:

Some London Transport Statistics
 (Year 1964)

Railway route miles	244
Tube	66
Sub-surface	22
Surface	156
Passenger traffic - railway	
Journeys	674 million
Average length	4.55 miles
Passenger miles	3,066 million
Passenger traffic - road	
Journeys	2,252 million
Average length	2.26 miles
Passenger miles	5,094 million
Vehicles	12,521
Railway motor cars	2,905
Railway trailer cars	1,269
Total railway	4,174
Omnibuses	8,347
Staff	73,739
Administrative, etc.	5,582
Civil engineering	5,134
Electrical eng.	1,714
Mech. eng. - railway	4,310
Mech. eng. - road	9,152
Railway operations	8,930
Road operations	35,946
Other	2,971

Input:

.ps 8

.vs 10p

.TS

center box;

c s s

ci s s

c c c

lB l n.

New Jersey Representatives

(Democrats)

.sp .5

Name ⊕Office address ⊕Phone

.sp .5

James J. Florio ⊕23 S. White Horse Pike, Somerdale 08083 ⊕609-627-8222

William J. Hughes ⊕2920 Atlantic Ave., Atlantic City 08401 ⊕609-345-4844

James J. Howard ⊕801 Bangs Ave., Asbury Park 07712 ⊕201-774-1600

Frank Thompson, Jr. ⊕10 Rutgers Pl., Trenton 08618 ⊕609-599-1619

Andrew Maguire ⊕115 W. Passaic St., Rochelle Park 07662 ⊕201-843-0240

Robert A. Roe ⊕U.S.P.O., 194 Ward St., Paterson 07510 ⊕201-523-5152

Henry Helstoski ⊕666 Paterson Ave., East Rutherford 07073 ⊕201-939-9090

Peter W. Rodino, Jr. ⊕Suite 1435A, 970 Broad St., Newark 07102 ⊕201-645-3213

Joseph G. Minish ⊕308 Main St., Orange 07050 ⊕201-645-6363

Helen S. Meyner ⊕32 Bridge St., Lambertville 08530 ⊕609-397-1830

Dominick V. Daniels ⊕895 Bergen Ave., Jersey City 07306 ⊕201-659-7700

Edward J. Patten ⊕Natl. Bank Bldg., Perth Amboy 08861 ⊕201-826-4610

.sp .5

.T&

ci s s

lB l n.

(Republicans)

.sp .5v

Millicent Fenwick ⊕41 N. Bridge St., Somerville 08876 ⊕201-722-8200

Edwin B. Forsythe ⊕301 Mill St., Moorestown 08057 ⊕609-235-6622

Matthew J. Rinaldo ⊕1961 Morris Ave., Union 07083 ⊕201-687-4235

.TE

.ps 10

.vs 12p

Output:

New Jersey Representatives (Democrats)		
Name	Office address	Phone
James J. Florio	23 S. White Horse Pike, Somerdale 08083	609-627-8222
William J. Hughes	2920 Atlantic Ave., Atlantic City 08401	609-345-4844
James J. Howard	801 Bangs Ave., Asbury Park 07712	201-774-1600
Frank Thompson, Jr.	10 Rutgers Pl., Trenton 08618	609-599-1619
Andrew Maguire	115 W. Passaic St., Rochelle Park 07662	201-843-0240
Robert A. Roe	U.S.P.O., 194 Ward St., Paterson 07510	201-523-5152
Henry Helstoski	666 Paterson Ave., East Rutherford 07073	201-939-9090
Peter W. Rodino, Jr.	Suite 1435A, 970 Broad St., Newark 07102	201-645-3213
Joseph G. Minish	308 Main St., Orange 07050	201-645-6363
Helen S. Meyner	32 Bridge St., Lambertville 08530	609-397-1830
Dominick V. Daniels	895 Bergen Ave., Jersey City 07306	201-659-7700
Edward J. Patten	Natl. Bank Bldg., Perth Amboy 08861	201-826-4610
(Republicans)		
Millicent Fenwick	41 N. Bridge St., Somerville 08876	201-722-8200
Edwin B. Forsythe	301 Mill St., Moorestown 08057	609-235-6622
Matthew J. Rinaldo	1961 Morris Ave., Union 07083	201-687-4235

This is a paragraph of normal text placed here only to indicate where the left and right margins are. In this way the reader can judge the appearance of centered tables or expanded tables, and observe how such tables are formatted.

Input:

```
.TS
expand;
c s s s
c c c c
l l n n.
Bell Labs Locations
Name ⊕Address ⊕Area Code ⊕Phone
Holmdel ⊕Holmdel, N. J. 07733 ⊕201 ⊕949-3000
Murray Hill ⊕Murray Hill, N. J. 07974 ⊕201 ⊕582-6377
Whippany ⊕Whippany, N. J. 07981 ⊕201 ⊕386-3000
Indian Hill ⊕Naperville, Illinois 60540 ⊕312 ⊕690-2000
.TE
```

Output:

Bell Labs Locations			
Name	Address	Area Code	Phone
Holmdel	Holmdel, N. J. 07733	201	949-3000
Murray Hill	Murray Hill, N. J. 07974	201	582-6377
Whippany	Whippany, N. J. 07981	201	386-3000
Indian Hill	Naperville, Illinois 60540	312	690-2000

Input:

.TS
 box;
 cb s s s
 c|c|c s
 ltw(1i)|ltw(2i)|lp8|lw(1.6i)p8.
 Some Interesting Places

Name ⊕ Description ⊕ Practical Information

⊖

T{
 American Museum of Natural History

T} ⊕ T{

The collections fill 11.5 acres (Michelin) or 25 acres (MTA)
 of exhibition halls on four floors.

There is a full-sized replica

of a blue whale and the world's largest star sapphire (stolen in 1964).

T} ⊕ Hours ⊕ 10-5, ex. Sun 11-5, Wed. to 9

\^ ⊕ \^ ⊕ Location ⊕ T{

Central Park West & 79th St.

T}

\^ ⊕ \^ ⊕ Admission ⊕ Donation: \$1.00 asked

\^ ⊕ \^ ⊕ Subway ⊕ AA to 81st St.

\^ ⊕ \^ ⊕ Telephone ⊕ 212-873-4225

⊖
 Bronx Zoo ⊕ T{

About a mile long and .6 mile wide, this is the largest zoo in America.

A lion eats 18 pounds

of meat a day while a sea lion eats 15 pounds of fish.

T} ⊕ Hours ⊕ T{

10-4:30 winter, to 5:00 summer

T}

\^ ⊕ \^ ⊕ Location ⊕ T{

185th St. & Southern Blvd, the Bronx.

T}

\^ ⊕ \^ ⊕ Admission ⊕ \$1.00, but Tu, We, Th free

\^ ⊕ \^ ⊕ Subway ⊕ 2, 5 to East Tremont Ave.

\^ ⊕ \^ ⊕ Telephone ⊕ 212-933-1759

⊖
 Brooklyn Museum ⊕ T{

Five floors of galleries contain American and ancient art.

There are American period rooms and architectural ornaments saved
 from wreckers, such as a classical figure from Pennsylvania Station.

T} ⊕ Hours ⊕ Wed-Sat, 10-5, Sun 12-5

\^ ⊕ \^ ⊕ Location ⊕ T{

Eastern Parkway & Washington Ave., Brooklyn.

T}

\^ ⊕ \^ ⊕ Admission ⊕ Free

\^ ⊕ \^ ⊕ Subway ⊕ 2, 3 to Eastern Parkway.

\^ ⊕ \^ ⊕ Telephone ⊕ 212-638-5000

⊖

T{
 New-York Historical Society

T} ⊕ T{

All the original paintings for Audubon's

.I

Birds of America

.R

are here, as are exhibits of American decorative arts, New York history,
 Hudson River school paintings, carriages, and glass paperweights.

T} ⊕ Hours ⊕ T{

Tues-Fri & Sun, 1-5; Sat 10-5

T}

\^ ⊕ \^ ⊕ Location ⊕ T{

Central Park West & 77th St.

T}

\^ ⊕ \^ ⊕ Admission ⊕ Free

\^ ⊕ \^ ⊕ Subway ⊕ AA to 81st St.

\^ ⊕ \^ ⊕ Telephone ⊕ 212-873-3400

.TE

Output:

Some Interesting Places			
Name	Description	Practical Information	
<i>American Museum of Natural History</i>	The collections fill 11.5 acres (Michelin) or 25 acres (MTA) of exhibition halls on four floors. There is a full-sized replica of a blue whale and the world's largest star sapphire (stolen in 1964).	Hours Location Admission Subway Telephone	10-5, ex. Sun 11-5, Wed. to 9 Central Park West & 79th St. Donation: \$1.00 asked AA to 81st St. 212-873-4225
<i>Bronx Zoo</i>	About a mile long and .6 mile wide, this is the largest zoo in America. A lion eats 18 pounds of meat a day while a sea lion eats 15 pounds of fish.	Hours Location Admission Subway Telephone	10-4:30 winter, to 5:00 summer 185th St. & Southern Blvd, the Bronx. \$1.00, but Tu, We, Th free 2, 5 to East Tremont Ave. 212-933-1759
<i>Brooklyn Museum</i>	Five floors of galleries contain American and ancient art. There are American period rooms and architectural ornaments saved from wreckers, such as a classical figure from Pennsylvania Station.	Hours Location Admission Subway Telephone	Wed-Sat, 10-5, Sun 12-5 Eastern Parkway & Washington Ave., Brooklyn. Free 2,3 to Eastern Parkway. 212-638-5000
<i>New-York Historical Society</i>	All the original paintings for Audubon's <i>Birds of America</i> are here, as are exhibits of American decorative arts, New York history, Hudson River school paintings, carriages, and glass paperweights.	Hours Location Admission Subway Telephone	Tues-Fri & Sun, 1-5; Sat 10-5 Central Park West & 77th St. Free AA to 81st St. 212-873-3400

ACKNOWLEDGEMENTS

Many thanks are due to J. C. Blinn, who has done a large amount of testing and assisted with the design of the program. He has also written many of the more intelligible sentences in this document and helped edit all of it. All phototypesetting programs on UNIX are dependent on the work of the late J. F. Ossanna, whose assistance with this program in particular had been most helpful. This program is patterned on a table formatter written by J. F. Gimpel. The assistance of T. A. Dolotta, B. W. Kernighan, and J. N. Sturman is gratefully acknowledged.

REFERENCES

- [1] J. F. Ossanna. *NROFF/TROFF User's Manual*, Bell Laboratories, 1976.
- [2] D. M. Ritchie and K. Thompson. The UNIX Time-Sharing System, *CACM* 17(7):365-75 (July 1974).
- [3] B. W. Kernighan and L. L. Cherry. A System for Typesetting Mathematics, *CACM* 18(3):151-56 (Mar. 1975).
- [4] M. E. Lesk. *Typing Documents on UNIX*, Bell Laboratories, 1976.
- [5] M. E. Lesk and B. W. Kernighan. Computer Typesetting of Technical Journals on UNIX, *Proc. AFIPS NCC*, vol. 46, pp. 879-88 (1977).
- [6] D. W. Smith and J. R. Mashey. *MM—Memorandum Macros*, Bell Laboratories, 1980.

List of Tbl Command Characters and Words

<i>Command</i>	<i>Meaning</i>	<i>Section</i>
a A	Alphabetic subcolumn	2
allbox	Draw box around all items	1
b B	Boldface item	2
box	Draw box around table	1
c C	Centered column	2
center	Center table in page	1
delim (xy)	Define <i>eqn</i> delimiters	1
doublebox	Draw double box around table	1
e E	Equal-width columns	2
expand	Make table full line width	1
f F	Font change	2
i I	Italic item	2
l L	Left adjusted column	2
linesize (n)	Set size for rules	1
n N	Numerical column	2
nnn	Column separation	2
p P	Point size change	2
r R	Right adjusted column	2
s S	Spanned item	2
t T	Vertical spanning at top	2
tab (x)	Change data separator character	1
T{...T}	Text block	3
u U	Staggered columns	2
v V	Vertical spacing change	2
w W	Minimum width value	2
z Z	Zero-width item	2
.xx	Included <i>troff</i> request	3
 	Vertical line	2
 	Double vertical line	2
^	Vertical span	2
\^	Vertical span	3
=	Double horizontal line	2,3
-	Horizontal line	2,3
_	Short horizontal line	3
\Rx	Repeat character	3
-	Name of standard input	Usage

Typesetting Mathematics—User's Guide (Second Edition)

Brian W. Kernighan

Lorinda L. Cherry

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This is the user's guide for a system for typesetting mathematics, using the phototypesetters on the UNIX† and GCOS operating systems.

Mathematical expressions are described in a language designed to be easy to use by people who know neither mathematics nor typesetting. Enough of the language to set in-line expressions like $\lim_{x \rightarrow \pi/2} (\tan x)^{\sin 2x} = 1$ or display equations like

$$\begin{aligned}
 G(z) &= e^{\ln G(z)} = \exp\left(\sum_{k \geq 1} \frac{S_k z^k}{k}\right) = \prod_{k \geq 1} e^{S_k z^k / k} \\
 &= \left(1 + S_1 z + \frac{S_1^2 z^2}{2!} + \dots\right) \left(1 + \frac{S_2 z^2}{2} + \frac{S_2^2 z^4}{2^2 \cdot 2!} + \dots\right) \dots \\
 &= \sum_{m \geq 0} \left[\sum_{\substack{k_1, k_2, \dots, k_m \geq 0 \\ k_1 + 2k_2 + \dots + mk_m = m}} \frac{S_1^{k_1}}{1^{k_1} k_1!} \frac{S_2^{k_2}}{2^{k_2} k_2!} \dots \frac{S_m^{k_m}}{m^{k_m} k_m!} \right] z^m
 \end{aligned}$$

can be learned in an hour or so.

The language interfaces directly with the phototypesetting language TROFF, so mathematical expressions can be embedded in the running text of a manuscript, and the entire document produced in one process. This user's guide is an example of its output.

The same language may be used with the UNIX formatter NROFF to set mathematical expressions on DASI and GSI terminals and Model 37 TELETYPE® terminals.

1. Introduction

EQN is a program for typesetting mathematics on the Graphics Systems phototypesetters on UNIX and GCOS. The EQN language was designed to be easy to use by people who know neither mathematics nor typesetting. Thus EQN knows relatively little about mathematics. In particular, mathematical symbols like +, −, ×, parentheses, and so on have no special meanings. EQN is quite happy to set

garbage (but it will look good).

EQN works as a preprocessor for the typesetter formatter, TROFF [1], so the normal mode of operation is to prepare a document with both mathematics and ordinary text interspersed, and let EQN set the mathematics while TROFF does the body of the text.

On UNIX, EQN will also produce mathematics on DASI and GSI terminals and on Model 37 TELETYPE terminals. The

† UNIX is a trademark of Bell Laboratories.

input is identical, but you have to use the programs NEQN and NROFF instead of EQN and TROFF. Of course, some things won't look as good because terminals don't provide the variety of characters, sizes and fonts that a typesetter does, but the output is usually adequate for proofreading.

To use EQN on UNIX,

```
eqn files | troff
```

GCOS use is discussed in section 26.

2. Displayed Equations

To tell EQN where a mathematical expression begins and ends, we mark it with lines beginning .EQ and .EN. Thus if you type the lines

```
.EQ
x=y+z
.EN
```

your output will look like

$$x=y+z$$

The .EQ and .EN are copied through untouched; they are not otherwise processed by EQN. This means that you have to take care of things like centering, numbering, and so on yourself. The most common way is to use the TROFF and NROFF macro packages '-ms' and '-mm' [3,4], which allow you to center, indent, left-justify, and number equations.

With the '-ms' package, equations are centered by default. To left-justify an equation, use .EQ L instead of .EQ. To indent it, use .EQ I. Any of these can be followed by an arbitrary 'equation number' which will be placed at the right margin. For example, the input

```
.EQ I (3.1a)
x = f(y/2) + y/2
.EN
```

produces the output

$$x=f(y/2)+y/2 \quad (3.1a)$$

There is also a shorthand notation so in-line expressions like π^2 can be entered without .EQ and .EN. We will talk about it in section 19.

3. Input spaces

Spaces and new-lines within an expression are thrown away by EQN. (Normal text is left absolutely alone.) Thus between .EQ and .EN,

$$x=y+z$$

and

$$x = y + z$$

and

$$x = y \\ + z$$

and so on all produce the same output

$$x=y+z$$

You should use spaces and new-lines freely to make your input equations readable and easy to edit. In particular, very long lines are a bad idea, since they are often hard to fix if you make a mistake.

4. Output spaces

To force extra spaces into the *output*, use a tilde "~" for each space you want:

$$x\tilde{=}y\tilde{+}z$$

gives

$$x = y + z$$

You can also use a circumflex "^", which gives a space half the width of a tilde. It is mainly useful for fine-tuning. Tabs may also be used to position pieces of an expression, but the tab stops must be set by TROFF commands.

5. Symbols, Special Names, Greek

EQN knows some mathematical symbols, some mathematical names, and the Greek alphabet. For example,

$$x=2 \text{ pi } \int \sin (\text{omega } t) dt$$

produces

$$x=2\pi \int \sin(\omega t) dt$$

Here the spaces in the input are **necessary** to tell EQN that *int*, *pi*, *sin* and *omega* are separate entities that should get special treatment. The *sin*, digit 2, and parentheses are set in roman type instead of italic; *pi* and *omega* are made Greek; and *int* becomes the

integral sign.

When in doubt, leave spaces around separate parts of the input. A *very* common error is to type $f(pi)$ without leaving spaces on both sides of the pi . As a result, EQN does not recognize pi as a special word, and it appears as $f(pi)$ instead of $f(\pi)$.

A complete list of EQN names appears in section 23. Knowledgeable users can also use TROFF four-character names for anything EQN doesn't know about, like $\backslash(bs$ for the Bell System sign $\text{\textcircled{B}}$).

6. Spaces, Again

The only way EQN can deduce that some sequence of letters might be special is if that sequence is separated from the letters on either side of it. This can be done by surrounding a special word by ordinary spaces (or tabs or new-lines), as we did in the previous section.

You can also make special words stand out by surrounding them with tildes or circumflexes:

$$\tilde{x} = \tilde{2} \tilde{\pi} \tilde{\int} \tilde{\sin}(\tilde{\omega} \tilde{t}) \tilde{dt}$$

is much the same as the last example, except that the tildes not only separate the magic words like *sin*, *omega*, and so on, but also add extra spaces, one space per tilde:

$$x = 2 \pi \int \sin(\omega t) dt$$

Special words can also be separated by braces { } and double quotes "...", which have special meanings that we will see soon.

7. Subscripts and Superscripts

Subscripts and superscripts are obtained with the words *sub* and *sup*.

$$x \text{ sup } 2 + y \text{ sub } k$$

gives

$$x^2 + y_k$$

EQN takes care of all the size changes and vertical motions needed to make the output look right. The words *sub* and *sup* must be surrounded by spaces; $x \text{ sub } 2$ will give you $x_{\text{sub}2}$ instead of x_2 . Furthermore, don't forget to leave a space (or a tilde, etc.) to mark the end of a subscript or superscript. A common error is to say something like

$$y = (x \text{ sup } 2) + 1$$

which causes

$$y = (x^2) + 1$$

instead of the intended

$$y = (x^2) + 1$$

Subscripted subscripts and super-scripted superscripts also work:

$$x \text{ sub } i \text{ sub } 1$$

is

$$x_{i_1}$$

A subscript and superscript on the same thing are printed one above the other if the subscript comes *first*:

$$x \text{ sub } i \text{ sup } 2$$

is

$$x_i^2$$

Other than this special case, *sub* and *sup* group to the right, so $x \text{ sup } y \text{ sub } z$ means x^{y_z} , not x^y_z .

8. Braces for Grouping

Normally, the end of a subscript or superscript is marked simply by a blank (or tab or tilde, etc.) What if the subscript or superscript is something that has to be typed with blanks in it? In that case, you can use the braces { and } to mark the beginning and end of the subscript or superscript:

$$e \text{ sup } \{i \text{ omega } t\}$$

is

$$e^{i\omega t}$$

Rule: Braces can *always* be used to force EQN to treat something as a unit, or just to make your intent perfectly clear. Thus:

$$x \text{ sub } \{i \text{ sub } 1\} \text{ sup } 2$$

is

$$x_{i_1}^2$$

with braces, but

$$x \text{ sub } i \text{ sub } 1 \text{ sup } 2$$

is

$$x_i^2$$

which is rather different.

Braces can occur within braces if necessary:

$$e^{\sup\{i \pi \sup\{\rho + 1\}\}}$$

is

$$e^{i\pi^{\rho+1}}$$

The general rule is that anywhere you could use some single thing like x , you can use an arbitrarily complicated thing if you enclose it in braces. EQN will look after all the details of positioning it and making it the right size.

In all cases, make sure you have the right number of braces. Leaving one out or adding an extra will cause EQN to complain bitterly.

Occasionally you will have to print braces. To do this, enclose them in double quotes, like "{". Quoting is discussed in more detail in section 14.

9. Fractions

To make a fraction, use the word *over*:

$$a+b \text{ over } 2c = 1$$

gives

$$\frac{a+b}{2c} = 1$$

The line is made the right length and positioned automatically. Braces can be used to make clear what goes over what:

$$\{\alpha + \beta\} \text{ over } \{\sin(x)\}$$

is

$$\frac{\alpha + \beta}{\sin(x)}$$

What happens when there is both an *over* and a *sup* in the same expression? In such an apparently ambiguous case, EQN does the *sup* before the *over*, so

$$-b \sup 2 \text{ over } \pi$$

is $\frac{-b^2}{\pi}$ instead of $-b^{\frac{2}{\pi}}$. The rules which decide which operation is done first in cases like this are summarized in section 23. When in doubt, however, use braces to make clear what goes with what.

10. Square Roots

To draw a square root, use *sqrt*:

$\text{sqrt } a+b + 1 \text{ over } \text{sqrt } \{ax \sup 2 + bx + c\}$
is

$$\sqrt{a+b} + \frac{1}{\sqrt{ax^2+bx+c}}$$

Warning — square roots of tall quantities look lousy, because a root-sign big enough to cover the quantity is too dark and heavy:

$$\text{sqrt } \{a \sup 2 \text{ over } b \text{ sub } 2\}$$

is

$$\sqrt{\frac{a^2}{b_2}}$$

Big square roots are generally better written as something to the power $\frac{1}{2}$:

$$(a^2/b_2)^{1/2}$$

which is

$$(a \sup 2 / b \text{ sub } 2) \sup \text{ half}$$

11. Summation, Integral, Etc.

Summations, integrals, and similar constructions are easy:

$$\text{sum from } i=0 \text{ to } \{i = \text{inf}\} x \sup i$$

produces

$$\sum_{i=0}^{i=\infty} x^i$$

Notice that we used braces to indicate where the upper part $i=\infty$ begins and ends. No braces were necessary for the lower part $i=0$, because it contained no blanks. The braces will never hurt, and if the *from* and *to* parts contain any blanks, you must use braces around them.

The *from* and *to* parts are both optional, but if both are used, they have to occur in that order.

Other useful characters can replace the *sum* in our example:

$$\text{int prod union inter}$$

become, respectively,

$$\int \prod \cup \cap$$

Since the thing before the *from* can be anything, even something in braces, *from-to* can

often be used in unexpected ways:

$$\lim_{n \rightarrow \infty} x_n = 0$$

is

$$\lim_{n \rightarrow \infty} x_n = 0$$

12. Size and Font Changes

By default, equations are set in 10-point type (the same size as this guide), with standard mathematical conventions to determine what characters are in roman and what in italic. Although EQN makes a valiant attempt to use esthetically pleasing sizes and fonts, it is not perfect. To change sizes and fonts, use *size n* and *roman*, *italic*, *bold* and *fat*. Like *sub* and *sup*, size and font changes affect only the thing that follows them, and revert to the normal situation at the end of it. Thus

x y

is

x y

and

$$\text{size 14 bold } x = y + \text{size 14 } \{\alpha + \beta\}$$

gives

$$\mathbf{x=y+\alpha+\beta}$$

As always, you can use braces if you want to affect something more complicated than a single letter. For example, you can change the size of an entire equation by

size 12 { ... }

Legal sizes which may follow *size* are 6, 7, 8, 9, 10, 11, 12, 14, 16, 18, 20, 22, 24, 28, 36. You can also change the size by a given amount; for example, you can say *size +2* to make the size two points bigger, or *size -3* to make it three points smaller. This has the advantage that you don't have to know what the current size is.

If you are using fonts other than roman, italic and bold, you can say *font X* where *X* is a one character TROFF name or number for the font. Since EQN is tuned for roman, italic and bold, other fonts may not give quite as good an appearance.

The *fat* operation takes the current font and widens it by overstriking: *fat grad* is ∇ and *fat {x sub i}* is x_i .

If an entire document is to be in a non-standard size or font, it is a severe nuisance to have to write out a size and font change for each equation. Accordingly, you can set a "global" size or font which thereafter affects all equations. At the beginning of any equation, you might say, for instance,

```
.EQ
gsize 16
gfont R
...
.EN
```

to set the size to 16 and the font to roman thereafter. In place of R, you can use any of the TROFF font names. The size after *gsize* can be a relative change with + or -.

Generally, *gsize* and *gfont* will appear at the beginning of a document but they can also appear throughout a document: the global font and size can be changed as often as needed. For example, in a footnote† you will typically want the size of equations to match the size of the footnote text, which is two points smaller than the main text. Don't forget to reset the global size at the end of the footnote.

13. Diacritical Marks

To get funny marks on top of letters, there are several words:

x dot	\dot{x}
x dotdot	\ddot{x}
x hat	\hat{x}
x tilde	\tilde{x}
x vec	\vec{x}
x dyad	\overline{x}
x bar	\bar{x}
x under	\underline{x}

The diacritical mark is placed at the right height. The *bar* and *under* are made the right length for the entire construct, as in $\overline{x+y+z}$; other marks are centered.

†Like this one, in which we have a few random expressions like x_i and π^2 . The sizes for these were set by the command *gsize -2*.

14. Quoted Text

Any input entirely within quotes ("...") is not subject to any of the font changes and spacing adjustments normally done by the equation setter. This provides a way to do your own spacing and adjusting if needed:

italic "sin(x)" + sin (x)

is

sin(x)+sin(x)

Quotes are also used to get braces and other EQN keywords printed:

"{ size alpha }"

is

{ *size alpha* }

and

roman "{ size alpha }"

is

{ size alpha }

The construction "" is often used as a place-holder when grammatically EQN needs something, but you don't actually want anything in your output. For example, to make ²He, you can't just type *sup 2 roman He* because a *sup* has to be a superscript on something. Thus you must say

"" sup 2 roman He

To get a literal quote use "\". TROFF characters like \b can appear unquoted, but more complicated things like horizontal and vertical motions with \h and \v should always be quoted. (If you've never heard of \h and \v, ignore this section.)

15. Lining Up Equations

Sometimes it's necessary to line up a series of equations at some horizontal position, often at an equals sign. This is done with two operations called *mark* and *lineup*.

The word *mark* may appear once at any place in an equation. It remembers the horizontal position where it appeared. Successive equations can contain one occurrence of the word *lineup*. The place where *lineup* appears is made to line up with

the place marked by the previous *mark* if at all possible. Thus, for example, you can say

```
.EQ I
x+y mark = z
.EN
.EQ I
x lineup = 1
.EN
```

to produce

$$x+y=z$$

$$x=1$$

For reasons too complicated to talk about, when you use EQN and '-ms', use either .EQ I or .EQ L. *mark* and *lineup* don't work with centered equations. Also bear in mind that *mark* doesn't look ahead;

x mark =1

...
x+y lineup =z

isn't going to work, because there isn't room for the *x+y* part after the *mark* remembers where the *x* is.

16. Big Brackets, Etc.

To get big brackets [], braces {}, parentheses (), and bars || around things, use the *left* and *right* commands:

```
left { a over b + 1 right }
~ ~ left ( c over d right )
+ left [ e right ]
```

is

$$\left\{ \frac{a}{b} + 1 \right\} = \left[\frac{c}{d} \right] + [e]$$

The resulting brackets are made big enough to cover whatever they enclose. Other characters can be used besides these, but they are not likely to look very good. One exception is the *floor* and *ceiling* characters:

```
left floor x over y right floor
< = left ceiling a over b right ceiling
```

produces

$$\left\lfloor \frac{x}{y} \right\rfloor \leq \left\lceil \frac{a}{b} \right\rceil$$

Several warnings about brackets are in order. First, braces are typically bigger than brackets and parentheses, because they are

made up of three, five, seven, etc., pieces, while brackets can be made up of two, three, etc. Second, big left and right parentheses often look poor, because the character set is poorly designed.

The *right* part may be omitted: a “left something” need not have a corresponding “right something”. If the *right* part is omitted, put braces around the thing you want the left bracket to encompass. Otherwise, the resulting brackets may be too large.

If you want to omit the *left* part, things are more complicated, because technically you can't have a *right* without a corresponding *left*. Instead you have to say

left "" right)

for example. The *left* "" means a “left nothing”. This satisfies the rules without hurting your output.

17. Piles

There is a general facility for making vertical piles of things; it comes in several flavors. For example:

```
A ~-~ left [
  pile { a above b above c }
  ~-~ pile { x above y above z }
right ]
```

will make

$$A = \begin{bmatrix} a & x \\ b & y \\ c & z \end{bmatrix}$$

The elements of the pile (there can be as many as you want) are centered one above another, at the right height for most purposes. The keyword *above* is used to separate the pieces; braces are used around the entire list. The elements of a pile can be as complicated as needed, even containing more piles.

Three other forms of pile exist: *lpile* makes a pile with the elements left-justified; *rpile* makes a right-justified pile; and *cpile* makes a centered pile, just like *pile*. The vertical spacing between the pieces is somewhat larger for *l*-, *r*- and *cpiles* than it is for ordinary piles.

```
roman sign (x) ~-~
left {
  lpile { 1 above 0 above -1 }
  ~-~ lpile
  { if x > 0 above if x = 0 above if x < 0 }
```

makes

$$\text{sign}(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x = 0 \\ -1 & \text{if } x < 0 \end{cases}$$

Notice the left brace without a matching right one.

18. Matrices

It is also possible to make matrices. For example, to make a neat array like

$$\begin{array}{c} x_i \ x^2 \\ y_i \ y^2 \end{array}$$

you have to type

```
matrix {
  ccol { x sub i above y sub i }
  ccol { x sup 2 above y sup 2 }
}
```

This produces a matrix with two centered columns. The elements of the columns are then listed just as for a pile, each element separated by the word *above*. You can also use *lcol* or *rcol* to left or right adjust columns. Each column can be separately adjusted, and there can be as many columns as you like.

The reason for using a matrix instead of two adjacent piles, by the way, is that if the elements of the piles don't all have the same height, they won't line up properly. A matrix forces them to line up, because it looks at the entire structure before deciding what spacing to use.

A word of warning about matrices — *each column must have the same number of elements in it*. The world will end if you get this wrong.

19. Shorthand for In-line Equations

In a mathematical document, it is necessary to follow mathematical conventions not just in display equations, but also in the body of the text, for example by making variable names like *x* italic. Although

this could be done by surrounding the appropriate parts with .EQ and .EN, the continual repetition of .EQ and .EN is a nuisance. Furthermore, with '-ms', .EQ and .EN imply a displayed equation.

EQN provides a shorthand for short in-line expressions. You can define two characters to mark the left and right ends of an in-line equation, and then type expressions right in the middle of text lines. To set both the left and right characters to dollar signs, for example, add to the beginning of your document the three lines

```
.EQ
delim $$
.EN
```

Having done this, you can then say things like

Let α_i be the primary variable, and let β be zero. Then we can show that x_1 is $>=0$.

This works as you might expect — spaces, new-lines, and so on are significant in the text, but not in the equation part itself. Multiple equations can occur in a single input line.

Enough room is left before and after a line that contains in-line expressions that something like $\sum_{i=1}^n x_i$ does not interfere with the lines surrounding it.

To turn off the delimiters,

```
.EQ
delim off
.EN
```

Warning: don't use braces, tildes, circumflexes, or double quotes as delimiters — chaos will result. Also, in-line font changes must be closed before in-line equations are encountered.

20. Definitions

EQN provides a facility so you can give a frequently-used string of characters a name, and thereafter just type the name instead of the whole string. For example, if the sequence

$$x_{i1} + y_{i1}$$

appears repeatedly throughout a paper, you can save re-typing it each time by defining it like this:

```
define xy 'x sub i sub 1 + y sub i sub 1'
```

This makes *xy* a shorthand for whatever characters occur between the single quotes in the definition. You can use any character instead of quote to mark the ends of the definition, so long as it doesn't appear inside the definition.

Now you can use *xy* like this:

```
.EQ
f(x) = xy ...
.EN
```

and so on. Each occurrence of *xy* will expand into what it was defined as. Be careful to leave spaces or their equivalent around the name when you actually use it, so EQN will be able to identify it as special.

There are several things to watch out for. First, although definitions can use previous definitions, as in

```
.EQ
define xi 'x sub i '
define xil 'xi sub 1 '
.EN
```

don't define something in terms of itself! A favorite error is to say

```
define X 'roman X '
```

This is a guaranteed disaster, since *X* is now defined in terms of itself. If you say

```
define X 'roman "X" '
```

however, the quotes protect the second *X*, and everything works fine.

EQN keywords can be redefined. You can make / mean *over* by saying

```
define / 'over '
```

or redefine *over* as / with

```
define over '/ '
```

If you need different things to print on a terminal and on the typesetter, it may be worth defining a symbol differently in NEQN and EQN. This can be done with *ndefine* and *tdefine*. A definition made with *ndefine* only takes effect if you are running NEQN; if you use *tdefine*, the definition only applies for

EQN. Names defined with plain *define* apply to both EQN and NEQN.

21. Local Motions

Although EQN tries to get most things at the right place on the paper, it isn't perfect, and occasionally you will need to tune the output to make it just right. Small extra horizontal spaces can be obtained with tilde and circumflex. You can also say *back n* and *fwd n* to move small amounts horizontally. *n* is how far to move in 1/100's of an em (an em is about the width of the letter 'm'.) Thus *back 50* moves back about half the width of an m. Similarly you can move things up or down with *up n* and *down n*. As with *sub* or *sup*, the local motions affect the next thing in the input, and this can be something arbitrarily complicated if it is enclosed in braces.

22. A Large Example

Here is the complete source for the three display equations in the abstract of this guide.

```
.EQ 1
G(z)-mark = e sup { ln - G(z) }
- = exp left (
sum from k >= 1 { S sub k z sup k } over k right )
- = prod from k >= 1 e sup { S sub k z sup k / k }
.EN
.EQ 1
lineup = left ( 1 + S sub 1 z +
{ S sub 1 sup 2 z sup 2 } over 2! + ... right )
left ( 1 + { S sub 2 z sup 2 } over 2
+ { S sub 2 sup 2 z sup 4 } over { 2 sup 2 cdot 2! }
+ ... right ) ...
.EN
.EQ 1
lineup = sum from m >= 0 left (
sum from
pile { k sub 1 , k sub 2 , ..., k sub m >= 0
above
k sub 1 + 2k sub 2 + ... + mk sub m = m }
{ S sub 1 sup { k sub 1 } } over { 1 sup k sub 1 k sub 1 ! } -
{ S sub 2 sup { k sub 2 } } over { 2 sup k sub 2 k sub 2 ! } -
...
{ S sub m sup { k sub m } } over { m sup k sub m k sub m ! }
right ) z sup m
.EN
```

23. Keywords, Precedences, Etc.

If you don't use braces, EQN will do operations in the order shown in this list.

dyad vec under bar tilde hat dot dotdot
fwd back down up
fat roman italic bold size
sub sup sqrt over
from to

These operations group to the left:

over sqrt left right

All others group to the right.

Digits, parentheses, brackets, punctuation marks, and these mathematical words are converted to Roman font when encountered:

sin cos tan sinh cosh tanh arc
 max min lim log ln exp
 Re Im and if for det

These character sequences are recognized and translated as shown.

> =	≧
< =	≦
= =	≡
! =	≠
+ -	±
->	→
<-	←
<<	≪
>>	≫
inf	∞
partial	∂
half	½
prime	'
approx	≈
nothing	
cdot	⋅
times	×
del	∇
grad	∇
...	...
,, , ,	,, , ,
sum	Σ
int	∫
prod	Π
union	∪
inter	∩

To obtain Greek letters, simply spell them out in whatever case you want:

DELTA	Δ	iota	ι
GAMMA	Γ	kappa	κ

LAMBDA	Λ	lambda	λ
OMEGA	Ω	mu	μ
PHI	Φ	nu	ν
PI	Π	omega	ω
PSI	Ψ	omicron	o
SIGMA	Σ	phi	ϕ
THETA	Θ	pi	π
UPSILON	Υ	psi	ψ
XI	Ξ	rho	ρ
alpha	α	sigma	σ
beta	β	tau	τ
chi	χ	theta	θ
delta	δ	upsilon	υ
epsilon	ϵ	xi	ξ
eta	η	zeta	ζ
gamma	γ		

These are all the words known to EQN (except for characters with names), together with the section where they are discussed.

above	17, 18	lpile	17
back	21	mark	15
bar	13	matrix	18
bold	12	ndefine	20
ccol	18	over	9
col	18	pile	17
cpile	17	rcol	18
define	20	right	16
delim	19	roman	12
dot	13	rpile	17
dotdot	13	size	12
down	21	sqrt	10
dyad	13	sub	7
fat	12	sup	7
font	12	tdefine	20
from	11	tilde	13
fwd	21	to	11
gfont	12	under	13
gsize	12	up	21
hat	13	vec	13
italic	12	~, ^	4, 6
lcol	18	{ }	8
left	16	"..."	8, 14
lineup	15		

24. Troubleshooting

If you make a mistake in an equation, like leaving out a brace (very common) or having one too many (very common) or having a *sup* with nothing before it (common), EQN will tell you with the message

syntax error between lines x and y, file z

where *x* and *y* are approximately the lines between which the trouble occurred, and *z* is the name of the file in question. The line numbers are approximate — look nearby as well. There are also self-explanatory messages that arise if you leave out a quote or try to run EQN on a non-existent file.

If you want to check a document before actually printing it (on UNIX only),

```
eqn files >/dev/null
```

will throw away the output but print the messages.

If you use something like dollar signs as delimiters, it is easy to leave one out. This causes very strange troubles. The program *checkeq* (on GCOS, use *.checkeq* instead) checks for misplaced or missing dollar signs and similar troubles.

In-line equations can only be so big because of an internal buffer in TROFF. If you get a message "word overflow", you have exceeded this limit. If you print the equation as a displayed equation this message will usually go away. The message "line overflow" indicates you have exceeded an even bigger buffer. The only cure for this is to break the equation into two separate ones.

On a related topic, EQN does not break equations by itself — you must split long equations up across multiple lines by yourself, marking each by a separate *.EQEN* sequence. EQN does warn about equations that are too long to fit on one line.

25. Use on UNIX

To print a document that contains mathematics on the UNIX typesetter,

```
eqn files | troff
```

If there are any TROFF options, they go after the TROFF part of the command. For example,

```
eqn files | troff -ms (or -mm)
```

To run the same document on the GCOS typesetter, use

```
eqn files | troff -g (other options) | gcat
```


A compatible version of EQN can be used on devices like *TELETYPE* terminals and DASI and GSI terminals which have half-line forward and reverse capabilities. To print equations on a Model 37 *TELETYPE* terminal, for example, use

```
neqn files | nroff
```

The language for equations recognized by NEQN is identical to that of EQN, although of course the output is more restricted.

To use a GSI or DASI terminal as the output device,

```
neqn files | nroff -Tx
```

where *x* is the terminal type you are using, such as *300* or *300s*.

EQN and NEQN can be used with the TBL program [2] for setting tables that contain mathematics. Use TBL before [N]EQN, like this:

```
tbl files | eqn | troff
tbl files | neqn | nroff
```

26. Acknowledgments

We are deeply indebted to J. F. Ossanna, the author of TROFF, for his willingness to extend TROFF to make our task easier, and for his continuous assistance during the development and evolution of EQN. We are also grateful to A. V. Aho for advice on language design, to S. C. Johnson for assistance with the YACC compiler-compiler, and to all the EQN users who have made helpful suggestions and criticisms.

References

- [1] J. F. Ossanna. *NROFF/TROFF User's Manual*, Bell Laboratories, 1976.
- [2] M. E. Lesk. *Typing Documents on UNIX*, Bell Laboratories, 1976.
- [3] M. E. Lesk. *TBL—A Program for Setting Tables*, Bell Laboratories, 1976.
- [4] D. W. Smith and J. R. Mashey. *MM—Memorandum Macros*, Bell Laboratories, 1980.

A System for Typesetting Mathematics

Brian W. Kernighan

Lorinda L. Cherry

Bell Laboratories

Murray Hill, New Jersey 07974

ABSTRACT

This paper describes the design and implementation of a system for typesetting mathematics. The language has been designed to be easy to learn and to use by people (for example, secretaries and mathematical typists) who know neither mathematics nor typesetting. Experience indicates that the language can be learned in an hour or so, for it has few rules and fewer exceptions. For typical expressions, the size and font changes, positioning, line drawing, and the like necessary to print according to mathematical conventions are all done automatically. For example, the input

sum from $i=0$ to infinity x sub $i = \pi$ over 2

produces

$$\sum_{i=0}^{\infty} x_i = \frac{\pi}{2}$$

The syntax of the language is specified by a small context-free grammar; a compiler-compiler is used to make a compiler that translates this language into typesetting commands. Output may be produced on either a phototypesetter or on a terminal with forward and reverse half-line motions. The system interfaces directly with text formatting programs, so mixtures of text and mathematics may be handled simply.

This paper is a revision of a paper originally published in CACM, March, 1975.

1. Introduction

"Mathematics is known in the trade as *difficult*, or *penalty copy* because it is slower, more difficult, and more expensive to set in type than any other kind of copy normally occurring in books and journals." [1]

One difficulty with mathematical text is the multiplicity of characters, sizes, and fonts. An expression such as

$$\lim_{x \rightarrow \pi/2} (\tan x)^{\sin 2x} = 1$$

requires an intimate mixture of roman, italic and greek letters, in three sizes, and a special character or two. ("Requires" is perhaps the wrong word, but mathematics has its own typographical conventions which are quite different from those of ordinary text.) Typesetting such an expression by traditional methods is still an essentially manual operation.

A second difficulty is the two dimensional character of mathematics, which the superscript and limits in the preceding example showed in its simplest form. This is carried further by

$$a_0 + \frac{b_1}{a_1 + \frac{b_2}{a_2 + \frac{b_3}{a_3 + \dots}}}$$

and still further by

$$\int \frac{dx}{ae^{mx} - be^{-mx}} = \begin{cases} \frac{1}{2m\sqrt{ab}} \log \frac{\sqrt{a}e^{mx} - \sqrt{b}}{\sqrt{a}e^{mx} + \sqrt{b}} \\ \frac{1}{m\sqrt{ab}} \tanh^{-1} \left(\frac{\sqrt{a}}{\sqrt{b}} e^{mx} \right) \\ \frac{-1}{m\sqrt{ab}} \coth^{-1} \left(\frac{\sqrt{a}}{\sqrt{b}} e^{mx} \right) \end{cases}$$

These examples also show line-drawing, built-up characters like braces and radicals, and a spectrum of positioning problems. (Section 6 shows what a user has to type to produce these on our system.)

2. Photocomposition

Photocomposition techniques can be used to solve some of the problems of typesetting mathematics. A phototypesetter is a device which exposes a piece of photographic paper or film, placing characters wherever they are wanted. The Graphic Systems phototypesetter[2] on the UNIX† operating system[3] works by shining light through a character stencil. The character is made the right size by lenses, and the light beam directed by fiber optics to the desired place on a piece of photographic paper. The exposed paper is developed and typically used in some form of photo-offset reproduction.

On UNIX, the phototypesetter is driven by a formatting program called TROFF [4]. TROFF was designed for setting running text. It also provides all of the facilities that one needs for doing mathematics, such as arbitrary horizontal and vertical motions, line-drawing, size changing, but the syntax for describing these special operations is difficult to learn, and difficult even for experienced users to type correctly.

For this reason we decided to use TROFF as an "assembly language," by designing a language for describing mathematical expressions, and compiling it into TROFF.

3. Language Design

The fundamental principle upon which we based our language design is that the language should be easy to use by people (for example, secretaries) who know neither mathematics nor typesetting.

This principle implies several things. First, "normal" mathematical conventions about operator precedence, parentheses, and the like cannot be used, for to give special meaning to such characters means that the user has to understand what he or she is typing. Thus the language should not assume, for instance, that parentheses are always balanced, for they are not in the half-open interval $(a, b]$. Nor should it assume that $\sqrt{a+b}$ can be replaced by $(a+b)^{1/2}$, or that $1/(1-x)$ is better written as $\frac{1}{1-x}$ (or vice versa).

Second, there should be relatively few rules, keywords, special symbols and operators,

and the like. This keeps the language easy to learn and remember. Furthermore, there should be few exceptions to the rules that do exist: if something works in one situation, it should work everywhere. If a variable can have a subscript, then a subscript can have a subscript, and so on without limit.

Third, "standard" things should happen automatically. Someone who types " $x=y+z+1$ " should get " $x=y+z+1$ ". Subscripts and superscripts should automatically be printed in an appropriately smaller size, with no special intervention. Fraction bars have to be made the right length and positioned at the right height. And so on. Indeed a mechanism for overriding default actions has to exist, but its application is the exception, not the rule.

We assume that the typist has a reasonable picture (a two-dimensional representation) of the desired final form, as might be handwritten by the author of a paper. We also assume that the input is typed on a computer terminal much like an ordinary typewriter. This implies an input alphabet of perhaps 100 characters, none of them special.

A secondary, but still important, goal in our design was that the system should be easy to implement, since neither of the authors had any desire to make a long-term project of it. Since our design was not firm, it was also necessary that the program be easy to change at any time.

To make the program easy to build and to change, and to guarantee regularity ("it should work everywhere"), the language is defined by a context-free grammar, described in Section 5. The compiler for the language was built using a compiler-compiler.

A priori, the grammar/compiler-compiler approach seemed the right thing to do. Our subsequent experience leads us to believe that any other course would have been folly. The original language was designed in a few days. Construction of a working system sufficient to try significant examples required perhaps a person-month. Since then, we have spent a modest amount of additional time over several years tuning, adding facilities, and occasionally changing the language as users make criticisms and suggestions.

We also decided quite early that we would let TROFF do our work for us whenever possible. TROFF is quite a powerful program, with a macro facility, text and arithmetic variables, numerical computation and testing, and conditional branching. Thus we have been able to avoid writing a lot of mundane but tricky software. For example, we store no text strings, but simply pass

† UNIX is a trademark of Bell Laboratories.

them on to TROFF. Thus we avoid having to write a storage management package. Furthermore, we have been able to isolate ourselves from most details of the particular device and character set currently in use. For example, we let TROFF compute the widths of all strings of characters; we need know nothing about them.

A third design goal is special to our environment. Since our program is only useful for typesetting mathematics, it is necessary that it interface cleanly with the underlying typesetting language for the benefit of users who want to set intermingled mathematics and text (the usual case). The standard mode of operation is that when a document is typed, mathematical expressions are input as part of the text, but marked by user settable delimiters. The program reads this input and treats as comments those things which are not mathematics, simply passing them through untouched. At the same time it converts the mathematical input into the necessary TROFF commands. The resulting ioutput is passed directly to TROFF where the comments and the mathematical parts both become text and/or TROFF commands.

4. The Language

We will not try to describe the language precisely here; interested readers may refer to the appendix for more details. Throughout this section, we will write expressions exactly as they are handed to the typesetting program (hereinafter called "EQN"), except that we won't show the delimiters that the user types to mark the beginning and end of the expression. The interface between EQN and TROFF is described at the end of this section.

As we said, typing $x=y+z+1$ should produce $x=y+z+1$, and indeed it does. Variables are made italic, operators and digits become roman, and normal spacings between letters and operators are altered slightly to give a more pleasing appearance.

Input is free-form. Spaces and new lines in the input are used by EQN to separate pieces of the input; they are not used to create space in the output. Thus

$$\begin{array}{l} x \\ = \\ y \\ + z + 1 \end{array}$$

also gives $x=y+z+1$. Free-form input is easier to type initially; subsequent editing is also easier, for an expression may be typed as many short lines.

Extra white space can be forced into the output by several characters of various sizes. A tilde "~" gives a space equal to the normal word spacing in text; a circumflex gives half this

much, and a tab character spaces to the next tab stop.

Spaces (or tildes, etc.) also serve to delimit pieces of the input. For example, to get

$$f(t)=2\pi \int \sin(\omega t) dt$$

we write

$$f(t) = 2 \text{ pi int sin (omega t) dt}$$

Here spaces are *necessary* in the input to indicate that *sin*, *pi*, *int*, and *omega* are special, and potentially worth special treatment. EQN looks up each such string of characters in a table, and if appropriate gives it a translation. In this case, *pi* and *omega* become their greek equivalents, *int* becomes the integral sign (which must be moved down and enlarged so it looks "right"), and *sin* is made roman, following conventional mathematical practice. Parentheses, digits and operators are automatically made roman wherever found.

Fractions are specified with the keyword *over*:

$$a+b \text{ over } c+d+e = 1$$

produces

$$\frac{a+b}{c+d+e}=1$$

Similarly, subscripts and superscripts are introduced by the keywords *sub* and *sup*:

$$x^2+y^2=z^2$$

is produced by

$$x \text{ sup } 2 + y \text{ sup } 2 = z \text{ sup } 2$$

The spaces after the 2's are necessary to mark the end of the superscripts; similarly the keyword *sup* has to be marked off by spaces or some equivalent delimiter. The return to the proper baseline is automatic. Multiple levels of subscripts or superscripts are of course allowed: "x sup y sup z" is x^{y^z} . The construct "something *sub* something *sup* something" is recognized as a special case, so "x sub i sup 2" is x_i^2 instead of x_i^2 .

More complicated expressions can now be formed with these primitives:

$$\frac{\partial^2 f}{\partial x^2} = \frac{x^2}{a^2} + \frac{y^2}{b^2}$$

is produced by

$$\{\text{partial sup } 2 \text{ f}\} \text{ over } \{\text{partial } x \text{ sup } 2\} = \\ x \text{ sup } 2 \text{ over } a \text{ sup } 2 + y \text{ sup } 2 \text{ over } b \text{ sup } 2$$

Braces {} are used to group objects together; in this case they indicate unambiguously what goes over what on the left-hand side of the

expression. The language defines the precedence of *sup* to be higher than that of *over*, so no braces are needed to get the correct association on the right side. Braces can always be used when in doubt about precedence.

The braces convention is an example of the power of using a recursive grammar to define the language. It is part of the language that if a construct can appear in some context, then *any expression* in braces can also occur in that context.

There is a *sqr*t operator for making square roots of the appropriate size: "sqr a+b" produces $\sqrt{a+b}$, and

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

is

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Since large radicals look poor on our typesetter, *sqr*t is not useful for tall expressions.

Limits on summations, integrals and similar constructions are specified with the keywords *from* and *to*. To get

$$\sum_{i=0}^{\infty} x_i \rightarrow 0$$

we need only type

$$\text{sum from } i=0 \text{ to } \text{inf } x \text{ sub } i \rightarrow 0$$

Centering and making the Σ big enough and the limits smaller are all automatic. The *from* and *to* parts are both optional, and the central part (e.g., the Σ) can in fact be anything:

$$\lim_{x \rightarrow \pi/2} (\tan x) = \text{inf}$$

is

$$\lim_{x \rightarrow \pi/2} (\tan x) = \text{inf}$$

Again, the braces indicate just what goes into the *from* part.

There is a facility for making braces, brackets, parentheses, and vertical bars of the right height, using the keywords *left* and *right*:

$$\text{left } [x+y \text{ over } 2a \text{ right }] = 1$$

makes

$$\left[\frac{x+y}{2a} \right] = 1$$

A *left* need not have a corresponding *right*, as we shall see in the next example. Any characters may follow *left* and *right*, but generally only various parentheses and bars are meaningful.

Big brackets, etc., are often used with another facility, called *piles*, which make vertical

piles of objects. For example, to get

$$\text{sign}(x) \equiv \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x = 0 \\ -1 & \text{if } x < 0 \end{cases}$$

we can type

$$\text{sign}(x) \text{ ~} \text{~} \text{~} \text{left } \{ \text{rpile } \{ 1 \text{ above } 0 \text{ above } -1 \} \text{~} \text{~} \text{~} \text{ipile } \{ \text{if above if above if } \} \text{~} \text{~} \text{~} \text{ipile } \{ x > 0 \text{ above } x = 0 \text{ above } x < 0 \} \}$$

The construction "left {" makes a left brace big enough to enclose the "rpile {...}", which is a right-justified pile of "above ... above ...". "ipile" makes a left-justified pile. There are also centered piles. Because of the recursive language definition, a pile can contain any number of elements; any element of a pile can of course contain piles.

Although EQN makes a valiant attempt to use the right sizes and fonts, there are times when the default assumptions are simply not what is wanted. For instance the italic *sign* in the previous example would conventionally be in roman. Slides and transparencies often require larger characters than normal text. Thus we also provide size and font changing commands: "size 12 bold {A x = y}" will produce **A x = y**. *Size* is followed by a number representing a character size in points. (One point is 1/72 inch; this paper is set in 9 point type.)

If necessary, an input string can be quoted in "...", which turns off grammatical significance, and any font or spacing changes that might otherwise be done on it. Thus we can say

$$\lim_{\text{roman}} \text{"sup"} x \text{ sub } n = 0$$

to ensure that the supremum doesn't become a superscript:

$$\lim \sup x_n = 0$$

Diacritical marks, long a problem in traditional typesetting, are straightforward:

$$\underline{\dot{x}} + \hat{x} + \tilde{y} + \hat{X} + \ddot{Y} = \bar{z} + \bar{Z}$$

is made by typing

$$x \text{ dot under } + x \text{ hat } + y \text{ tilde } + X \text{ hat } + Y \text{ dotdot } = z + Z \text{ bar}$$

There are also facilities for globally changing default sizes and fonts, for example for making viewgraphs or for setting chemical equations. The language allows for matrices, and for lining up equations at the same horizontal position.

Finally, there is a definition facility, so a user can say

define name "..."

at any time in the document; henceforth, any occurrence of the token "name" in an expression will be expanded into whatever was inside the double quotes in its definition. This lets users tailor the language to their own specifications, for it is quite possible to redefine keywords like *sup* or *over*. Section 6 shows an example of definitions.

The EQN preprocessor reads intermixed text and equations, and passes its output to TROFF. Since TROFF uses lines beginning with a period as control words (e.g., ".ce" means "center the next output line"), EQN uses the sequence ".EQ" to mark the beginning of an equation and ".EN" to mark the end. The ".EQ" and ".EN" are passed through to TROFF untouched, so they can also be used by a knowledgeable user to center equations, number them automatically, etc. By default, however, ".EQ" and ".EN" are simply ignored by TROFF, so by default equations are printed in-line.

".EQ" and ".EN" can be supplemented by TROFF commands as desired; for example, a centered display equation can be produced with the input:

```
.ce
.EQ
x sub i = y sub i ...
.EN
```

Since it is tedious to type ".EQ" and ".EN" around very short expressions (single letters, for instance), the user can also define two characters to serve as the left and right delimiters of expressions. These characters are recognized anywhere in subsequent text. For example if the left and right delimiters have both been set to "#", the input:

Let #x sub i#, #y# and #alpha# be positive produces:

Let x_i , y and α be positive

Running a preprocessor is strikingly easy on UNIX. To typeset text stored in file "f", one issues the command:

```
eqn f | troff
```

The vertical bar connects the output of one process (EQN) to the input of another (TROFF).

5. Language Theory

The basic structure of the language is not a particularly original one. Equations are pictured as a set of "boxes," pieced together in various ways. For example, something with a subscript

is just a box followed by another box moved downward and shrunk by an appropriate amount. A fraction is just a box centered above another box, at the right altitude, with a line of correct length drawn between them.

The grammar for the language is shown below. For purposes of exposition, we have collapsed some productions. In the original grammar, there are about 70 productions, but many of these are simple ones used only to guarantee that some keyword is recognized early enough in the parsing process. Symbols in capital letters are terminal symbols; lower case symbols are non-terminals, i.e., syntactic categories. The vertical bar | indicates an alternative; the brackets [] indicate optional material. A TEXT is a string of non-blank characters or any string inside double quotes; the other terminal symbols represent literal occurrences of the corresponding keyword.

```
eqn : box | eqn box
box : text
    | { eqn }
    | box OVER box
    | SQRT box
    | box SUB box | box SUP box
    | [ L | C | R ] PILE { list }
    | LEFT text eqn [ RIGHT text ]
    | box [ FROM box ] [ TO box ]
    | SIZE text box
    | [ ROMAN | BOLD | ITALIC ] box
    | box [ HAT | BAR | DOT | DOTDOT | TILDE ]
    | DEFINE text text
list : eqn | list ABOVE eqn
text : TEXT
```

The grammar makes it obvious why there are few exceptions. For example, the observation that something can be replaced by a more complicated something in braces is implicit in the productions:

```
eqn : box | eqn box
box : text | { eqn }
```

Anywhere a single character could be used, *any* legal construction can be used.

Clearly, our grammar is highly ambiguous. What, for instance, do we do with the input

a over b over c ?

Is it

{a over b} over c

or is it

a over {b over c} ?

To answer questions like this, the grammar is supplemented with a small set of rules that describe the precedence and associativity of operators. In particular, we specify (more or less arbitrarily) that *over* associates to the left, so the first alternative above is the one chosen. On the other hand, *sub* and *sup* bind to the right, because this is closer to standard mathematical practice. That is, we assume x^{a^b} is $x^{(a^b)}$, not $(x^a)^b$.

The precedence rules resolve the ambiguity in a construction like

a sup 2 over b

We define *sup* to have a higher precedence than *over*, so this construction is parsed as $\frac{a^2}{b}$ instead of $a^{\frac{2}{b}}$.

Naturally, a user can always force a particular parsing by placing braces around expressions.

The ambiguous grammar approach seems to be quite useful. The grammar we use is small enough to be easily understood, for it contains none of the productions that would be normally used for resolving ambiguity. Instead the supplemental information about precedence and associativity (also small enough to be understood) provides the compiler-compiler with the information it needs to make a fast, deterministic parser for the specific language we want. When the language is supplemented by the disambiguating rules, it is in fact LR(1) and thus easy to parse[5].

The output code is generated as the input is scanned. Any time a production of the grammar is recognized, (potentially) some TROFF commands are output. For example, when the lexical analyzer reports that it has found a TEXT (i.e., a string of contiguous characters), we have recognized the production:

text : TEXT

The translation of this is simple. We generate a local name for the string, then hand the name and the string to TROFF, and let TROFF perform the storage management. All we save is the name of the string, its height, and its baseline.

As another example, the translation associated with the production

box : box OVER box

is:

Width of output box =
 slightly more than largest input width
 Height of output box =
 slightly more than sum of input heights
 Base of output box =
 slightly more than height of bottom input box
 String describing output box =
 move down;
 move right enough to center bottom box;
 draw bottom box (i.e., copy string for bottom box);
 move up; move left enough to center top box;
 draw top box (i.e., copy string for top box);
 move down and left; draw line full width;
 return to proper base line.

Most of the other productions have equally simple semantic actions. Picturing the output as a set of properly placed boxes makes the right sequence of positioning commands quite obvious. The main difficulty is in finding the right numbers to use for esthetically pleasing positioning.

With a grammar, it is usually clear how to extend the language. For instance, one of our users suggested a TENSOR operator, to make constructions like

$$\begin{matrix} l & k & j \\ m & \mathbf{T} & \\ & n & i \end{matrix}$$

Grammatically, this is easy: it is sufficient to add a production like

box : TENSOR { list }

Semantically, we need only juggle the boxes to the right places.

6. Experience

There are really three aspects of interest—how well EQN sets mathematics, how well it satisfies its goal of being “easy to use,” and how easy it was to build.

The first question is easily addressed. This entire paper has been set by the program. Readers can judge for themselves whether it is good enough for their purposes. One of our users commented that although the output is not as good as the best hand-set material, it is still better than average, and much better than the worst. In any case, who cares? Printed books cannot compete with the birds and flowers of illuminated manuscripts on esthetic grounds, either, but they have some clear economic advantages.

Some of the deficiencies in the output could be cleaned up with more work on our part. For example, we sometimes leave too much space between a roman letter and an italic one. If we were willing to keep track of the fonts

involved, we could do this better more of the time.

Some other weaknesses are inherent in our output device. It is hard, for instance, to draw a line of an arbitrary length without getting a perceptible overstrike at one end.

As to ease of use, at the time of writing, the system has been used by two distinct groups. One user population consists of mathematicians, chemists, physicists, and computer scientists. Their typical reaction has been something like:

- (1) It's easy to write, although I make the following mistakes...
- (2) How do I do...?
- (3) It botches the following things... Why don't you fix them?
- (4) You really need the following features...

The learning time is short. A few minutes gives the general flavor, and typing a page or two of a paper generally uncovers most of the misconceptions about how it works.

The second user group is much larger, the secretaries and mathematical typists who were the original target of the system. They tend to be enthusiastic converts. They find the language easy to learn (most are largely self-taught), and have little trouble producing the output they want. They are of course less critical of the esthetics of their output than users trained in mathematics. After a transition period, most find using a computer more interesting than a regular typewriter.

The main difficulty that users have seems to be remembering that a blank is a delimiter; even experienced users use blanks where they shouldn't and omit them when they are needed. A common instance is typing

$$f(x \text{ sub } i)$$

which produces

$$f(x_i)$$

instead of

$$f(x_i)$$

Since the EQN language knows no mathematics, it cannot deduce that the right parenthesis is not part of the subscript.

The language is somewhat prolix, but this doesn't seem excessive considering how much is being done, and it is certainly more compact than the corresponding TROFF commands. For example, here is the source for the continued fraction expression in Section 1 of this paper:

$$a \text{ sub } 0 + b \text{ sub } 1 \text{ over } \left\{ a \text{ sub } 1 + b \text{ sub } 2 \text{ over } \left\{ a \text{ sub } 2 + b \text{ sub } 3 \text{ over } \left\{ a \text{ sub } 3 + \dots \right\} \right\} \right\}$$

This is the input for the large integral of Section 1; notice the use of definitions:

```
define emx "{e sup mx}"
define mab "{m sqrt ab}"
define sa "{sqrt a}"
define sb "{sqrt b}"
int dx over {a emx - be sup -mx} ^=-
left { lpile {
  1 over {2 mab} ^log^
  {sa emx - sb} over {sa emx + sb}
  above
  1 over mab ^ tanh sup -1 ( sa over sb emx )
  above
  -1 over mab ^ coth sup -1 ( sa over sb emx )
}
```

As to ease of construction, we have already mentioned that there are really only a few person-months invested. Much of this time has gone into two things—fine-tuning (what is the most esthetically pleasing space to use between the numerator and denominator of a fraction?), and changing things found deficient by our users (shouldn't a tilde be a delimiter?).

The program consists of a number of small, essentially unconnected modules for code generation, a simple lexical analyzer, a canned parser which we did not have to write, and some miscellany associated with input files and the macro facility. The program is now about 1600 lines of C [6], a high-level language reminiscent of BCPL. About 20 percent of these lines are "print" statements, generating the output code.

The semantic routines that generate the actual TROFF commands can be changed to accommodate other formatting languages and devices. For example, in less than 24 hours, one of us changed the entire semantic package to drive NROFF, a variant of TROFF, for typesetting mathematics on teletypewriter devices capable of reverse line motions. Since many potential users do not have access to a typesetter, but still have to type mathematics, this provides a way to get a typed version of the final output which is close enough for debugging purposes, and sometimes even for ultimate use.

7. Conclusions

We think we have shown that it is possible to do acceptably good typesetting of mathematics on a phototypesetter, with an input language that is easy to learn and use and that satisfies many users' demands. Such a package can be

implemented in short order, given a compiler-compiler and a decent typesetting program underneath.

Defining a language, and building a compiler for it with a compiler-compiler seems like the only sensible way to do business. Our experience with the use of a grammar and a compiler-compiler has been uniformly favorable. If we had written everything into code directly, we would have been locked into our original design. Furthermore, we would have never been sure where the exceptions and special cases were. But because we have a grammar, we can change our minds readily and still be reasonably sure that if a construction works in one place it will work everywhere.

Acknowledgements

We are deeply indebted to J. F. Ossanna, the author of TROFF, for his willingness to modify TROFF to make our task easier and for his continuous assistance during the development of our program. We are also grateful to A. V. Aho for help with language theory, to S. C. Johnson for aid with the compiler-compiler, and to our early users A. V. Aho, S. I. Feldman, S. C. Johnson, R. W. Hamming, and M. D. McIlroy for their constructive criticisms.

References

- [1] *A Manual of Style*, 12th Edition, University of Chicago Press, 1969, p. 295.
- [2] *Model CIA/T Phototypesetter*, Graphic Systems, Inc., Hudson, NH.
- [3] Ritchie, D. M., and Thompson, K. The UNIX Time-Sharing System, *CACM* 17(7):365-75 (July 1974).
- [4] Ossanna, J. F. *NROFF/TROFF User's Manual*, Bell Laboratories, 1977.
- [5] Aho, A. V., and Johnson, S. C. LR Parsing, *Comp. Surv.* 6(2):99-124 (June 1974).
- [6] Kernighan, B. W., and Ritchie, D. M. *The C Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1978.

January 1981

The C Programming Language—Reference Manual*

Dennis M. Ritchie
Bell Laboratories
Murray Hill, New Jersey 07974

1. INTRODUCTION

This manual describes the C language on the DEC PDP-11, the DEC VAX-11/780, the Honeywell 6000, the IBM System/370, and the Interdata 8/32. Where differences exist, it concentrates on the PDP-11, but tries to point out implementation-dependent details. With few exceptions, such dependencies follow directly from the properties of the hardware; the various compilers are generally quite compatible.

2. LEXICAL CONVENTIONS

There are six classes of tokens: identifiers, keywords, constants, strings, operators, and other separators. Blanks, tabs, new-lines, and comments (collectively, "white space") as described below are ignored except as they serve to separate tokens. Some white space is required to separate otherwise adjacent identifiers, keywords, and constants.

If the input stream has been parsed into tokens up to a given character, the next token is taken to include the longest string of characters which could possibly constitute a token.

2.1 Comments

The characters `/*` introduce a comment, which terminates with the characters `*/`. Comments do not nest.

2.2 Identifiers (Names)

An identifier is a sequence of letters and digits; the first character must be a letter; the underscore `_` counts as a letter. Upper- and lower-case letters are different. No more than the first eight characters are significant, although more may be used. External identifiers, which are used by various assemblers and loaders, are more restricted:

DEC PDP-11	7 characters, 2 cases
DEC VAX-11	7 characters, 2 cases
Honeywell 6000	6 characters, 1 case
IBM 360/370	7 characters, 1 case
Interdata 8/32	8 characters, 2 cases

2.3 Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise:

<code>auto</code>	<code>do</code>	<code>float</code>	<code>register</code>	<code>switch</code>
<code>break</code>	<code>double</code>	<code>for</code>	<code>return</code>	<code>typedef</code>
<code>case</code>	<code>else</code>	<code>goto</code>	<code>short</code>	<code>union</code>
<code>char</code>	<code>entry</code>	<code>if</code>	<code>sizeof</code>	<code>unsigned</code>
<code>continue</code>	<code>enum</code>	<code>int</code>	<code>static</code>	<code>void</code>
<code>default</code>	<code>extern</code>	<code>long</code>	<code>struct</code>	<code>while</code>

The `entry` keyword is not currently implemented by any compiler but is reserved for future use. Some implementations also reserve the words `fortran` and `asm`.

* This manual is reprinted, with minor changes, from *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie, Prentice Hall, Inc., 1978. It specifies the language definition as of September, 1980.

2.4 Constants

There are several kinds of constants, as listed below. Hardware characteristics that affect sizes are summarized in §2.6.

2.4.1 Integer constants

An integer constant consisting of a sequence of digits is taken to be octal if it begins with 0 (digit zero), decimal otherwise. A sequence of digits preceded by 0x or 0X (digit zero) is taken to be a hexadecimal integer. The hexadecimal digits include a or A through f or F with values 10 through 15. A decimal constant whose value exceeds the largest signed machine integer is taken to be long; an octal or hex constant which exceeds the largest unsigned machine integer is likewise taken to be long.

2.4.2 Explicit long constants

A decimal, octal, or hexadecimal integer constant immediately followed by l (letter ell) or L is a long constant. As discussed below, on some machines integer and long values may be considered identical.

2.4.3 Character constants

A character constant is a character enclosed in single quotes, as in 'x'. The value of a character constant is the numerical value of the character in the machine's character set.

Certain non-graphic characters, the single quote ' and the backslash \, may be represented according to the following table of escape sequences:

new-line	NL (LF)	\n
horizontal tab	HT	\t
vertical tab	VT	\v
backspace	BS	\b
carriage return	CR	\r
form feed	FF	\f
backslash	\	\\
single quote	'	\'
bit pattern	ddd	\ddd

The escape \ddd consists of the backslash followed by 1, 2, or 3 octal digits which are taken to specify the value of the desired character. A special case of this construction is \0 (not followed by a digit), which indicates the character NUL. If the character following a backslash is not one of those specified, the backslash is ignored.

2.4.4 Floating constants

A floating constant consists of an integer part, a decimal point, a fraction part, an e or E, and an optionally signed integer exponent. The integer and fraction parts both consist of a sequence of digits. Either the integer part or the fraction part (not both) may be missing; either the decimal point or the e and the exponent (not both) may be missing. Every floating constant is taken to be double-precision.

2.4.5 Enumeration constants

Names declared as enumerators (see §8.5) are constants of the corresponding enumeration type. They behave like int constants.

2.5 Strings

A string is a sequence of characters surrounded by double quotes, as in "...". A string has type "array of characters" and storage class static (see §4 below) and is initialized with the given characters. All strings, even when written identically, are distinct. The compiler places a null byte \0 at the end of each string so that programs which scan the string can find its end. In a string, the double quote character " must be preceded by a \; in addition, the same escapes as described for character constants may be used. Finally, a \ and the immediately following new-line are ignored.

2.6 Hardware characteristics

The following table summarizes certain hardware properties that vary from machine to machine.

	DEC PDP-11	DEC VAX-11	Honeywell 6000	IBM 370	Interdata 8/32
	ASCII	ASCII	ASCII	EBCDIC	ASCII
char	8 bits	8 bits	9 bits	8 bits	8 bits
int	16	32	36	32	32
short	16	16	36	16	16
long	32	32	36	32	32
float	32	32	36	32	32
double	64	64	72	64	64
range	$\pm 10^{\pm 38}$	$\pm 10^{\pm 38}$	$\pm 10^{\pm 38}$	$\pm 10^{\pm 76}$	$\pm 10^{\pm 76}$

3. SYNTAX NOTATION

In the syntax notation used in this manual, syntactic categories are indicated by *italic* type, and literal words and characters in *constant-width* type. Alternative categories are listed on separate lines. An optional terminal or non-terminal symbol is indicated by the subscript "opt," so that

{ *expression*_{opt} }

indicates an optional expression enclosed in braces. The syntax is summarized in §18.

4. WHAT'S IN A NAME?

C bases the interpretation of an identifier upon two attributes of the identifier: its *storage class* and its *type*. The storage class determines the location and lifetime of the storage associated with an identifier; the type determines the meaning of the values found in the identifier's storage.

There are four declarable storage classes: automatic, static, external, and register. Automatic variables are local to each invocation of a block (§9.2), and are discarded upon exit from the block; static variables are local to a block, but retain their values upon reentry to a block even after control has left the block; external variables exist and retain their values throughout the execution of the entire program, and may be used for communication between functions, even separately compiled functions. Register variables are (if possible) stored in the fast registers of the machine; like automatic variables they are local to each block and disappear on exit from the block.

C supports several fundamental types of objects:

Objects declared as characters (**char**) are large enough to store any member of the implementation's character set, and if a genuine character from that character set is stored in a character variable, its value is equivalent to the integer code for that character. Other quantities may be stored into character variables, but the implementation is machine-dependent.

Up to three sizes of integer, declared **short int**, **int**, and **long int**, are available. Longer integers provide no less storage than shorter ones, but the implementation may make either short integers, or long integers, or both, equivalent to plain integers. "Plain" integers have the natural size suggested by the host machine architecture; the other sizes are provided to meet special needs.

Each enumeration (§8.5) is conceptually a separate type with its own set of named constants. The properties of an **enum** type are identical to those of **int** type.

Unsigned integers, declared **unsigned**, obey the laws of arithmetic modulo 2^n where n is the number of bits in the representation. (On the PDP-11, unsigned long quantities are not supported.)

Single-precision floating point (**float**) and double-precision floating point (**double**) may be synonymous in some implementations.

Because objects of the foregoing types can usefully be interpreted as numbers, they will be referred to as *arithmetic* types. Types **char**, **int** of all sizes, and **enum** will collectively be called *integral* types. **float** and **double** will collectively be called *floating* types.

The **void** type specifies an empty set of values. It is used as the type returned by functions that generate no value.

Besides the fundamental arithmetic types there is a conceptually infinite class of derived types constructed from the fundamental types in the following ways:

arrays of objects of most types;
functions which return objects of a given type;
pointers to objects of a given type;
structures containing a sequence of objects of various types;
unions capable of containing any one of several objects of various types.

In general these methods of constructing objects can be applied recursively.

5. OBJECTS AND LVALUES

An *object* is a manipulatable region of storage; an *lvalue* is an expression referring to an object. An obvious example of an lvalue expression is an identifier. There are operators which yield lvalues: for example, if *E* is an expression of pointer type, then **E* is an lvalue expression referring to the object to which *E* points. The name "lvalue" comes from the assignment expression *E1 = E2* in which the left operand *E1* must be an lvalue expression. The discussion of each operator below indicates whether it expects lvalue operands and whether it yields an lvalue.

6. CONVERSIONS

A number of operators may, depending on their operands, cause conversion of the value of an operand from one type to another. This section explains the result to be expected from such conversions. §6.6 summarizes the conversions demanded by most ordinary operators; it will be supplemented as required by the discussion of each operator.

6.1 Characters and integers

A character or a short integer may be used wherever an integer may be used. In all cases the value is converted to an integer. Conversion of a shorter integer to a longer always involves sign extension; integers are signed quantities. Whether or not sign-extension occurs for characters is machine dependent, but it is guaranteed that a member of the standard character set is non-negative. Of the machines treated here, only the PDP-11 and VAX-11 sign-extend. On these machines, `char` variables range in value from -128 to 127. The more explicit type `unsigned char` forces the values to range from 0 to 255.

On machines that treat characters as signed, the characters of the ASCII set are all positive. However, a character constant specified with an octal escape suffers sign extension and may appear negative; for example, `'\377'` has the value -1.

When a longer integer is converted to a shorter or to a `char`, it is truncated on the left; excess bits are simply discarded.

6.2 Float and double

All floating arithmetic in C is carried out in double-precision; whenever a `float` appears in an expression it is lengthened to `double` by zero-padding its fraction. When a `double` must be converted to `float`, for example by an assignment, the `double` is rounded before truncation to `float` length.

6.3 Floating and integral

Conversions of floating values to integral type tend to be rather machine-dependent; in particular the direction of truncation of negative numbers varies from machine to machine. The result is undefined if the value will not fit in the space provided.

Conversions of integral values to floating type are well behaved. Some loss of precision occurs if the destination lacks sufficient bits.

6.4 Pointers and integers

An expression of integral type may be added to or subtracted from a pointer; in such a case the first is converted as specified in the discussion of the addition operator.

Two pointers to objects of the same type may be subtracted; in this case the result is converted to an integer as specified in the discussion of the subtraction operator.

6.5 Unsigned

Whenever an unsigned integer and a plain integer are combined, the plain integer is converted to unsigned and the result is unsigned. The value is the least unsigned integer congruent to the signed integer (modulo 2^{wordsize}). In a 2's complement representation, this conversion is conceptual and there is no actual change in the bit pattern.

When an unsigned integer is converted to `long`, the value of the result is the same numerically as that of the unsigned integer. Thus the conversion amounts to padding with zeros on the left.

6.6 Arithmetic conversions

A great many operators cause conversions and yield result types in a similar way. This pattern will be called the “usual arithmetic conversions.”

First, any operands of type `char` or `short` are converted to `int`, and any of type `float` are converted to `double`.

Then, if either operand is `double`, the other is converted to `double` and that is the type of the result.

Otherwise, if either operand is `long`, the other is converted to `long` and that is the type of the result.

Otherwise, if either operand is `unsigned`, the other is converted to `unsigned` and that is the type of the result.

Otherwise, both operands must be `int`, and that is the type of the result.

6.7 Void

The (nonexistent) value of a `void` object may not be used in any way, and neither explicit nor implicit conversion may be applied. Because a `void` expression denotes a nonexistent value, such an expression may be used only as an expression statement (§9.1) or as the left operand of a comma expression (§7.15).

An expression may be converted to type `void` by use of a cast. For example, this makes explicit the discarding of the value of a function call used as an expression statement.

7. EXPRESSIONS

The precedence of expression operators is the same as the order of the major subsections of this section, highest precedence first. Thus, for example, the expressions referred to as the operands of `+` (§7.4) are those expressions defined in §§7.1-7.3. Within each subsection, the operators have the same precedence. Left- or right-associativity is specified in each subsection for the operators discussed therein. The precedence and associativity of all the expression operators is summarized in the grammar of §18.

Otherwise the order of evaluation of expressions is undefined. In particular the compiler considers itself free to compute subexpressions in the order it believes most efficient, even if the subexpressions involve side effects. The order in which side effects take place is unspecified. Expressions involving a commutative and associative operator (`*`, `+`, `&`, `!`, `^`) may be rearranged arbitrarily, even in the presence of parentheses; to force a particular order of evaluation an explicit temporary must be used.

The handling of overflow and divide check in expression evaluation is machine-dependent. Most existing implementations of C ignore integer overflows; treatment of division by 0, and all floating-point exceptions, varies between machines, and is usually adjustable by a library function.

7.1 Primary expressions

Primary expressions involving `..`, `->`, subscripting, and function calls group left to right.

primary-expression:

identifier

constant

string

(expression)

primary-expression [expression]

primary-expression (expression-list_{opt})

primary-expression . identifier

primary-expression -> identifier

expression-list:

expression

expression-list , expression

An identifier is a primary expression, provided it has been suitably declared as discussed below. Its type is specified by its declaration. If the type of the identifier is “array of ...”, however, then the value of the identifier-expression is a pointer to the first object in the array, and the type of the expression is “pointer to ...”. Moreover, an array identifier is not an lvalue expression. Likewise, an identifier which is declared “function returning ...”, when used except in the function-name position of a call, is converted to “pointer to function returning ...”.

A constant is a primary expression. Its type may be `int`, `long`, or `double` depending on its form. Character constants have type `int`; floating constants are `double`.

A string is a primary expression. Its type is originally “array of `char`”; but following the same rule given above for identifiers, this is modified to “pointer to `char`” and the result is a pointer to the first character in the string. (There is an exception in certain initializers; see §8.6.)

A parenthesized expression is a primary expression whose type and value are identical to those of the unadorned expression. The presence of parentheses does not affect whether the expression is an lvalue.

A primary expression followed by an expression in square brackets is a primary expression. The intuitive meaning is that of a subscript. Usually, the primary expression has type “pointer to ...”, the subscript expression is `int`, and the type of the result is “...”. The expression `E1[E2]` is identical (by definition) to `*((E1)+(E2))`. All the clues needed to understand this notation are contained in this section together with the discussions in §§ 7.1, 7.2, and 7.4 on identifiers, `*`, and `+` respectively; §14.3 below summarizes the implications.

A function call is a primary expression followed by parentheses containing a possibly empty, comma-separated list of expressions which constitute the actual arguments to the function. The primary expression must be of type “function returning ...”, and the result of the function call is of type “...”. As indicated below, a hitherto unseen identifier followed immediately by a left parenthesis is contextually declared to represent a function returning an integer; thus in the most common case, integer-valued functions need not be declared.

Any actual arguments of type `float` are converted to `double` before the call; any of type `char` or `short` are converted to `int`; and as usual, array names are converted to pointers. No other conversions are performed automatically; in particular, the compiler does not compare the types of actual arguments with those of formal arguments. If conversion is needed, use a cast; see §7.2, 8.7.

In preparing for the call to a function, a copy is made of each actual parameter; thus, all argument-passing in C is strictly by value. A function may change the values of its formal parameters, but these changes cannot affect the values of the actual parameters. On the other hand, it is possible to pass a pointer on the understanding that the function may change the value of the object to which the pointer points. An array name is a pointer expression. The order of evaluation of arguments is undefined by the language; take note that the various compilers differ.

Recursive calls to any function are permitted.

A primary expression followed by a dot followed by an identifier is an expression. The first expression must be a structure or a union, and the identifier must name a member of the structure or union. The value is the named member of the structure or union, and it is an lvalue if the first expression is an lvalue.

A primary expression followed by an arrow (built from a `-` and a `>`) followed by an identifier is an expression. The first expression must be a pointer to a structure or a union and the identifier must name a member of that structure or union. The result is an lvalue referring to the named member of the structure or union to which the pointer expression points. Thus the expression `E1->MOS` is the same as `(*E1).MOS`. Structures and unions are discussed in §8.5.

7.2 Unary operators

Expressions with unary operators group right-to-left.

```
unary-expression:
    * expression
    & lvalue
    - expression
    ! expression
    ~ expression
    ++ lvalue
    -- lvalue
    lvalue ++
    lvalue --
    ( type-name ) expression
    sizeof expression
    sizeof ( type-name )
```

The unary `*` operator means *indirection*: the expression must be a pointer, and the result is an lvalue referring to the object to which the expression points. If the type of the expression is “pointer to ...”, the type of the result is “...”.

The result of the unary & operator is a pointer to the object referred to by the lvalue. If the type of the lvalue is "...", the type of the result is "pointer to ...".

The result of the unary - operator is the negative of its operand. The usual arithmetic conversions are performed. The negative of an unsigned quantity is computed by subtracting its value from 2^n , where n is the number of bits in an int. There is no unary + operator.

The result of the logical negation operator ! is 1 if the value of its operand is 0, 0 if the value of its operand is non-zero. The type of the result is int. It is applicable to any arithmetic type or to pointers.

The ~ operator yields the one's complement of its operand. The usual arithmetic conversions are performed. The type of the operand must be integral.

The object referred to by the lvalue operand of prefix ++ is incremented. The value is the new value of the operand, but is not an lvalue. The expression ++x is equivalent to x+=1. See the discussions of addition (§7.4) and assignment operators (§7.14) for information on conversions.

The lvalue operand of prefix -- is decremented analogously to the prefix ++ operator.

When postfix ++ is applied to an lvalue the result is the value of the object referred to by the lvalue. After the result is noted, the object is incremented in the same manner as for the prefix ++ operator. The type of the result is the same as the type of the lvalue expression.

When postfix -- is applied to an lvalue the result is the value of the object referred to by the lvalue. After the result is noted, the object is decremented in the manner as for the prefix -- operator. The type of the result is the same as the type of the lvalue expression.

An expression preceded by the parenthesized name of a data type causes conversion of the value of the expression to the named type. This construction is called a *cast*. Type names are described in §8.7.

The sizeof operator yields the size, in bytes, of its operand. (A *byte* is undefined by the language except in terms of the value of sizeof. However, in all existing implementations a byte is the space required to hold a char.) When applied to an array, the result is the total number of bytes in the array. The size is determined from the declarations of the objects in the expression. This expression is semantically an unsigned constant* and may be used anywhere a constant is required. Its major use is in communication with routines like storage allocators and I/O systems.

The sizeof operator may also be applied to a parenthesized type name. In that case it yields the size, in bytes, of an object of the indicated type.

The construction sizeof(*type*) is taken to be a unit, so the expression sizeof(*type*)-2 is the same as (sizeof(*type*))-2.

7.3 Multiplicative operators

The multiplicative operators *, /, and % group left-to-right. The usual arithmetic conversions are performed.

multiplicative-expression:

*expression * expression*

expression / expression

expression % expression

The binary * operator indicates multiplication. The * operator is associative and expressions with several multiplications at the same level may be rearranged by the compiler.

The binary / operator indicates division. When positive integers are divided truncation is toward 0, but the form of truncation is machine-dependent if either operand is negative. On all machines covered by this manual, the remainder has the same sign as the dividend. It is always true that $(a/b)*b + a\%b$ is equal to a (if b is not 0).

The binary % operator yields the remainder from the division of the first expression by the second. The usual arithmetic conversions are performed. The operands must not be float.

7.4 Additive operators

The additive operators + and - group left-to-right. The usual arithmetic conversions are performed. There are some additional type possibilities for each operator.

* As of this writing, sizeof expressions are unsigned only for the PDP-11 compiler; other compilers treat them as integers.

additive-expression:

expression + expression
expression - expression

The result of the + operator is the sum of the operands. A pointer to an object in an array and a value of any integral type may be added. The latter is in all cases converted to an address offset by multiplying it by the length of the object to which the pointer points. The result is a pointer of the same type as the original pointer, and which points to another object in the same array, appropriately offset from the original object. Thus if P is a pointer to an object in an array, the expression P+1 is a pointer to the next object in the array.

No further type combinations are allowed for pointers.

The + operator is associative and expressions with several additions at the same level may be rearranged by the compiler.

The result of the - operator is the difference of the operands. The usual arithmetic conversions are performed. Additionally, a value of any integral type may be subtracted from a pointer, and then the same conversions as for addition apply.

If two pointers to objects of the same type are subtracted, the result is converted (by division by the length of the object) to an int representing the number of objects separating the pointed-to objects. This conversion will in general give unexpected results unless the pointers point to objects in the same array, since pointers, even to objects of the same type, do not necessarily differ by a multiple of the object-length.

7.5 Shift operators

The shift operators << and >> group left-to-right. Both perform the usual arithmetic conversions on their operands, each of which must be integral. Then the right operand is converted to int; the type of the result is that of the left operand. The result is undefined if the right operand is negative, or greater than or equal to the length of the object in bits.

shift-expression:

expression << expression
expression >> expression

The value of E1<<E2 is E1 (interpreted as a bit pattern) left-shifted E2 bits; vacated bits are 0-filled. The value of E1>>E2 is E1 right-shifted E2 bit positions. The right shift is guaranteed to be logical (0-fill) if E1 is unsigned; otherwise it may be arithmetic (fill by a copy of the sign bit).

7.6 Relational operators

The relational operators group left-to-right, but this fact is not very useful; a<b<c does not mean what it seems to.

relational-expression:

expression < expression
expression > expression
expression <= expression
expression >= expression

The operators < (less than), > (greater than), <= (less than or equal to) and >= (greater than or equal to) all yield 0 if the specified relation is false and 1 if it is true. The type of the result is int. The usual arithmetic conversions are performed. Two pointers may be compared; the result depends on the relative locations in the address space of the pointed-to objects. Pointer comparison is portable only when the pointers point to objects in the same array.

7.7 Equality operators*equality-expression:*

expression == expression
expression != expression

The == (equal to) and the != (not equal to) operators are exactly analogous to the relational operators except for their lower precedence. (Thus a<b == c<d is 1 whenever a<b and c<d have the same truth-value).

A pointer may be compared to an integer only if the integer is the constant 0. A pointer to which 0 has been assigned is guaranteed not to point to any object, and will appear to be equal to 0; in conventional usage, such a pointer is considered to be null.

7.8 Bitwise AND operator

and-expression:
expression & expression

The & operator is associative and expressions involving & may be rearranged. The usual arithmetic conversions are performed; the result is the bitwise AND function of the operands. The operator applies only to integral operands.

7.9 Bitwise exclusive OR operator

exclusive-or-expression:
expression ^ expression

The ^ operator is associative and expressions involving ^ may be rearranged. The usual arithmetic conversions are performed; the result is the bitwise exclusive OR function of the operands. The operator applies only to integral operands.

7.10 Bitwise inclusive OR operator

inclusive-or-expression:
expression | expression

The | operator is associative and expressions involving | may be rearranged. The usual arithmetic conversions are performed; the result is the bitwise inclusive OR function of its operands. The operator applies only to integral operands.

7.11 Logical AND operator

logical-and-expression:
expression && expression

The && operator groups left-to-right. It returns 1 if both its operands are non-zero, 0 otherwise. Unlike &, && guarantees left-to-right evaluation; moreover the second operand is not evaluated if the first operand is 0.

The operands need not have the same type, but each must have one of the fundamental types or be a pointer. The result is always `int`.

7.12 Logical OR operator

logical-or-expression:
expression || expression

The || operator groups left-to-right. It returns 1 if either of its operands is non-zero, and 0 otherwise. Unlike |, || guarantees left-to-right evaluation; moreover, the second operand is not evaluated if the value of the first operand is non-zero.

The operands need not have the same type, but each must have one of the fundamental types or be a pointer. The result is always `int`.

7.13 Conditional operator

conditional-expression:
expression ? expression : expression

Conditional expressions group right-to-left. The first expression is evaluated and if it is non-zero, the result is the value of the second expression, otherwise that of third expression. If possible, the usual arithmetic conversions are performed to bring the second and third expressions to a common type; otherwise, if both are pointers of the same type, the result has the common type; otherwise, one must be a pointer and the other the constant 0, and the result has the type of the pointer. Only one of the second and third expressions is evaluated.

7.14 Assignment operators

There are a number of assignment operators, all of which group right-to-left. All require an lvalue as their left operand, and the type of an assignment expression is that of its left operand. The value is the value stored in the left operand after the assignment has taken place. The two parts of a compound assignment operator are separate tokens.

assignment-expression:

```

lvalue = expression
lvalue += expression
lvalue -= expression
lvalue *= expression
lvalue /= expression
lvalue %= expression
lvalue >>= expression
lvalue <<= expression
lvalue &= expression
lvalue ^= expression
lvalue |= expression

```

In the simple assignment with `=`, the value of the expression replaces that of the object referred to by the lvalue. If both operands have arithmetic type, the right operand is converted to the type of the left preparatory to the assignment. Second, both operands may be structures or unions of the same type. Finally, if the left operand is a pointer, the right operand must in general be a pointer of the same type; however the constant 0 may be assigned to a pointer, and it is guaranteed that this value will produce a null pointer distinguishable from a pointer to any object.

The behavior of an expression of the form `E1 op = E2` may be inferred by taking it as equivalent to `E1 = E1 op (E2)`; however, `E1` is evaluated only once. In `+=` and `-=`, the left operand may be a pointer, in which case the (integral) right operand is converted as explained in §7.4; all right operands and all non-pointer left operands must have arithmetic type.

7.15 Comma operator

comma-expression:

```

expression , expression

```

A pair of expressions separated by a comma is evaluated left-to-right and the value of the left expression is discarded. The type and value of the result are the type and value of the right operand. This operator groups left-to-right. In contexts where comma is given a special meaning, for example in lists of actual arguments to functions (§7.1) and lists of initializers (§8.6), the comma operator as described in this section can only appear in parentheses; for example,

```
f(a, (t=3, t+2), c)
```

has three arguments, the second of which has the value 5.

8. DECLARATIONS

Declarations are used to specify the interpretation which C gives to each identifier; they do not necessarily reserve storage associated with the identifier. Declarations have the form

declaration:

```

decl-specifiers declarator-listopt ;

```

The declarators in the declarator-list contain the identifiers being declared. The decl-specifiers consist of a sequence of type and storage class specifiers.

decl-specifiers:

```

type-specifier decl-specifiersopt
sc-specifier decl-specifiersopt

```

The list must be self-consistent in a way described below.

8.1 Storage class specifiers

The sc-specifiers are:

```
sc-specifier:
    auto
    static
    extern
    register
    typedef
```

The `typedef` specifier does not reserve storage and is called a “storage class specifier” only for syntactic convenience; it is discussed in §8.8. The meanings of the various storage classes were discussed in §4.

The `auto`, `static`, and `register` declarations also serve as definitions in that they cause an appropriate amount of storage to be reserved. In the `extern` case there must be an external definition (§10) for the given identifiers somewhere outside the function in which they are declared.

A `register` declaration is best thought of as an `auto` declaration, together with a hint to the compiler that the variables declared will be heavily used. Only the first few such declarations are effective. Moreover, only variables of certain types will be stored in registers; on the PDP-11, they are `int` or pointer. One other restriction applies to register variables: the address-of operator `&` cannot be applied to them. Smaller, faster programs can be expected if register declarations are used appropriately, but future improvements in code generation may render them unnecessary.

At most one sc-specifier may be given in a declaration. If the sc-specifier is missing from a declaration, it is taken to be `auto` inside a function, `extern` outside. Exception: functions are never automatic.

8.2 Type specifiers

The type-specifiers are

```
type-specifier:
    char
    short
    int
    long
    unsigned
    float
    double
    void
    struct-or-union-specifier
    typedef-name
    enum-specifier
```

The words `long`, `short`, and `unsigned` may be thought of as adjectives; the following combinations are acceptable.

```
short int
long int
unsigned int
unsigned char
long float
```

The meaning of the last is the same as `double`. Otherwise, at most one type-specifier may be given in a declaration. If the type-specifier is missing from a declaration, it is taken to be `int`.

Specifiers for structures, unions and enumerations are discussed in §8.5; declarations with `typedef` names are discussed in §8.8.

8.3 Declarators

The declarator-list appearing in a declaration is a comma-separated sequence of declarators, each of which may have an initializer.

declarator-list:
init-declarator
init-declarator , *declarator-list*

init-declarator:
declarator initializer_{opt}

Initializers are discussed in §8.6. The specifiers in the declaration indicate the type and storage class of the objects to which the declarators refer. Declarators have the syntax:

declarator:
identifier
 (*declarator*)
 * *declarator*
declarator ()
declarator [*constant-expression_{opt}*]

The grouping is the same as in expressions.

8.4 Meaning of declarators

Each declarator is taken to be an assertion that when a construction of the same form as the declarator appears in an expression, it yields an object of the indicated type and storage class. Each declarator contains exactly one identifier; it is this identifier that is declared.

If an unadorned identifier appears as a declarator, then it has the type indicated by the specifier heading the declaration.

A declarator in parentheses is identical to the unadorned declarator, but the binding of complex declarators may be altered by parentheses. See the examples below.

Now imagine a declaration

T D1

where **T** is a type-specifier (like `int`, etc.) and **D1** is a declarator. Suppose this declaration makes the identifier have type "... **T**," where the "..." is empty if **D1** is just a plain identifier (so that the type of **x** in "`int x`" is just `int`). Then if **D1** has the form

***D**

the type of the contained identifier is "... pointer to **T**."

If **D1** has the form

D ()

then the contained identifier has the type "... function returning **T**."

If **D1** has the form

D [*constant-expression*]

or

D []

then the contained identifier has type "... array of **T**." In the first case the constant expression is an expression whose value is determinable at compile time, and whose type is `int`. (Constant expressions are defined precisely in §15.) When several "array of" specifications are adjacent, a multi-dimensional array is created; the constant expressions which specify the bounds of the arrays may be missing only for the first member of the sequence. This elision is useful when the array is external and the actual definition, which allocates storage, is given elsewhere. The first constant-expression may also be omitted when the declarator is followed by initialization. In this case the size is calculated from the number of initial elements supplied.

An array may be constructed from one of the basic types, from a pointer, from a structure or union, or from another array (to generate a multi-dimensional array).

Not all the possibilities allowed by the syntax above are actually permitted. The restrictions are as follows: functions may not return arrays or functions, although they may return pointers to such things; there are no arrays of functions, although there may be arrays of pointers to functions. Likewise a structure or union may not contain a function, but it may contain a pointer to a function.

As an example, the declaration

```
int i, *ip, f(), *fip(), (*pfi());
```

declares an integer `i`, a pointer `ip` to an integer, a function `f` returning an integer, a function `fip` returning a pointer to an integer, and a pointer `pfi` to a function which returns an integer. It is especially useful to compare the last two. The binding of `*fip()` is `*(fip())`, so that the declaration suggests, and the same construction in an expression requires, the calling of a function `fip`, and then using indirection through the (pointer) result to yield an integer. In the declarator `(*pfi)()`, the extra parentheses are necessary, as they are also in an expression, to indicate that indirection through a pointer to a function yields a function, which is then called; it returns an integer.

As another example,

```
float fa[17], *afp[17];
```

declares an array of `float` numbers and an array of pointers to `float` numbers. Finally,

```
static int x3d[3][5][7];
```

declares a static three-dimensional array of integers, with rank $3 \times 5 \times 7$. In complete detail, `x3d` is an array of three items; each item is an array of five arrays; each of the latter arrays is an array of seven integers. Any of the expressions `x3d`, `x3d[i]`, `x3d[i][j]`, `x3d[i][j][k]` may reasonably appear in an expression. The first three have type "array," the last has type `int`.

8.5 Structure, union and enumeration declarations

A structure is an object consisting of a sequence of named members. Each member may have any type. A union is an object which may, at a given time, contain any one of several members. Structure and union specifiers have the same form.

struct-or-union-specifier:

```
struct-or-union { struct-decl-list }
struct-or-union identifier { struct-decl-list }
struct-or-union identifier
```

struct-or-union:

```
struct
union
```

The `struct-decl-list` is a sequence of declarations for the members of the structure or union:

struct-decl-list:

```
struct-declaration
struct-declaration struct-decl-list
```

struct-declaration:

```
type-specifier struct-declarator-list ;
```

struct-declarator-list:

```
struct-declarator
struct-declarator , struct-declarator-list
```

In the usual case, a `struct-declarator` is just a declarator for a member of a structure or union. A structure member may also consist of a specified number of bits. Such a member is also called a *field*; its length is set off from the field name by a colon.

struct-declarator:

```
declarator
declarator : constant-expression
: constant-expression
```

Within a structure, the objects declared have addresses which increase as their declarations are read left-to-right. Each non-field member of a structure begins on an addressing boundary appropriate to its type; therefore, there may be unnamed holes in a structure. Field members are packed into machine integers; they do not straddle words. A field which does not fit into the space remaining in a word is put into the next word. No field may be wider than a word.

Fields are assigned right-to-left on the PDP-11 and VAX-11, left-to-right on other machines.

A struct-declarator with no declarator, only a colon and a width, indicates an unnamed field useful for padding to conform to externally-imposed layouts. As a special case, an unnamed field with a width of 0 specifies alignment of the next field at a word boundary. The "next field" presumably is a field, not an ordinary structure member, because in the latter case the alignment would have been automatic.

The language does not restrict the types of things that are declared as fields, but implementations are not required to support any but integer fields. Moreover, even `int` fields may be considered to be unsigned. On the PDP-11, fields are not signed and have only integer values; on the VAX-11, fields declared with `int` are treated as containing a sign. For these reasons, it is strongly recommended that fields be declared as `unsigned`. In all implementations, there are no arrays of fields, and the address-of operator `&` may not be applied to them, so that there are no pointers to fields.

A union may be thought of as a structure all of whose members begin at offset 0 and whose size is sufficient to contain any of its members. At most one of the members can be stored in a union at any time.

A structure or union specifier of the second form, that is, one of

```
struct identifier { struct-decl-list }
union identifier { struct-decl-list }
```

declares the identifier to be the *structure tag* (or union tag) of the structure specified by the list. A subsequent declaration may then use the third form of specifier, one of

```
struct identifier
union identifier
```

Structure tags allow definition of self-referential structures; they also permit the long part of the declaration to be given once and used several times. It is illegal to declare a structure or union which contains an instance of itself, but a structure or union may contain a pointer to an instance of itself.

The names of members and tags do not conflict with each other or with ordinary variables. A particular name may not be used twice in the same structure, but the same name may be used in several different structures in the same scope.

A simple example of a structure declaration is

```
struct tnode {
    char tword[20];
    int count;
    struct tnode *left;
    struct tnode *right;
};
```

which contains an array of 20 characters, an integer, and two pointers to similar structures. Once this declaration has been given, the declaration

```
struct tnode s, *sp;
```

declares `s` to be a structure of the given sort and `sp` to be a pointer to a structure of the given sort. With these declarations, the expression

```
sp->count
```

refers to the `count` field of the structure to which `sp` points;

```
s.left
```

refers to the left subtree pointer of the structure `s`; and

```
s.right->tword[0]
```

refers to the first character of the `tword` member of the right subtree of `s`.

Enumerations are unique types with named constants. However, the current language treats enumeration variables and constants as being of `int` type.

```

enum-specifier:
    enum { enum-list }
    enum identifier { enum-list }
    enum identifier

enum-list:
    enumerator
    enum-list , enumerator

enumerator:
    identifier
    identifier = constant-expression

```

The identifiers in an enum-list are declared as constants, and may appear wherever constants are required. If no enumerators with = appear, then the values of the corresponding constants begin at 0 and increase by 1 as the declaration is read from left to right. An enumerator with = gives the associated identifier the value indicated; subsequent identifiers continue the progression from the assigned value.

The names of enumerators in the same scope must all be distinct from each other and from those of ordinary variables.

The role of the identifier in the enum-specifier is entirely analogous to that of the structure tag in a struct-specifier; it names a particular enumeration. For example,

```

enum color { chartreuse, burgundy, claret=10, winedark };
...
enum color *cp, col;
...
col = claret;
cp = &col;
...
if (*cp == burgundy) ...

```

makes `color` the enumeration-tag of a type describing various colors, and then declares `cp` as a pointer to an object of that type, and `col` as an object of that type. The possible values are drawn from the set {0,1,10,11}.

8.6 Initialization

A declarator may specify an initial value for the identifier being declared. The initializer is preceded by =, and consists of an expression or a list of values nested in braces.

```

initializer:
    = expression
    = { initializer-list }
    = { initializer-list , }

initializer-list:
    expression
    initializer-list , initializer-list
    { initializer-list }

```

All the expressions in an initializer for a static or external variable must be constant expressions, which are described in §15, or expressions which reduce to the address of a previously declared variable, possibly offset by a constant expression. Automatic or register variables may be initialized by arbitrary expressions involving constants, and previously declared variables and functions.

Static and external variables which are not initialized are guaranteed to start off as 0; automatic and register variables which are not initialized are guaranteed to start off as garbage.

When an initializer applies to a *scalar* (a pointer or an object of arithmetic type), it consists of a single expression, perhaps in braces. The initial value of the object is taken from the expression; the same conversions as for assignment are performed.

When the declared variable is an *aggregate* (a structure or array) then the initializer consists of a brace-enclosed, comma-separated list of initializers for the members of the aggregate, written in increasing subscript or member order. If the aggregate contains subaggregates, this rule applies recursively to the members of the aggregate. If there are fewer initializers in the list than there are members of the

aggregate, then the aggregate is padded with 0's. It is not permitted to initialize unions or automatic aggregates.

Braces may be elided as follows. If the initializer begins with a left brace, then the succeeding comma-separated list of initializers initializes the members of the aggregate; it is erroneous for there to be more initializers than members. If, however, the initializer does not begin with a left brace, then only enough elements from the list are taken to account for the members of the aggregate; any remaining members are left to initialize the next member of the aggregate of which the current aggregate is a part.

A final abbreviation allows a `char` array to be initialized by a string. In this case successive characters of the string initialize the members of the array.

For example,

```
int x[] = { 1, 3, 5 };
```

declares and initializes `x` as a 1-dimensional array which has three members, since no size was specified and there are three initializers.

```
float y[4][3] = {
    { 1, 3, 5 },
    { 2, 4, 6 },
    { 3, 5, 7 },
};
```

is a completely-bracketed initialization: 1, 3, and 5 initialize the first row of the array `y[0]`, namely `y[0][0]`, `y[0][1]`, and `y[0][2]`. Likewise the next two lines initialize `y[1]` and `y[2]`. The initializer ends early and therefore `y[3]` is initialized with 0. Precisely the same effect could have been achieved by

```
float y[4][3] = {
    1, 3, 5, 2, 4, 6, 3, 5, 7
};
```

The initializer for `y` begins with a left brace, but that for `y[0]` does not, therefore 3 elements from the list are used. Likewise the next three are taken successively for `y[1]` and `y[2]`. Also,

```
float y[4][3] = {
    { 1 }, { 2 }, { 3 }, { 4 }
};
```

initializes the first column of `y` (regarded as a two-dimensional array) and leaves the rest 0.

Finally,

```
char msg[] = "Syntax error on line %s\n";
```

shows a character array whose members are initialized with a string.

8.7 Type names

In two contexts (to specify type conversions explicitly by means of a cast, and as an argument of `sizeof`) it is desired to supply the name of a data type. This is accomplished using a "type name," which in essence is a declaration for an object of that type which omits the name of the object.

type-name:

type-specifier abstract-declarator

abstract-declarator:

empty

(abstract-declarator)

** abstract-declarator*

abstract-declarator ()

abstract-declarator [constant-expression_{opt}]

To avoid ambiguity, in the construction

(abstract-declarator)

the abstract-declarator is required to be non-empty. Under this restriction, it is possible to identify uniquely the location in the abstract-declarator where the identifier would appear if the construction were

a declarator in a declaration. The named type is then the same as the type of the hypothetical identifier. For example,

```
int
int *
int *[3]
int (*)(3)
int *()
int (*)()
```

name respectively the types “integer,” “pointer to integer,” “array of 3 pointers to integers,” “pointer to an array of 3 integers,” “function returning pointer to integer,” and “pointer to function returning an integer.”

8.8 Typedef

Declarations whose “storage class” is `typedef` do not define storage, but instead define identifiers which can be used later as if they were type keywords naming fundamental or derived types.

```
typedef-name:
    identifier
```

Within the scope of a declaration involving `typedef`, each identifier appearing as part of any declarator therein becomes syntactically equivalent to the type keyword naming the type associated with the identifier in the way described in §8.4. For example, after

```
typedef int MILES, *KCLICKSP;
typedef struct { double re, im;} complex;
```

the constructions

```
MILES distance;
extern KCLICKSP metricp;
complex z, *zp;
```

are all legal declarations; the type of `distance` is `int`, that of `metricp` is “pointer to `int`,” and that of `z` is the specified structure. `zp` is a pointer to such a structure.

`typedef` does not introduce brand new types, only synonyms for types which could be specified in another way. Thus in the example above `distance` is considered to have exactly the same type as any other `int` object.

9. STATEMENTS

Except as indicated, statements are executed in sequence.

9.1 Expression statement

Most statements are expression statements, which have the form

```
expression ;
```

Usually expression statements are assignments or function calls.

9.2 Compound statement, or block

So that several statements can be used where one is expected, the compound statement (also, and equivalently, called “block”) is provided:

```
compound-statement:
    { declaration-listopt statement-listopt }
```

```
declaration-list:
    declaration
    declaration declaration-list
```

```
statement-list:
    statement
    statement statement-list
```

If any of the identifiers in the declaration-list were previously declared, the outer declaration is pushed down for the duration of the block, after which it resumes its force.

Any initializations of `auto` or `register` variables are performed each time the block is entered at the top. It is currently possible (but a bad practice) to transfer into a block; in that case the initializations are not performed. Initializations of `static` variables are performed only once when the program begins execution. Inside a block, `extern` declarations do not reserve storage so initialization is not permitted.

9.3 Conditional statement

The two forms of the conditional statement are

```
if ( expression ) statement
if ( expression ) statement else statement
```

In both cases the expression is evaluated and if it is non-zero, the first substatement is executed. In the second case the second substatement is executed if the expression is 0. As usual the "else" ambiguity is resolved by connecting an `else` with the last encountered `else-less if`.

9.4 While statement

The while statement has the form

```
while ( expression ) statement
```

The substatement is executed repeatedly so long as the value of the expression remains non-zero. The test takes place before each execution of the statement.

9.5 Do statement

The do statement has the form

```
do statement while ( expression ) ;
```

The substatement is executed repeatedly until the value of the expression becomes zero. The test takes place after each execution of the statement.

9.6 For statement

The for statement has the form

```
for ( expression-1opt ; expression-2opt ; expression-3opt ) statement
```

This statement is equivalent to

```
expression-1 ;
while ( expression-2 ) {
    statement
    expression-3 ;
}
```

Thus the first expression specifies initialization for the loop; the second specifies a test, made before each iteration, such that the loop is exited when the expression becomes 0; the third expression often specifies an incrementing that is performed after each iteration.

Any or all of the expressions may be dropped. A missing `expression-2` makes the implied `while` clause equivalent to `while(1)`; other missing expressions are simply dropped from the expansion above.

9.7 Switch statement

The `switch` statement causes control to be transferred to one of several statements depending on the value of an expression. It has the form

```
switch ( expression ) statement
```

The usual arithmetic conversion is performed on the expression, but the result must be `int`. The statement is typically compound. Any statement within the statement may be labeled with one or more case prefixes as follows:

```
case constant-expression :
```

where the constant expression must be `int`. No two of the case constants in the same switch may have the same value. Constant expressions are precisely defined in §15.

There may also be at most one statement prefix of the form

```
default :
```

When the `switch` statement is executed, its expression is evaluated and compared with each case constant. If one of the case constants is equal to the value of the expression, control is passed to the statement following the matched case prefix. If no case constant matches the expression, and if there is a `default` prefix, control passes to the prefixed statement. If no case matches and if there is no `default` then none of the statements in the switch is executed.

`case` and `default` prefixes in themselves do not alter the flow of control, which continues unimpeded across such prefixes. To exit from a switch, see `break`, §9.8.

Usually the statement that is the subject of a switch is compound. Declarations may appear at the head of this statement, but initializations of automatic or register variables are ineffective.

9.8 Break statement

The statement

```
break ;
```

causes termination of the smallest enclosing `while`, `do`, `for`, or `switch` statement; control passes to the statement following the terminated statement.

9.9 Continue statement

The statement

```
continue ;
```

causes control to pass to the loop-continuation portion of the smallest enclosing `while`, `do`, or `for` statement; that is to the end of the loop. More precisely, in each of the statements

```
while (...) {      do {          for (...) {
    ...            ...          ...
    contin: ;      contin: ;      contin: ;
}                  } while (...); }

```

a `continue` is equivalent to `goto contin`. (Following the `contin:` is a null statement, §9.13.)

9.10 Return statement

A function returns to its caller by means of the `return` statement, which has one of the forms

```
return ;
return expression ;
```

In the first case the returned value is undefined. In the second case, the value of the expression is returned to the caller of the function. If required, the expression is converted, as if by assignment, to the type of the function in which it appears. Flowing off the end of a function is equivalent to a return with no returned value.

9.11 Goto statement

Control may be transferred unconditionally by means of the statement

```
goto identifier ;
```

The identifier must be a label (§9.12) located in the current function.

9.12 Labeled statement

Any statement may be preceded by label prefixes of the form

```
identifier :
```

which serve to declare the identifier as a label. The only use of a label is as a target of a `goto`. The scope of a label is the current function, excluding any sub-blocks in which the same identifier has been redeclared. See §11.

9.13 Null statement

The null statement has the form

```
;
```

A null statement is useful to carry a label just before the } of a compound statement or to supply a null body to a looping statement such as `while`.

10. EXTERNAL DEFINITIONS

A C program consists of a sequence of external definitions. An external definition declares an identifier to have storage class `extern` (by default) or perhaps `static`, and a specified type. The type-specifier (§8.2) may also be empty, in which case the type is taken to be `int`. The scope of external definitions persists to the end of the file in which they are declared just as the effect of declarations persists to the end of a block. The syntax of external definitions is the same as that of all declarations, except that only at this level may the code for functions be given.

10.1 External function definitions

Function definitions have the form

```
function-definition:
    decl-specifiersopt function-declarator function-body
```

The only sc-specifiers allowed among the decl-specifiers are `extern` or `static`; see §11.2 for the distinction between them. A function declarator is similar to a declarator for a “function returning ...” except that it lists the formal parameters of the function being defined.

```
function-declarator:
    declarator ( parameter-listopt )
```

```
parameter-list:
    identifier
    identifier , parameter-list
```

The function-body has the form

```
function-body:
    declaration-list compound-statement
```

The identifiers in the parameter list, and only those identifiers, may be declared in the declaration list. Any identifiers whose type is not given are taken to be `int`. The only storage class which may be specified is `register`; if it is specified, the corresponding actual parameter will be copied, if possible, into a register at the outset of the function.

A simple example of a complete function definition is

```
int max(a, b, c)
int a, b, c;
{
    int m;
    m = (a > b) ? a : b;
    return((m > c) ? m : c);
}
```

Here `int` is the type-specifier; `max(a, b, c)` is the function-declarator; `int a, b, c;` is the declaration-list for the formal parameters; `{ ... }` is the block giving the code for the statement.

C converts all `float` actual parameters to `double`, so formal parameters declared `float` have their declaration adjusted to read `double`. Also, since a reference to an array in any context (in particular as an actual parameter) is taken to mean a pointer to the first element of the array, declarations of formal parameters declared “array of ...” are adjusted to read “pointer to ...”.

10.2 External data definitions

An external data definition has the form

```
data-definition:  
declaration
```

The storage class of such data may be `extern` (which is the default) or `static`, but not `auto` or `register`.

11. SCOPE RULES

A C program need not all be compiled at the same time: the source text of the program may be kept in several files, and precompiled routines may be loaded from libraries. Communication among the functions of a program may be carried out both through explicit calls and through manipulation of external data.

Therefore, there are two kinds of scope to consider: first, what may be called the *lexical scope* of an identifier, which is essentially the region of a program during which it may be used without drawing "undefined identifier" diagnostics; and second, the scope associated with external identifiers, which is characterized by the rule that references to the same external identifier are references to the same object.

11.1 Lexical scope

The lexical scope of identifiers declared in external definitions persists from the definition through the end of the source file in which they appear. The lexical scope of identifiers which are formal parameters persists through the function with which they are associated. The lexical scope of identifiers declared at the head of a block persists until the end of the block. The lexical scope of labels is the whole of the function in which they appear.

In all cases, however, if an identifier is explicitly declared at the head of a block, including the block constituting a function, any declaration of that identifier outside the block is suspended until the end of the block.

Remember also (§8.5) that identifiers associated with ordinary variables on the one hand and those associated with structure and union members and tags on the other form two disjoint classes which do not conflict. Members and tags follow the same scope rules as other identifiers. `typedef` names are in the same class as ordinary identifiers. They may be redeclared in inner blocks, but an explicit type must be given in the inner declaration:

```
typedef float distance;  
...  
{  
    auto int distance;  
    ...
```

The `int` must be present in the second declaration, or it would be taken to be a declaration with no declarators and type `distance*`.

11.2 Scope of externals

If a function refers to an identifier declared to be `extern`, then somewhere among the files or libraries constituting the complete program there must be an external definition for the identifier. All functions in a given program which refer to the same external identifier refer to the same object, so care must be taken that the type and size specified in the definition are compatible with those specified by each function which references the data.

The appearance of the `extern` keyword in an external definition indicates that storage for the identifiers being declared will be allocated in another file. Thus in a multi-file program, an external data definition without the `extern` specifier must appear in exactly one of the files. Any other files which wish to give an external definition for the identifier must include the `extern` in the definition. The identifier can be initialized only in the declaration where storage is allocated.

Identifiers declared `static` at the top level in external definitions are not visible in other files. Functions may be declared `static`.

* It is agreed that the ice is thin here.

12. COMPILER CONTROL LINES

The C compiler contains a preprocessor capable of macro substitution, conditional compilation, and inclusion of named files. Lines beginning with `#` communicate with this preprocessor. These lines have syntax independent of the rest of the language; they may appear anywhere and have effect which lasts (independent of scope) until the end of the source program file.

12.1 Token replacement

A compiler-control line of the form

```
#define identifier token-string
```

causes the preprocessor to replace subsequent instances of the identifier with the given string of tokens. Semicolons in, or at the end of, the token-string are part of that string. A line of the form

```
#define identifier( identifier , ... , identifier ) token-string
```

where there is no space between the first identifier and the `(`, is a macro definition with arguments. Subsequent instances of the first identifier followed by a `(`, a sequence of tokens delimited by commas, and a `)` are replaced by the token string in the definition. Each occurrence of an identifier mentioned in the formal parameter list of the definition is replaced by the corresponding token string from the call. The actual arguments in the call are token strings separated by commas; however commas in quoted strings or protected by parentheses do not separate arguments. The number of formal and actual parameters must be the same. Strings and character constants in the token-string are scanned for formal parameters, but strings and character constants in the rest of the program are not scanned for defined identifiers. to replacement.

In both forms the replacement string is rescanned for more defined identifiers. In both forms a long definition may be continued on another line by writing `\` at the end of the line to be continued.

This facility is most valuable for definition of "manifest constants," as in

```
#define TABSIZE 100
int table[TABSIZE];
```

A control line of the form

```
#undef identifier
```

causes the identifier's preprocessor definition to be forgotten.

12.2 File inclusion

A compiler control line of the form

```
#include "filename "
```

causes the replacement of that line by the entire contents of the file *filename*. The named file is searched for first in the directory of the original source file, and then in a sequence of specified or standard places. Alternatively, a control line of the form

```
#include <filename >
```

searches only the specified or standard places, and not the directory of the source file. (How the places are specified is not part of the language.)

`#include`'s may be nested.

12.3 Conditional compilation

A compiler control line of the form

```
#if constant-expression
```

checks whether the constant expression evaluates to non-zero. (Constant expressions are discussed in §15; the following additional restriction applies here: the constant expression may not contain `sizeof` or an enumeration constant.) A control line of the form

```
#ifdef identifier
```

checks whether the identifier is currently defined in the preprocessor; that is, whether it has been the subject of a `#define` control line. A control line of the form

```
#ifndef identifier
```

checks whether the identifier is currently undefined in the preprocessor.

All three forms are followed by an arbitrary number of lines, possibly containing a control line

```
#else
```

and then by a control line

```
#endif
```

If the checked condition is true then any lines between `#else` and `#endif` are ignored. If the checked condition is false then any lines between the test and an `#else` or, lacking an `#else`, the `#endif`, are ignored.

These constructions may be nested.

12.4 Line control

For the benefit of other preprocessors which generate C programs, a line of the form

```
#line constant "filename"
```

causes the compiler to believe, for purposes of error diagnostics, that the line number of the next source line is given by the constant and the current input file is named by the identifier. If the identifier is absent the remembered file name does not change.

13. IMPLICIT DECLARATIONS

It is not always necessary to specify both the storage class and the type of identifiers in a declaration. The storage class is supplied by the context in external definitions and in declarations of formal parameters and structure members. In a declaration inside a function, if a storage class but no type is given, the identifier is assumed to be `int`; if a type but no storage class is indicated, the identifier is assumed to be `auto`. An exception to the latter rule is made for functions, because `auto` functions do not exist. If the type of an identifier is "function returning ...", it is implicitly declared to be `extern`.

In an expression, an identifier followed by (and not already declared is contextually declared to be "function returning `int`".

14. TYPES REVISITED

This section summarizes the operations which can be performed on objects of certain types.

14.1 Structures and unions

Structures and unions may be assigned, passed as arguments to functions, and returned by functions. Other plausible operators, such as equality comparison and structure casts, are not implemented.

In a reference to a structure or union member, the name on the right must specify a member of the aggregate named or pointed to by the expression on the left. In general, a member of a union may not be inspected unless the value of the union has been assigned using that same member. However, one special guarantee is made by the language in order to simplify the use of unions: if a union contains several structures that share a common initial sequence, and if the union currently contains one of these structures, it is permitted to inspect the common initial part of any of the contained structures. For example, the following is a legal fragment:


```

union {
    struct {
        int type;
    } n;
    struct {
        int type;
        int intnode;
    } ni;
    struct {
        int type;
        float floatnode;
    } nf;
} u;
...
u.nf.type = FLOAT;
u.nf.floatnode = 3.14;
...
if (u.n.type == FLOAT)
    ... sin(u.nf.floatnode) ...

```

14.2 Functions

There are only two things that can be done with a function: call it, or take its address. If the name of a function appears in an expression not in the function-name position of a call, a pointer to the function is generated. Thus, to pass one function to another, one might say

```

int f();
...
g(f);

```

Then the definition of `g` might read

```

g(funcp)
int (*funcp)();
{
    ...
    (*funcp)();
    ...
}

```

Notice that `f` must be declared explicitly in the calling routine since its appearance in `g(f)` was not followed by `(`.

14.3 Arrays, pointers, and subscripting

Every time an identifier of array type appears in an expression, it is converted into a pointer to the first member of the array. Because of this conversion, arrays are not lvalues. By definition, the subscript operator `[]` is interpreted in such a way that `E1[E2]` is identical to `*((E1)+(E2))`. Because of the conversion rules which apply to `+`, if `E1` is an array and `E2` an integer, then `E1[E2]` refers to the `E2`-th member of `E1`. Therefore, despite its asymmetric appearance, subscripting is a commutative operation.

A consistent rule is followed in the case of multi-dimensional arrays. If `E` is an n -dimensional array of rank $i \times j \times \dots \times k$, then `E` appearing in an expression is converted to a pointer to an $(n-1)$ -dimensional array with rank $j \times \dots \times k$. If the `*` operator, either explicitly or implicitly as a result of subscripting, is applied to this pointer, the result is the pointed-to $(n-1)$ -dimensional array, which itself is immediately converted into a pointer.

For example, consider

```
int x[3][5];
```

Here `x` is a 3×5 array of integers. When `x` appears in an expression, it is converted to a pointer to (the first of three) 5-membered arrays of integers. In the expression `x[i]`, which is equivalent to `*(x+i)`, `x` is first converted to a pointer as described; then `i` is converted to the type of `x`, which involves multiplying `i` by the length the object to which the pointer points, namely 5 integer objects. The results are

added and indirection applied to yield an array (of 5 integers) which in turn is converted to a pointer to the first of the integers. If there is another subscript the same argument applies again; this time the result is an integer.

It follows from all this that arrays in C are stored row-wise (last subscript varies fastest) and that the first subscript in the declaration helps determine the amount of storage consumed by an array but plays no other part in subscript calculations.

14.4 Explicit pointer conversions

Certain conversions involving pointers are permitted but have implementation-dependent aspects. They are all specified by means of an explicit type-conversion operator, §§7.2 and 8.7.

A pointer may be converted to any of the integral types large enough to hold it. Whether an `int` or `long` is required is machine dependent. The mapping function is also machine dependent, but is intended to be unsurprising to those who know the addressing structure of the machine. Details for some particular machines are given below.

An object of integral type may be explicitly converted to a pointer. The mapping always carries an integer converted from a pointer back to the same pointer, but is otherwise machine dependent.

A pointer to one type may be converted to a pointer to another type. The resulting pointer may cause addressing exceptions upon use if the subject pointer does not refer to an object suitably aligned in storage. It is guaranteed that a pointer to an object of a given size may be converted to a pointer to an object of a smaller size and back again without change.

For example, a storage-allocation routine might accept a size (in bytes) of an object to allocate, and return a `char` pointer; it might be used in this way.

```
extern char *alloc();
double *dp;

dp = (double *) alloc(sizeof(double));
*dp = 22.0 / 7.0;
```

`alloc` must ensure (in a machine-dependent way) that its return value is suitable for conversion to a pointer to `double`; then the *use* of the function is portable.

The pointer representation on the PDP-11 corresponds to a 16-bit integer and measures bytes. `chars` have no alignment requirements; everything else must have an even address.

On the VAX-11, pointers are 32 bits long and measure bytes. Elementary objects are aligned on a boundary equal to their length, except that `double` quantities need be aligned only on even 4-byte boundaries. Aggregates are aligned on the strictest boundary required by any of their constituents.

On the Honeywell 6000, a pointer corresponds to a 36-bit integer; the word part is in the left 18 bits, and the two bits that select the character in a word lie just to their right. Thus `char` pointers measure units of 2^{16} bytes; everything else is measured in units of 2^{18} machine words. `double` quantities and aggregates containing them must lie on an even word address ($0 \bmod 2^{19}$).

The IBM 370 and the Interdata 8/32 are similar. On each, pointers are 32-bit quantities that measure bytes; elementary objects are aligned on a boundary equal to their length, so pointers to `short` must be $0 \bmod 2$, to `int` and `float` $0 \bmod 4$, and to `double` $0 \bmod 8$. Aggregates are aligned on the strictest boundary required by any of their constituents.

15. CONSTANT EXPRESSIONS

In several places C requires expressions which evaluate to a constant: after `case`, as array bounds, and in initializers. In the first two cases, the expression can involve only integer constants, character constants, enumeration constants, and `sizeof` expressions, possibly connected by the binary operators

+ - * / % & | ^ << >> == != < > <= >=

or by the unary operators

- ~

or by the ternary operator

?:

Parentheses can be used for grouping, but not for function calls.

More latitude is permitted for initializers; besides constant expressions as discussed above, one can also apply the unary `&` operator to external or static objects, and to external or static arrays subscripted with a constant expression. The unary `&` can also be applied implicitly by appearance of unsubscripted

arrays and functions. The basic rule is that initializers must evaluate either to a constant or to the address of a previously declared external or static object plus or minus a constant.

Less latitude is allowed for constant expressions after `#if`; `sizeof` expressions and enumeration constants are not permitted.

16. PORTABILITY CONSIDERATIONS

Certain parts of C are inherently machine dependent. The following list of potential trouble spots is not meant to be all-inclusive, but to point out the main ones.

Purely hardware issues like word size and the properties of floating point arithmetic and integer division have proven in practice to be not much of a problem. Other facets of the hardware are reflected in differing implementations. Some of these, particularly sign extension (converting a negative character into a negative integer) and the order in which bytes are placed in a word, are a nuisance that must be carefully watched. Most of the others are only minor problems.

The number of `register` variables that can actually be placed in registers varies from machine to machine, as does the set of valid types. Nonetheless, the compilers all do things properly for their own machine; excess or invalid `register` declarations are ignored.

Some difficulties arise only when dubious coding practices are used. It is exceedingly unwise to write programs that depend on any of these properties.

The order of evaluation of function arguments is not specified by the language. It is right to left on the PDP-11 and VAX-11, left to right on the others. The order in which side effects take place is also unspecified.

Since character constants are really objects of type `int`, multi-character character constants may be permitted. The specific implementation is very machine dependent, however, because the order in which characters are assigned to a word varies from one machine to another.

Fields are assigned to words and characters to integers right-to-left on the PDP-11 and VAX-11 and left-to-right on other machines. These differences are invisible to isolated programs which do not indulge in type punning (for example, by converting an `int` pointer to a `char` pointer and inspecting the pointed-to storage), but must be accounted for when conforming to externally-imposed storage layouts.

The language accepted by the various compilers differs in minor details. Most notably, the current PDP-11 compiler will not initialize structures containing bit-fields, and does not accept a few assignment operators in certain contexts where the value of the assignment is used.

17. ANACHRONISMS

Because C is an evolving language, certain obsolete constructions may be found in older programs. Although some versions of the compiler support such anachronisms, they have by and large disappeared, leaving only a portability problem behind.

Earlier versions of C used the form `=op` instead of `op=` for assignment operators. This leads to ambiguities, typified by

```
x = - 1
```

which assigns `-1` to `x`, but previously decremented `x`.

The syntax of initializers has changed: previously, the equals sign that introduces an initializer was not present, so instead of

```
int x = 1;
```

one used

```
int x 1;
```

The change was made because the initialization

```
int f (1)
```

resembles a function declaration closely enough to confuse the compilers.

A structure or union member reference is a chain of member references (qualifications) that are prefixed by either a pointer to a structure or union or a structure or union proper. Because each qualification implies the addition of an offset within an address computation, older compilers (which failed to check for membership in the appropriate structure or union) allowed omission of those qualifications with an offset of zero. Complete qualification is now required.

Previous versions of the compiler were lax in detecting mixed assignments involving pointers and arithmetic quantities. These are now remarked upon.

18. SYNTAX SUMMARY

This summary of C syntax is intended more for aiding comprehension than as an exact statement of the language.

18.1 Expressions

The basic expressions are:

expression:
primary
 * *expression*
 & *lvalue*
 - *expression*
 ! *expression*
 ~ *expression*
 ++ *lvalue*
 -- *lvalue*
lvalue ++
lvalue --
 sizeof *expression*
 (*type-name*) *expression*
expression binop *expression*
expression ? *expression* : *expression*
lvalue asgnop *expression*
expression , *expression*

primary:
identifier
constant
string
 (*expression*)
primary (*expression-list*_{opt})
primary [*expression*]
primary . *identifier*
primary -> *identifier*

lvalue:
identifier
primary [*expression*]
lvalue . *identifier*
primary -> *identifier*
 * *expression*
 (*lvalue*)

The primary-expression operators

() [] . ->

have highest priority and group left-to-right. The unary operators

* & - ! ~ ++ -- sizeof (*type-name*)

have priority below the primary operators but higher than any binary operator, and group right-to-left. Binary operators group left-to-right; they have priority decreasing as indicated below. The conditional operator groups right to left.

binop:

```

*      /      %
+      -
>>    <<<
<      >      <=      >=
==     !=
&
^
!
&&
||
?:

```

Assignment operators all have the same priority, and all group right-to-left.

asgnop:

```

=  +=  -=  *=  /=  %=  >>=  <<=  &=  ^=  |=

```

The comma operator has the lowest priority, and groups left-to-right.

18.2 Declarations

declaration:
decl-specifiers *init-declarator-list*_{opt} ;

decl-specifiers:
type-specifier *decl-specifiers*_{opt}
sc-specifier *decl-specifiers*_{opt}

sc-specifier:
auto
static
extern
register
typedef

type-specifier:
char
short
int
long
unsigned
float
double
void
struct-or-union-specifier
typedef-name
enum-specifier

enum-specifier:
enum { *enum-list* }
enum *identifier* { *enum-list* }
enum *identifier*

enum-list:
enumerator
enum-list , *enumerator*

enumerator:
identifier
identifier = constant-expression

init-declarator-list:
init-declarator
init-declarator , init-declarator-list

init-declarator:
declarator initializer_{opt}

declarator:
identifier
(declarator)
** declarator*
declarator ()
declarator [constant-expression_{opt}]

struct-or-union-specifier:
struct { struct-decl-list }
struct identifier { struct-decl-list }
struct identifier
union { struct-decl-list }
union identifier { struct-decl-list }
union identifier

struct-decl-list:
struct-declaration
struct-declaration struct-decl-list

struct-declaration:
type-specifier struct-declarator-list ;

struct-declarator-list:
struct-declarator
struct-declarator , struct-declarator-list

struct-declarator:
declarator
declarator : constant-expression
: constant-expression

initializer:
= expression
= { initializer-list }
= { initializer-list , }

initializer-list:
expression
initializer-list , initializer-list
{ initializer-list }

type-name:
type-specifier abstract-declarator

abstract-declarator:
 empty
 (*abstract-declarator*)
 * *abstract-declarator*
abstract-declarator ()
abstract-declarator [*constant-expression*_{opt}]

typedef-name:
 identifier

18.3 Statements

compound-statement:
 { *declaration-list*_{opt} *statement-list*_{opt} }

declaration-list:
 declaration
 declaration *declaration-list*

statement-list:
 statement
 statement *statement-list*

statement:
compound-statement
expression ;
 if (*expression*) *statement*
 if (*expression*) *statement* else *statement*
 while (*expression*) *statement*
 do *statement* while (*expression*) ;
 for (*expression-1*_{opt} ; *expression-2*_{opt} ; *expression-3*_{opt}) *statement*
 switch (*expression*) *statement*
 case *constant-expression* : *statement*
 default : *statement*
 break ;
 continue ;
 return ;
 return *expression* ;
 goto *identifier* ;
identifier : *statement*
 ;

18.4 External definitions

program:
external-definition
external-definition program

external-definition:
function-definition
data-definition

function-definition:
*type-specifier*_{opt} *function-declarator* *function-body*

function-declarator:
declarator (*parameter-list*_{opt})

parameter-list:
identifier
identifier , parameter-list

function-body:
declaration-list compound-statement

data-definition:
*extern*_{opt} *type-specifier*_{opt} *init-declarator-list*_{opt} ;
*static*_{opt} *type-specifier*_{opt} *init-declarator-list*_{opt} ;

18.5 Preprocessor

```
#define identifier token-string  
#define identifier ( identifier , ... , identifier ) token-string  
#undef identifier  
#include "filename"  
#include <filename>  
#if constant-expression  
#ifdef identifier  
#ifndef identifier  
#else  
#endif  
#line constant "filename"
```

January 1981

A Guide to the C Library for UNIX Users

C. D. Perez

Bell Laboratories
Murray Hill, New Jersey 07974

1. INTRODUCTION

The C language on the UNIX† system has been traditionally provided with a rich supply of often-used routines formed into libraries selectable at load time. When the interest in portability heightened, the C library kept pace with other software being modified, and the library known as */lib/libS.a* superseded the older attempts at portability [1]. This new library [2] concentrated on input-output functions that removed the user from close contact with operating-system features. It also introduced new string functions and some memory-allocation routines.

It is to the advantage of the C programmer to become acquainted with the C library functions and to keep up-to-date with new versions. To select routines from the C library is to choose available code that has been tuned for portability and efficiency. This document is meant to acquaint the programmer with a selection of functions from the C library that are commonly used and to point out differences among functions, special features, and occasionally precautions about function usage. The veteran user of the C library will find in this compendium an update to previously published information about the library.

Section 2 describes the current changes and additions to the contents of */lib/libS.a* since the Ritchie document was published. The bulk of the information appears in Attachment A, which is intended to be a user's reference tool. Function descriptions appear alphabetically within logical groupings. Where it seems helpful, examples are supplied. The values returned by the functions are identified in a way that suggests their use in portable code.

2. UPDATE INFORMATION

2.1 General

The standard library */lib/libS.a* no longer exists separate from the rest of the C library; these routines have been incorporated into the standard UNIX C library, */lib/libc.a*. This library encompasses input-output functions, routines for character type recognition and translation, space allocation, file status and a few miscellaneous routines of general use as well as many functions specific to the operating system.

Three files exist with definitions of constants, and macros that are used by many of the C library functions. *Stdio.h* contains the definitions of `NULL`, `EOF`, `FILE`, and `BUFSIZ`. The standard input file (*stdin*), standard output file (*stdout*), and standard error file (*stderr*) are also defined there. These are included in a program with `#include <stdio.h>`. The file *ctype.h* provides the macro definitions for the variety of character classifications that is now possible. Any program using those facilities must contain the line `#include <ctype.h>`. The functions that handle signals need to include the signal definitions. This can be done with `#include <signal.h>`.

† UNIX is a trademark of Bell Laboratories.

2.2 Space Allocation.

Calloc was designed to be used for acquiring space initialized to zero; *malloc* is now available to allocate a chunk of uninitialized space, and *realloc* to change the size of an already allocated amount of space. *Cfree* has been renamed *free*, and returns space acquired by any of the above three functions.

2.3 Input-Output Functions

The function *fopen* may now be supplied with new options that allow updating of a file (*r+*, *w+*, *a+*). An added routine *fdopen* acts as a bridge between the low-level UNIX input-output functions and the "standard" technique of opening a stream. *Printf* provides more versatile formatting. For operating systems that support the concept of *pipes*, and the *shell*, the functions *popen* and *pclose* add a facility for creating a pipe between the calling process and a command supplied as the argument.

2.4 Status

To acquire information about a file, *feof*, *ferror*, and *fileno* have been available. Now a function named *clearerr* is added. It resets the error condition indicated by *ferror* while the stream remains open.

2.5 Character Types

New macros added to the collection in *ctype.h* are *isalnum* (alphanumeric test), *ispunct* (for recognizing punctuation characters), *isctrl* (to identify certain control characters), *isascii* (to find ASCII characters), *isgraph* (having visible graphic representation), and *isxdigit* (hexadecimal digits with either upper-case or lower-case letters). *Toascii* can be used to translate characters into ASCII; *toupper* and *tolower* are useful in changing the case of a letter.

2.6 Some Conventions

When the overhead of a function call could be substantial, because the routine suggests repetitive use, it is likely to have been implemented as a macro. *Getchar* is an illustration of this. Any "function" coded as a macro is noted in its description. In these cases the user should beware of the hazards of macro expansion on complex arguments. Cases should be avoided where arguments are automatically incremented or decremented, are evaluated more than once, contain their own macros or function calls, or whose order of operations is unclear after expansion. In short, only simple arguments are safe to use with macros. In a few cases the C library provides both a function call and a faster macro version to perform a similar task.

Some function names have changed in order to follow the established convention. To insure that the uniqueness of function names is preserved even if truncation occurs on some systems, those functions dealing with entire strings are named *str...*; those functions that consider only the first *n* characters of a string are named *strn...*

2.7 Other Additions

Software signals are implemented by two functions, *gsignal* and *ssignal*, to generate and catch error conditions respectively [3]. This facility allows the user to raise signals to be handled in whatever way seems useful; the C Library code will eventually raise signals so that calling programs, such as UNIX commands, might be enhanced to respond to such signals.

Tmpnam can be used to create a name for a temporary file. *Ctermid* retrieves the terminal identifier from the system, while *cuserid* retrieves the user ID. In each of these three functions, the user may choose to supply space for the safe storage of that name, or accept an internal storage place of suitable size.

Tmpfile provides an unnamed temporary file that continues in existence until the termination of the process that requested it.

3. ACKNOWLEDGEMENTS

The author is grateful to J. F. Maranzano and L. Rosler for their careful reading of this document, and to A. R. Koenig for his help during its preparation. T. A. Dolotta helped to format this document.

4. REFERENCES

- [1] Lesk, M. E. *The Portable C Library*, Bell Laboratories (May 1975).
- [2] Ritchie, D. M. *A New Input/Output Package*, Bell Laboratories (May 1977).
- [3] Koenig, A. R. A Proposal for Software Signals, private communication (Apr. 14, 1978).
- [4] Dolotta, T. A., Olsson, S. B., and Petrucelli, A. G. (eds.). *UNIX User's Manual—Release 3.0*, Bell Laboratories (June 1980).

Attachment A

COMMON C LIBRARY FUNCTIONS

FILE ACCESS

fclose *#include* <stdio.h>
 int fclose(*stream*)
 FILE **stream*;

Fclose closes a file that was opened by *fopen*, frees any buffers after emptying them, and returns zero on success, non-zero on error. *Exit* calls *fclose* for all open files as part of its processing.

fdopen *#include* <stdio.h>
 FILE **fdopen* (*fd*, *type*)
 int *fd*;
 char **type*;

Fdopen is used strictly on UNIX systems and therefore is not a portable function. Its value is in providing a bridge between the low-level input-output (I/O) facilities of UNIX and the standard I/O functions. *Fdopen* associates a stream with a valid file descriptor obtained from a UNIX system call (e.g., *open*). *Type* is the same mode (*r*, *w*, *a*, *r+*, *w+*, *a+*) that was used in the original creation of a file identified by *fd*. *Fdopen* returns a pointer to the associated stream, or NULL if unsuccessful.

Example:

```
int fd;
char *name = "myfile";
FILE *strm;

fd = open(name,0);
    :
    :
    if((strm = fdopen(fd,"r")) == NULL)
        fprintf(stderr,"Error on %d\n",fd);
```

fileno *#include* <stdio.h>
 int fileno (*stream*)
 FILE **stream*;

Implemented as a macro on UNIX, (and contained in the file *stdio.h*), *fileno* returns an integer file descriptor associated with a valid *stream*. Any existing non-UNIX implementations may have different meanings for the integer which is returned. *Fileno* is used by many other standard functions in the C library.

fopen *#include* <stdio.h>
 FILE **fopen* (*filename*, *type*)
 char **filename*, **type*;

Fopen opens a file named *filename* and returns a pointer to a structure (hereafter referred to as *stream*), containing the data necessary to handle a stream of data. *Type* is one of the following character strings:

- r** used to open for reading.
- w** used to open for writing, which truncates an existing file to zero length or creates a new file.

- a used to append, that is, open for writing at the end of a file, or create a new file.
- r+ update reading, which means open for reading and allow writing, positions the file pointer at the beginning of the file.
- w+ update writing, which means open for writing and allow reading, truncates an existing file to zero length or creates a new file.
- a+ update appending, which means open for writing, positions to the end of the file and allows for subsequent reads and writes (all writes being forced to current end-of-file position). If the file does not exist, it will be created.

For the update options, *fseek* or *rewind* can be used to trigger the change from reading to writing, or vice versa. (Reaching EOF on input will also permit writing without further formality.) *Fopen* returns a NULL pointer if *filename* cannot be opened. On non-UNIX implementations, file names may be different from UNIX-like names. The update functions are particularly applicable to stream I/O and allow the creation of temporary files for both reading and writing. The non-UNIX implementations contain many options other than those mentioned above.

Example:

```
FILE *fp;
char *file;

if((fp = fopen(file,"r")) == NULL)
    fprintf(stderr, "Cannot open %s\n",file);
```

freopen `#include <stdio.h>`
*FILE *freopen (newfile, type, stream)*
*char *newfile, *type;*
*FILE *stream;*

Freopen accepts a pointer, *stream*, to a previously opened file; the old file is closed, and then the new file is opened. The principal motivation for *freopen* is the desire to attach the names *stdin*, *stdout*, and *stderr* to specified files. On a successful *freopen*, the stream pointer is returned; otherwise, NULL is returned, indicating that either the closing of *stream* failed, or the file closing took place and the reopening failed. *Freopen* is of limited portability; it can not be implemented in all environments.

Example:

```
char *newfile;
FILE *nfile;

if((nfile = freopen(newfile,"r",stdout)) == NULL)
    fprintf(stderr,"Cannot reopen %s\n",newfile);
```

fseek `#include <stdio.h>`
int fseek (stream, offset, ptrname)
*FILE *stream;*
long offset;
int ptrname;

Fseek positions a stream to a location *offset* distance from the beginning, current position, or end of a file, depending on the values 0, 1, 2, respectively, for *ptrname*. On UNIX the offset unit is bytes; other implementations may be different. (For example, on GCOS the offset is three 12-bit fields of block, logical-record number, and offset-into-record number.) The return values are 0 on success and EOF on failure. *Fseek* may be used with both buffered and unbuffered files. As implemented, the function cannot be ported to the OS/370 environment.

Example:

To position to the end of a file:

```
FILE *stream;
fseek(stream, 0L, 2);
```

pclose *#include <stdio.h>*
int pclose (stream)
*FILE *stream;*

Pclose closes a stream opened by *popen*. It returns the exit status of the command that was issued as the first argument of its corresponding *popen*, or -1 if the stream was not opened by *popen*. The function name *pclose* means an entirely different thing in the OS/370 environment.

popen *#include <stdio.h>*
*FILE *popen (command, type)*
*char *command, *type;*

Popen is used to create a pipe between the calling process and a command to be executed. The first argument is a shell command line; *type* is the I/O mode for the pipe, and may be either *r* for reading or *w* for writing. The function returns a stream pointer to be used for I/O on the standard input or output of the command. A NULL pointer is returned if an error occurs.

Example:

```
FILE *pstrm;
if((pstrm=popen("tr mvp MVP", "w"))== NULL)
    fprintf(stderr, "popen error\n");
fprintf(pstrm, "a message via the pipe...\n");
if(pclose(pstrm) == -1)
    fprintf(stderr, "Pclose error\n");
```

results in:

```
a Message Via the Pipe
```

rewind *#include <stdio.h>*
int rewind(stream)
*FILE *stream;*

Rewind sets the position of the next operation at the beginning of the file associated with *stream*, retaining the current mode of the file. It is the equivalent of *fseek (stream, 0L, 0)*;

setbuf *#include <stdio.h>*
setbuf (stream, buf)
*FILE *stream;*
*char *buf;*

This function allows the user to choose his own buffer for I/O, or to choose the unbuffered mode. Use it after opening and before reading or writing; it reduces the number of system read/write requests. When *buf* is set to NULL, I/O is unbuffered. The default status for all I/O streams is buffered *unless* the stream is connected to a communication-line device. When the character routine *putc* is used with an output *stream* that is unbuffered, there will result one system call per character transferred. On the other hand, when any of the string output routines *printf*, *sprintf*, *fwrite*, *puts*, and *fputs* is used with an output *stream* that is unbuffered, buffering will be temporarily and transparently established so that the resultant output character string will be passed to the system in one system call.

The choice to buffer I/O brings with it the responsibility for flushing any data that may remain in a last, partially-filled buffer. *Flush* or *fclose* perform this task. The constant `BUFSIZ` in *stdio.h* tells how big the character array `buf` is. It is well-chosen for the machine on which UNIX is running. (On GCOS the function is implemented as a null macro, because GCOS does not need such a function.)

Example:

```
setbuf (stdout, malloc(BUFSIZ));
```

FILE STATUS

clearerr *#include <stdio.h>*
clearerr(stream)
*FILE *stream;*

Clearerr is used to reset the error condition on `stream`. The need for *clearerr* arises on UNIX implementations where the error indicator is not reset after a query.

feof *#include <stdio.h>*
int feof (stream)
*FILE *stream;*

Feof, which is implemented as a macro on UNIX, returns non-zero if an input operation on `stream` has reached end of file; otherwise, a zero is returned. *Feof* should be used in conjunction with any I/O function whose return value is not a clear indicator of an end-of-file condition. Such functions are *fread* and *getw*.

Example:

```
int *x;
FILE *stream;

do
    *x++ = getw(stream);
while(!feof(stream));
```

ferror *#include <stdio.h>*
int ferror (stream)
*FILE *stream;*

Ferror tests for an indication of error on `stream`. It returns a non-zero value (true) when an error is found, and a zero otherwise. Calls to *ferror* do not clear the error condition, hence the *clearerr* function is needed for that purpose. The user should be aware that, after an error, further use of the file may cause strange results. On UNIX *ferror* is implemented as a macro.

Example:

```
FILE *stream;
int *x;

while(!ferror(stream))
    putw(*x++, stream);
```

ftell *#include <stdio.h>*
long ftell (stream)
*FILE *stream;*

Ftell is used to determine the current offset relative to the beginning of the file associated with `stream`. It returns the current value of the offset; in UNIX it returns the offset value in bytes. On error, a value of `-1` is returned. This function is useful in obtaining an offset for subsequent *fseek* calls.

INPUT FUNCTIONS

fgetc *#include <stdio.h>*
int fgetc (stream)
*FILE *stream;*

This is the function version of the macro *getc* and acts identically to *getc*. Because *fgetc* is a function and not a macro, it can be used in debugging to set breakpoints on *fgetc* and when the side effects of macro processing of the argument is a problem. Furthermore, it can be passed as an argument.

fgets *#include <stdio.h>*
*char *fgets (s,n,stream)*
*char *s;*
int n;
*FILE *stream;*

Fgets reads from *stream* into the area pointed to by *s* either *n*-1 characters or an entire string including its new-line terminator, whichever comes first. A final null character is affixed to the data read. It returns the pointer *s* on success, and NULL on end-of-file or error. *Fgets* differs from the function *gets* in that it can read from other than *stdin*, and that it appends the new-line at the end of input when the size of the string is longer than or equal to *n*. More importantly, it provides control over the size of the string to be read that is not available with *gets*.

Example:

```
char msg[MAX];
FILE *myfile;

while(fgets(msg,MAX,myfile) != NULL)
    printf("%s\n",msg);
```

fread *#include <stdio.h>*
*int fread((char *)ptr, sizeof(*ptr), nitems, stream)*
*FILE *stream;*

This function reads from *stream* the next *nitems* whose size is the same as the size of the item pointed to by *ptr*, into a sufficiently large area starting at *ptr*. It returns the number of items read. In UNIX, *fread* makes use of the function *getc*. It is often used in combination with *feof* and *ferror* to obtain a clear indication of the file status.

Example:

```
FILE *pstm;
char mesg[100];

while(fread((char *)mesg,sizeof(*mesg),1,pstm) == 1)
    printf("%s\n",mesg);
```

fscanf *#include <stdio.h>*
int fscanf (stream, format[, argptr]...)
*char *format;*
*FILE *stream;*

Fscanf accepts input from the file associated with *stream*, and deposits it into the storage area pointed to by the respective argument pointers after conversion according to the specified formats. Format specifications are those that appear in the *UNIX User's Manual* [4] entry for *scanf*(3S). *Fscanf* differs from *scanf* in that it can read from other than *stdin*. The function returns the number of successfully deposited input arguments, or EOF on error or unexpected end-of-input.

Example:

```

FILE *file;
long pay;
char name[15];
char pan[7];

    fscanf(file,"%6s%14s%ld\n",pan,name,&pay);
    if(pay<50000)
        printf("%ld raise for %s.\n",pay/10,name);

```

If the input data is:

```
020202MaryJones 15000
```

the resulting output is:

```
$1500 raise for MaryJones.
```

```

getc    #include <stdio.h>
          int getc (stream)
          FILE *stream;

```

Getc returns the next character from the named *stream*. On UNIX it is implemented as a macro to avoid the overhead of a function call. On error or end-of-file it returns an EOF. *Fgetc* should be used when it is necessary to avoid the side effects of argument processing by the macro *getc*.

```

getchar #include <stdio.h>
          int getchar()

```

This is identical to *getc (stdin)*.

```

gets    #include <stdio.h>
          char *gets(s)
          char *s;

```

Gets reads a string of characters up to a new-line from *stdin* and places them in the area pointed to by *s*. The new-line character which ended the string is replaced by the null character. The return values are *s* on success, NULL on error or end-of-file. The simple example below presumes the size of the string read into *msg* will not exceed *SIZE* in length. If used in conjunction with *strlen*, a dangerous overflow can be detected, though not prevented.

Example:

```

char msg[SIZE];
char *s;
    s = msg;
    while (gets(s) != NULL)
        printf("%s\n",s);

```

```

getw    #include <stdio.h>
          int getw (stream)
          FILE *stream;

```

Getw reads the next word from the file associated with *stream*. On success it returns the word; on error or end of file, it returns EOF. However, because EOF could be a valid word, this function is best used with *feof* and *ferror*.

Example:

```
FILE *stream;
int *x;
do
    *x++ = getw(stream);
while (!feof(stream));
```

scanf

```
#include <stdio.h>
int scanf (format[, argptr]...)
char *format;
```

Scanf reads input from *stdin* and deposits it, according to the specified formats, in the storage area pointed to by the respective argument pointers. The correct format specifications can be found in the *UNIX User's Manual* [4] entry for *scanf*(3S). For input from other streams than *stdin* use *fscanf*; for input from a character array use *sscanf*. The return values are the number of successfully deposited input arguments, or EOF on error or unexpected end-of-input.

Example:

```
long number;
scanf("%ld",&number);
printf("%ld is %s", number, number%2? "odd": "even");
```

sscanf

```
#include <stdio.h>
sscanf (s, format [, pointer]...)
char *s;
char *format;
```

Sscanf accepts input from a character string *s* and deposits it, according to the specified formats, in the storage area pointed to by the respective argument pointers. Format specifications appear in the *UNIX User's Manual* [4] entry for *scanf*(3S). This function returns the number of successfully deposited arguments.

Example:

```
char datestr[] = {"THU MAR 29 11:04:40 EST 1979"};
char month[4];
char year[5];
sscanf(datestr,"%*3s%3s%*2s%*8s%*3s%4s",month,year);
printf("%s, %s\n",month,year);
```

The result is:

```
MAR, 1979
```

ungetc

```
#include <stdio.h>
int ungetc (c, stream)
int c;
FILE *stream;
```

Ungetc puts the character *c* back on the file associated with *stream*. One character (but never EOF) is assured of being put back. If successful, the function returns *c*; otherwise, EOF is returned.

Example:

```
while(!isspace (c = getc(stdin)))
    ;
ungetc(c,stdin);
```

This code puts the first character that is not white space back onto the standard input stream.

OUTPUT FUNCTIONS

fflush *#include <stdio.h>*
 int fflush (stream)
 *FILE *stream;*

Fflush takes action to guarantee that any data contained in file buffers and not yet written out will be written. It is used by *fclose* to flush a stream. No action is taken on files not open for writing. The return values are zero for success, EOF on error.

fprintf *#include <stdio.h>*
 int fprintf (stream, format[, arg]...)
 *FILE *stream;*
 *char *format;*

Fprintf provides formatted output to a named stream. The function *printf* may be used if the destination is *stdout*. Specifications for formats are available in the *UNIX User's Manual* [4] entry for *printf(3S)*. On success, *fprintf* returns the number of characters transmitted; otherwise, EOF is returned.

Example:

```
int *filename;
int c;
    if(c==EOF)
        fprintf(stderr,"EOF on %s\n",filename);
```

fputc *#include <stdio.h>*
 int fputc (c,stream)
 int c;
 *FILE *stream;*

Fputc performs the same task as *putc*; that is, it writes the character *c* to the file associated with *stream*, but is implemented as a function rather than a macro. It is preferred to *putc* when the side effects of macro processing of arguments are a problem. On success, it returns the character written; on failure it returns EOF.

Example:

```
FILE *in, *out;
int c;
    while ((c = fgetc(in)) != EOF)
        fputc(c,out);
```

fputs *#include <stdio.h>*
 int fputs(s,stream)
 *char *s;*
 *FILE *stream;*

Fputs copies a string to the output file associated with *stream*. In UNIX it uses the function *putc* to do this. It is different from *puts* in two ways: it allows any output stream to be specified, and it does not affix a new-line to the output. For an example, see *puts*.

fwrite *#include <stdio.h>*
 *int fwrite ((char *)ptr, sizeof (*ptr),nitems,stream)*
 *FILE *stream;*

Beginning at *ptr*, this function writes up to *nitems* of data of the type pointed to by *ptr* into output *stream*. It returns the number of items actually written.

For the GCOS implementation, `ptr` must be on a machine-word boundary. Like `fread` this function should be used in conjunction with `ferror` to detect the error condition.

Example:

```
char msg[] = {"My message to write out\n"};
FILE *pstrm;

if(fwrite(msg,(sizeof(*msg)-1),1,pstrm) != 1)
    fprintf(stderr,"Output error\n");
```

printf

```
#include <stdio.h>
int printf(format[, arg]...)
char *format;
```

`Printf` provides formatted output on `stdout`. The specifications for the available formats are given in the *UNIX User's Manual* [4] entry for `printf(3S)`. `Fprintf` and `sprintf` are related functions that write output onto other than the standard output. In case of error, implementations are not consistent in their output. On success, `printf` returns the number of characters transmitted; otherwise, EOF is returned.

Example:

```
int num = 10;
char msg[] = {"ten"};
printf("%d - %o - %s\n", num, num, msg);
```

results in the line:

```
10 - 12 - ten;
```

putc

```
#include <stdio.h>
int putc(c,stream)
int c;
FILE *stream;
```

`Putc` writes the character `c` to the file associated with `stream`. On success, it returns the character written; on error it returns EOF. Because it is implemented as a macro, side effects may result from argument processing. In such cases, the equivalent function `fputc` should be used.

Example:

```
#define PROMPT()      putc('\7',stderr)      /* BEL */
```

putchar

```
#include <stdio.h>
int putchar(c)
int c;
```

`Putchar` is defined as `putc(c, stdout)`. It returns the character written on success, or EOF on error.

Example:

```
char *cp;
char x[SIZE];

for(cp=x;cp<(x+SIZE);cp++)
    putchar(*cp);
```

puts

```
#include <stdio.h>
int puts(s)
char *s;
```

The function copies the string pointed to by *s* without its terminating null character to *stdout*. A new-line character is appended. The UNIX implementation uses the macro *putchar* (which calls *putc*).

Example:

```
puts("I will append a new-line");
fputs("\tsome more data ", stdout);
puts("and now a new-line");
```

The resulting output is:

```
I will append a new-line
    some more data and now a new-line
```

putw *#include <stdio.h>*
int putw(w,stream)
*FILE *stream;*
int w;

Putw appends word *w* to the output *stream*. As with *getw*, the proper way to check for an error or end-of-file is to use the *feof* and *ferror* functions.

Example:

```
int info;
while(!feof(stream))
    putw(info,stream);
```

sprintf *#include <stdio.h>*
int sprintf(s, format, [, arg] ...)
*char *s;*
*char *format;*

Sprintf allows for formatted output to be placed in a character array pointed to by *s*. *Sprintf* adds a null at the end of the formatted output. See the *UNIX User's Manual* [4] entry for *printf(3S)* for the specification of formats. It is the user's responsibility to provide an array of sufficient length. Other related functions, *printf* and *fprintf*, handle similar kinds of formatted output. *Sprintf* can be used to build formatted arrays in memory, to be changed dynamically before output, or to be used to call other routines. The comparable input function is *scanf*. On success, *sprintf* returns the number of characters transmitted; otherwise, EOF is returned.

Example:

```
char cmd[100];
char *doc = "/usr/src/cmd/cp.c"
int width = 50;
int length = 60;

    sprintf(cmd,"pr -w%d -l%d %s\n",width,length,doc);
system(cmd);
```

The above code executes the *pr* command to print the source of the *cp* command.

STRING FUNCTIONS

strcat *char *strcat(dst,src)*
*char *dst, *src;*

Strcat appends characters in the string pointed to by *src* to the end of the string pointed to by *dst*, and places a null character after the last character copied. It returns a pointer to *dst*. To concatenate strings up to a maximum number of characters, use *strncat*.

Example:

```
char *myfile;
char dir[L_cuserid+5] = "/usr/";
    myfile = (strcat(dir,cuserid(0)));
```

The result is the concatenation of the login name onto the end of the string `dir`.

strchr *char *strchr(s,c)*
*char *s;*
int c;

Strchr searches a string pointed to by `s`, for the leftmost occurrence of the character `c`. It returns a pointer to the character found, or `NULL` if `c` does not occur in the string.

Example:

```
int length;
char *a;
register char *b;

length = ((b=strchr(a,' ')) == NULL?0:b - a);
```

The resulting `length` is the number of characters up to the first blank in the string pointed to by `a`.

strcmp *char *strcmp(s1,s2)*
*char *s1, *s2;*

Strcmp compares the characters in the string `s1` and `s2`. It returns an integer value, greater than, equal to, or less than zero, depending on whether `s1` is lexicographically greater than, equal to, or less than `s2`.

Example:

```
#define EQ(x,y) !strcmp(x,y)
```

strcpy *char *strcpy(dst, src)*
*char *dst, *src;*

Strcpy copies the characters (including the null terminator) from the string pointed to by `src` into the string pointed to by `dst`. A pointer to `dst` is returned.

Example:

```
char dst[] = "UPPER CASE";
char src[] = "this is lower case";

printf("%s\n",strcpy(dst,src+8));
```

results in:

```
lower case
```

strlen *int strlen(s)*
*char *s;*

Strlen counts the number of characters starting at the character pointed to by `s` up to, but not including, the first null character. It returns the integer count.

Example:

```
char nextitem[SIZE];
char series[MAX];

if(strlen(series)) strcat(series,",");
strcat(series,nextitem);
```

strncat *char *strncat(dst, src, n)*
 *char *dst, *src;*
 int n;

Strncat appends a maximum of *n* characters of the string pointed to by *src* and then a null character to the string pointed to by *dst*. It returns a pointer to *dst*.

Example:

```
char dst[] = "cover";
char src[] = "letter";

printf("%s\n", strncat(dst, src, 3));
```

The output is:

```
coverlet
```

strncmp *int strncmp(s1, s2, n)*
 *char *s1, *s2;*
 int n;

Strncmp compares two strings for at most *n* characters and returns an integer greater than, equal to, or less than zero as *s1* is lexicographically greater than, equal to or less than *s2*.

Example:

```
char filename [] = "/dev/ttyx";
if(strncmp (filename+5, "tty", 3) == 0)
    printf("success\n");
```

strncpy *char *strncpy(dst, src, n)*
 *char *dst, *src;*
 int n;

Strncpy copies *n* characters of the string pointed to by *src* into the string pointed to by *dst*. Null padding or truncation of *src* occurs as necessary. A pointer to *dst* is returned.

Example:

```
char buf [MAX];
char date [29] = {"Fri Jun 29 09:35:44 EDT 1979"};
char *day = buf;

strncpy(day, date, 3);
```

After executing this code, *day* points to the string **Fri**.

strrchr *char *strrchr(s, c)*
 *char *s;*
 int c;

Strrchr searches a string pointed to by *s*, for the rightmost occurrence of the character *c*. It returns a pointer to the character found, or **NULL** if *c* does not occur in the string.

Example:

```
char reverse[] = "NAME NO ONE MAN";
printf(strrchr (reverse, 'M'));
```

results in:

```
MAN
```

CHARACTER CLASSIFICATION

isalnum *#include <ctype.h>*
 int isalnum(c)
 int c;

This macro determines whether or not the character *c* is an alphanumeric character ([A-Za-z0-9]). It returns zero for false and non-zero for true.

isalpha *#include <ctype.h>*
 int isalpha(c)
 int c;

This macro determines whether or not the character *c* is an alphabetic character ([A-Za-z]). It returns zero for false and non-zero for true.

isascii *#include <ctype.h>*
 int isascii(c)
 int c;

This macro determines whether or not the integer value supplied is an ASCII character; that is, a character whose octal value ranges from 000 to 177. It returns zero for false and non-zero for true.

iscntrl *#include <ctype.h>*
 int iscntrl(c)
 int c;

This macro determines whether or not the character *c* when mapped to ASCII is a control character (that is, octal 177 or 000-037). It returns zero for false and non-zero for true.

isdigit *#include <ctype.h>*
 int isdigit(c)
 int c;

This macro determines whether or not the character *c* is a digit. It returns zero for false and non-zero for true.

isgraph *#include <ctype.h>*
 int isgraph(c)
 int c;

This macro determines whether or not the character *c* has a graphic representation (that is, is an ASCII code between octal 041 and 176 inclusive).

islower *#include <ctype.h>*
 int islower(c)
 int c;

This macro determines whether or not the character *c* is a lower-case letter. It returns zero for false and non-zero for true.

isprint *#include <ctype.h>*
 int isprint(c)
 int c;

This macro determines whether or not the character *c* is a printable character. (This includes spaces.) It returns zero for false and non-zero for true.

ispunct *#include <ctype.h>*
 int ispunct(c)
 int c;

This macro determines whether or not the character *c* is a punctuation character (neither a control character nor an alphanumeric). It returns zero for false and non-zero for true.

isspace *#include <ctype.h>*
 int isspace(c)
 int c;

This macro determines whether or not the character *c* is a form of white space (that is, a blank, horizontal or vertical tab, carriage return, form-feed or new-line). It returns zero for false and non-zero for true.

isupper *#include <ctype.h>*
 int isupper(c)
 int c;

This macro determines whether or not the character *c* is an upper-case letter. It returns zero for false and non-zero for true.

isxdigit *#include <ctype.h>*
 int isxdigit(c)
 int c;

This macro determines whether or not the character *c* is a hexadecimal digit (upper- and lower-case letters are equivalent). It returns zero for false and non-zero for true.

CHARACTER TRANSLATION

toascii *#include <ctype.h>*
 int toascii(c)
 int c;

The macro *toascii* maps the input character into its ASCII equivalent; it usually does nothing in the UNIX environment. In a non-ASCII environment, it is useful when one needs to convert into ASCII any characters that are used as indices into tables that are sorted in the ASCII collating sequence.

Example:

```
FILE *oddstrm;
      if(!isdigit (toascii(getw(oddstrm))))
          fprintf(stderr,"bad data\n");
```

tolower *#include <ctype.h>*
 int tolower(c)
 int c;

If the argument *c* passed to the function *tolower* is an upper-case letter, the lower-case representation of *c* is returned; otherwise, *c* is returned. For a faster routine, use *_tolower*, which is implemented as a macro; however, its argument *must* be an upper-case letter.

Example:

```
if(tolower(getchar()) != 'y')
    exit(0);
```

toupper *#include <ctype.h>*
 int toupper (c)
 int c;

If the argument *c* passed to the function *toupper* is a lower-case letter, the upper-case representation of *c* is returned; otherwise, *c* is returned. For a faster routine, use *_toupper*; however, its argument *must* be a lower-case letter.

Example:

```
if(toupper (getchar()) != 'Y')
    exit(0);
```

SPACE ALLOCATION

calloc *char *calloc(n, size)*
 unsigned n, size;

Calloc allocates enough storage for an array of *n* items aligned for any use, each of *size* bytes. The space is initialized to zero. *Calloc* returns a pointer to the beginning of the allocated space, or a *NULL* pointer on failure.

Example:

```
char *t;
int n;
unsigned size;

if(t=calloc((unsigned)n, size) == NULL)
    fprintf(stderr,"Out of space.\n");
```

free *free(ptr)*
 *char *ptr;*

Free is used in conjunction with the space allocating functions *malloc*, *calloc*, or *realloc*. *Ptr* is a pointer supplied by one of these routines. The effect is to free the space previously allocated.

malloc *char *malloc(size)*
 unsigned size;

Malloc allocates *size* bytes of storage beginning on a word boundary. It returns a pointer to the beginning of the allocated space, or a *NULL* pointer on failure to acquire space. For space initialized to zero, see *calloc*.

Example:

```
int n;
char *t;
unsigned size;

if(t=malloc((unsigned)n) == NULL)
    fprintf(stderr,"Out of space.\n");
```

realloc *char *realloc (ptr, size)*
 *char *ptr;*
 unsigned size;

Given *ptr* which was supplied by a call to *malloc* or *calloc*, and a new byte *size*, *size*, *realloc* returns a pointer to the block of space of *size* bytes. This function is useful to do storage compacting along with *malloc* and *free*.

MISCELLANEOUS FUNCTIONS

ctermid *#include <stdio.h>*
 *char *ctermid(s)*
 *char *s;*

Ctermid provides a string that can be used as a file name, (*/dev/tty*), to identify the controlling terminal for the running process. Unlike the function *ttyname* it is disassociated from the machine-dependent concept of a file descriptor. If an argument of zero is supplied, the string is stored internally and will be overwritten on the next call to *ctermid*. A non-zero argument is treated as a pointer to a sufficiently large storage area where the string is placed.

cuserid *#include <stdio.h>*
 *char *cuserid(s)*
 *char *s;*

Cuserid composes a string representation of the login name of the owner of the current process. A zero argument results in the string being stored in an internal area; in this case a pointer to that area is returned on success, and a NULL on failure. A non-zero argument is assumed to be a pointer to a repository of size *L_cuserid* (contained in *ctype.h*). On failure a null character will be inserted in place of a string and a NULL is returned.

Example:

```
puts (cuserid((char *) NULL));
```

gsignal *#include <signal.h>*
 int gsignal (sig)
 int sig;

Along with *ssignal*, *gsignal* implements a facility for software signals. A software signal is raised by a call to *gsignal*. Raising a software signal causes the action established by *signal* to be taken. The argument *sig* identifies the signal to be set. If *sig* is a value defined in *signal.h*, then *gsignal* returns that value. If an action function was established for *sig*, then the action is reset to the default value, the action function is performed with argument *sig*, and the return value is the return value of the action function. In any abnormal case, *gsignal* returns the value 0 and takes no other action.

Example:

```
char *buf;
if((buf = gets(string))!=NULL) gsignal(2);
```

ssignal *#include <signal.h>*
 *int (*ssignal (sig,action))()*
 *int sig, (*action)();*

Ssignal along with *gsignal* implements a software signal facility. An action for a software signal is established by a call to *ssignal*. *Sig* is the number identifying the type of signal for which an action is to be established. The numbers currently defined are found in *signal.h*. *Action* is either the name of a user-defined action function or one of the constants defined in *signal.h*. *Ssignal* returns the action previously established for that signal type; in abnormal circumstances *ssignal* returns a default of zero.

Example:

```

main() {
    int error();
    ssignal(2, error);
    :
}
error(x) {
    int x;
    printf("Software signal %d has been caught.\n",x);
}

```

system *#include <stdio.h>*
system(string)
*char *string;*

System passes the argument *string* to the operating system as a command line. It returns the exit status of the command executed.

Example:

```

if(!system ("cmp -s file1 file2"))
    printf("The two files are identical.\n");

```

tmpnam *#include <stdio.h>*
*char *tmpnam(s)*
*char *s;*

Tmpnam generates a file name that can be used for a temporary file. If *s* is zero, it returns a pointer to a character string containing that name in an internal storage area. For a non-zero value in *s*, the file name is stored in a sufficiently large area pointed to by *s* (see *L_tmpnam* in *ctype.h*) and *s* is the return value as well.

tmpfile *#include <stdio.h>*
*FILE *tmpfile ()*

Tmpfile creates a scratch file opened for update. It stays in existence only during the life of the process issuing the function call and is inherited across forks. It returns a pointer to the *FILE* associated with the opened stream. On error, it returns *NULL*.

LINT, a C Program Checker

S. C. Johnson

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Lint is a command which examines C source programs, detecting a number of bugs and obscurities. It enforces the type rules of C more strictly than the C compilers. It may also be used to enforce a number of portability restrictions involved in moving programs between different machines and/or operating systems. Another option detects a number of wasteful, or error prone, constructions which nevertheless are, strictly speaking, legal.

Lint accepts multiple input files and library specifications, and checks them for consistency.

The separation of function between *lint* and the C compilers has both historical and practical rationale. The compilers turn C programs into executable files rapidly and efficiently. This is possible in part because the compilers do not do sophisticated type checking, especially between separately compiled programs. *Lint* takes a more global, leisurely view of the program, looking much more carefully at the compatibilities.

This document discusses the use of *lint*, gives an overview of the implementation, and gives some hints on the writing of machine independent C code.

Introduction and Usage

Suppose there are two C¹ source files, *file1.c* and *file2.c*, which are ordinarily compiled and loaded together. Then the command

```
lint file1.c file2.c
```

produces messages describing inconsistencies and inefficiencies in the programs. The program enforces the typing rules of C more strictly than the C compilers (for both historical and practical reasons) enforce them. The command

```
lint -p file1.c file2.c
```

will produce, in addition to the above messages, additional messages which relate to the portability of the programs to other operating systems and machines. Replacing the `-p` by `-h` will produce messages about various error-prone or wasteful constructions which, strictly speaking, are not bugs. Saying `-hp` gets the whole works.

The next several sections describe the major messages; the document closes with sections discussing the implementation and giving suggestions for writing portable C. An appendix gives a summary of the *lint* options.

A Word about Philosophy

Many of the facts which *lint* needs may be impossible to discover. For example, whether a given function in a program ever gets called may depend on the input data. Deciding whether *exit* is ever called is equivalent to solving the famous "halting problem," known to be recursively undecidable.

Thus, most of the *lint* algorithms are a compromise. If a function is never mentioned, it can never be called. If a function is mentioned, *lint* assumes it can be called; this is not necessarily so, but in practice is quite reasonable.

Lint tries to give information with a high degree of relevance. Messages of the form "xxx might be a bug" are easy to generate, but are acceptable only in proportion to the fraction of real bugs they uncover. If this fraction of real bugs is too small, the messages lose their credibility and serve merely to clutter up the output, obscuring the more important messages.

Keeping these issues in mind, we now consider in more detail the classes of messages which *lint* produces.

Unused Variables and Functions

As sets of programs evolve and develop, previously used variables and arguments to functions may become unused; it is not uncommon for external variables, or even entire functions, to become unnecessary, and yet not be removed from the source. These "errors of commission" rarely cause working programs to fail, but they are a source of inefficiency, and make programs harder to understand and change. Moreover, information about such unused variables and functions can occasionally serve to discover bugs; if a function does a necessary job, and is never called, something is wrong!

Lint complains about variables and functions which are defined but not otherwise mentioned. An exception is variables which are declared through explicit `extern` statements but are never referenced; thus the statement

```
extern float sin();
```

will evoke no comment if *sin* is never used. Note that this agrees with the semantics of the C compiler. In some cases, these unused external declarations might be of some interest; they can be discovered by adding the `-x` flag to the *lint* invocation.

Certain styles of programming require many functions to be written with similar interfaces; frequently, some of the arguments may be unused in many of the calls. The `-v` option is available to suppress the printing of complaints about unused arguments. When `-v` is in effect, no messages are produced about unused arguments except for those arguments which are unused and also declared as register arguments; this can be considered an active (and preventable) waste of the register resources of the machine.

There is one case where information about unused, or undefined, variables is more distracting than helpful. This is when *lint* is applied to some, but not all, files out of a collection which are to be loaded together. In this case, many of the functions and variables defined may not be used, and, conversely, many functions and variables defined elsewhere may be used. The `-u` flag may be used to suppress the spurious messages which might otherwise appear.

Set/Used Information

Lint attempts to detect cases where a variable is used before it is set. This is very difficult to do well; many algorithms take a good deal of time and space, and still produce messages about perfectly valid programs. *Lint* detects local variables (automatic and register storage classes) whose first use appears physically earlier in the input file than the first assignment to the variable. It assumes that taking the address of a variable constitutes a "use," since the actual use may occur at any later time, in a data dependent fashion.

The restriction to the physical appearance of variables in the file makes the algorithm very simple and quick to implement, since the true flow of control need not be discovered. It does

mean that *lint* can complain about some programs which are legal, but these programs would probably be considered bad on stylistic grounds (e.g. might contain at least two **goto**'s). Because static and external variables are initialized to 0, no meaningful information can be discovered about their uses. The algorithm deals correctly, however, with initialized automatic variables, and variables which are used in the expression which first sets them.

The set/used information also permits recognition of those local variables which are set and never used; these form a frequent source of inefficiencies, and may also be symptomatic of bugs.

Flow of Control

Lint attempts to detect unreachable portions of the programs which it processes. It will complain about unlabeled statements immediately following **goto**, **break**, **continue**, or **return** statements. An attempt is made to detect loops which can never be left at the bottom, detecting the special cases **while**(1) and **for**(;) as infinite loops. *Lint* also complains about loops which cannot be entered at the top; some valid programs may have such loops, but at best they are bad style, at worst bugs.

Lint has an important area of blindness in the flow of control algorithm: it has no way of detecting functions which are called and never return. Thus, a call to *exit* may cause unreachable code which *lint* does not detect; the most serious effects of this are in the determination of returned function values (see the next section).

One form of unreachable statement is not usually complained about by *lint*; a **break** statement that cannot be reached causes no message. Programs generated by *yacc*,² and especially *lex*,³ may have literally hundreds of unreachable **break** statements. The **-O** flag in the C compiler will often eliminate the resulting object code inefficiency. Thus, these unreached statements are of little importance, there is typically nothing the user can do about them, and the resulting messages would clutter up the *lint* output. If these messages are desired, *lint* can be invoked with the **-b** option.

Function Values

Sometimes functions return values which are never used; sometimes programs incorrectly use function "values" which have never been returned. *Lint* addresses this problem in a number of ways.

Locally, within a function definition, the appearance of both

```
return( expr );
```

and

```
return ;
```

statements is cause for alarm; *lint* will give the message

```
function name contains return(e) and return
```

The most serious difficulty with this is detecting when a function return is implied by flow of control reaching the end of the function. This can be seen with a simple example:

```
f ( a ) {
    if ( a ) return ( 3 );
    g ();
}
```

Notice that, if *a* tests false, *f* will call *g* and then return with no defined return value; this will trigger a complaint from *lint*. If *g*, like *exit*, never returns, the message will still be produced when in fact nothing is wrong.

In practice, some potentially serious bugs have been discovered by this feature; it also accounts for a substantial fraction of the “noise” messages produced by *lint*.

On a global scale, *lint* detects cases where a function returns a value, but this value is sometimes, or always, unused. When the value is always unused, it may constitute an inefficiency in the function definition. When the value is sometimes unused, it may represent bad style (e.g., not testing for error conditions).

The dual problem, using a function value when the function does not return one, is also detected. This is a serious problem. Amazingly, this bug has been observed on a couple of occasions in “working” programs; the desired function value just happened to have been computed in the function return register!

Type Checking

Lint enforces the type checking rules of C more strictly than the compilers do. The additional checking is in four major areas: across certain binary operators and implied assignments, at the structure selection operators, between the definition and uses of functions, and in the use of enumerations.

There are a number of operators which have an implied balancing between types of the operands. The assignment, conditional (?:), and relational operators have this property; the argument of a **return** statement, and expressions used in initialization also suffer similar conversions. In these operations, **char**, **short**, **int**, **long**, **unsigned**, **float**, and **double** types may be freely intermixed. The types of pointers must agree exactly, except that arrays of *x*'s can, of course, be intermixed with pointers to *x*'s.

The type checking rules also require that, in structure references, the left operand of the \rightarrow be a pointer to structure, the left operand of the \cdot be a structure, and the right operand of these operators be a member of the structure implied by the left operand. Similar checking is done for references to unions.

Strict rules apply to function argument and return value matching. The types **float** and **double** may be freely matched, as may the types **char**, **short**, **int**, and **unsigned**. Also, pointers can be matched with the associated arrays. Aside from this, all actual arguments must agree in type with their declared counterparts.

With enumerations, checks are made that enumeration variables or members are not mixed with other types, or other enumerations, and that the only operations applied are =, initialization, ==, !=, and function arguments and return values.

Type Casts

The type cast feature in C was introduced largely as an aid to producing more portable programs. Consider the assignment

```
p = 1 ;
```

where *p* is a character pointer. *Lint* will quite rightly complain. Now, consider the assignment

```
p = (char *)1 ;
```

in which a cast has been used to convert the integer to a character pointer. The programmer obviously had a strong motivation for doing this, and has clearly signaled his intentions. It seems harsh for *lint* to continue to complain about this. On the other hand, if this code is moved to another machine, such code should be looked at carefully. The $-c$ flag controls the printing of comments about casts. When $-c$ is in effect, casts are treated as though they were assignments subject to complaint; otherwise, all legal casts are passed without comment, no matter how strange the type mixing seems to be.

Non-portable Character Use

On the PDP-11, characters are signed quantities, with a range from -128 to 127 . On most of the other C implementations, characters take on only positive values. Thus, *lint* will flag certain comparisons and assignments as being illegal or non-portable. For example, the fragment

```
char c;
...
if( (c = getchar()) < 0 ) ...
```

works on the PDP-11, but will fail on machines where characters always take on positive values. The real solution is to declare *c* an integer, since *getchar* is actually returning integer values. In any case, *lint* will say "non-portable character comparison".

A similar issue arises with bit-fields; when assignments of constant values are made to bit-fields, the field may be too small to hold the value. This is especially true because on some machines bit-fields are considered as signed quantities. While it may seem unintuitive to consider that a two bit field declared of type **int** cannot hold the value 3, the problem disappears if the bit-field is declared to have type **unsigned**.

Assignments of longs to ints

Bugs may arise from the assignment of **long** to an **int**, which loses accuracy. This may happen in programs which have been incompletely converted to use **typedefs**. When a **typedef** variable is changed from **int** to **long**, the program can stop working because some intermediate results may be assigned to **ints**, losing accuracy. Since there are a number of legitimate reasons for assigning **longs** to **ints**, the detection of these assignments is enabled by the **-a** flag.

Strange Constructions

Several perfectly legal, but somewhat strange, constructions are flagged by *lint*; the messages hopefully encourage better code quality, clearer style, and may even point out bugs. The **-h** flag is used to enable these checks. For example, in the statement

```
*p++ ;
```

the ***** does nothing; this provokes the message "null effect" from *lint*. The program fragment

```
unsigned x ;
if( x < 0 ) ...
```

is clearly somewhat strange; the test will never succeed. Similarly, the test

```
if( x > 0 ) ...
```

is equivalent to

```
if( x != 0 )
```

which may not be the intended action. *Lint* will say "degenerate unsigned comparison" in these cases. If one says

```
if( 1 != 0 ) ...
```

lint will report "constant in conditional context", since the comparison of 1 with 0 gives a constant result.

Another construction detected by *lint* involves operator precedence. Bugs which arise from misunderstandings about the precedence of operators can be accentuated by spacing and formatting, making such bugs extremely hard to find. For example, the statements

```
if( x&077 == 0 ) ...
```

or

```
x << 2 + 40
```

probably do not do what was intended. The best solution is to parenthesize such expressions, and *lint* encourages this by an appropriate message.

Finally, when the `-h` flag is in force *lint* complains about variables which are redeclared in inner blocks in a way that conflicts with their use in outer blocks. This is legal, but is considered by many (including the author) to be bad style, usually unnecessary, and frequently a bug.

Ancient History

There are several forms of older syntax which are being officially discouraged. These fall into two classes, assignment operators and initialization.

The older forms of assignment operators (e.g., `=+`, `=-`, `...`) could cause ambiguous expressions, such as

```
a = - 1 ;
```

which could be taken as either

```
a = - 1 ;
```

or

```
a = -1 ;
```

The situation is especially perplexing if this kind of ambiguity arises as the result of a macro substitution. The newer, and preferred operators (`+=`, `-=`, etc.) have no such ambiguities. To spur the abandonment of the older forms, *lint* complains about these old fashioned operators.

A similar issue arises with initialization. The older language allowed

```
int x 1 ;
```

to initialize `x` to 1. This also caused syntactic difficulties: for example,

```
int x ( -1 ) ;
```

looks somewhat like the beginning of a function declaration:

```
int x ( y ) { ...
```

and the compiler must read a fair ways past `x` in order to sure what the declaration really is.. Again, the problem is even more perplexing when the initializer involves a macro. The current syntax places an equals sign between the variable and the initializer:

```
int x = -1 ;
```

This is free of any possible syntactic ambiguity.

Pointer Alignment

Certain pointer assignments may be reasonable on some machines, and illegal on others, due entirely to alignment restrictions. For example, on the PDP-11, it is reasonable to assign integer pointers to double pointers, since double precision values may begin on any integer boundary. On the Honeywell 6000, double precision values must begin on even word boundaries; thus, not all such assignments make sense. *Lint* tries to detect cases where pointers are assigned to other pointers, and such alignment problems might arise. The message "possible pointer alignment problem" results from this situation whenever either the `-p` or `-h` flags are in effect.

Multiple Uses and Side Effects

In complicated expressions, the best order in which to evaluate subexpressions may be highly machine dependent. For example, on machines (like the PDP-11) in which the stack runs backwards, function arguments will probably be best evaluated from right-to-left; on machines with a stack running forward, left-to-right seems most attractive. Function calls embedded as arguments of other functions may or may not be treated similarly to ordinary arguments. Similar issues arise with other operators which have side effects, such as the assignment operators and the increment and decrement operators.

In order that the efficiency of C on a particular machine not be unduly compromised, the C language leaves the order of evaluation of complicated expressions up to the local compiler, and, in fact, the various C compilers have considerable differences in the order in which they will evaluate complicated expressions. In particular, if any variable is changed by a side effect, and also used elsewhere in the same expression, the result is explicitly undefined.

Lint checks for the important special case where a simple scalar variable is affected. For example, the statement

```
a[i] = b[i++];
```

will draw the complaint:

```
warning: i evaluation order undefined
```

Implementation

Lint consists of two programs and a driver. The first program is a version of the Portable C Compiler^{4,5} which is the basis of the IBM 370, Honeywell 6000, and Interdata 8/32 C compilers. This compiler does lexical and syntax analysis on the input text, constructs and maintains symbol tables, and builds trees for expressions. Instead of writing an intermediate file which is passed to a code generator, as the other compilers do, *lint* produces an intermediate file which consists of lines of ascii text. Each line contains an external variable name, an encoding of the context in which it was seen (use, definition, declaration, etc.), a type specifier, and a source file name and line number. The information about variables local to a function or file is collected by accessing the symbol table, and examining the expression trees.

Comments about local problems are produced as detected. The information about external names is collected onto an intermediate file. After all the source files and library descriptions have been collected, the intermediate file is sorted to bring all information collected about a given external name together. The second, rather small, program then reads the lines from the intermediate file and compares all of the definitions, declarations, and uses for consistency.

The driver controls this process, and is also responsible for making the options available to both passes of *lint*.

Portability

C on the Honeywell and IBM systems is used, in part, to write system code for the host operating system. This means that the implementation of C tends to follow local conventions rather than adhere strictly to UNIX† system conventions. Despite these differences, many C programs have been successfully moved to GCOS and the various IBM installations with little effort. This section describes some of the differences between the implementations, and discusses the *lint* features which encourage portability.

Uninitialized external variables are treated differently in different implementations of C. Suppose two files both contain a declaration without initialization, such as

† UNIX is a trademark of Bell Laboratories.

```
int a ;
```

outside of any function. The UNIX loader will resolve these declarations, and cause only a single word of storage to be set aside for *a*. Under the GCOS and IBM implementations, this is not feasible (for various stupid reasons!) so each such declaration causes a word of storage to be set aside and called *a*. When loading or library editing takes place, this causes fatal conflicts which prevent the proper operation of the program. If *lint* is invoked with the `-p` flag, it will detect such multiple definitions.

A related difficulty comes from the amount of information retained about external names during the loading process. On the UNIX system, externally known names have seven significant characters, with the upper/lower case distinction kept. On the IBM systems, there are eight significant characters, but the case distinction is lost. On GCOS, there are only six characters, of a single case. This leads to situations where programs run on the UNIX system, but encounter loader problems on the IBM or GCOS systems. *Lint* `-p` causes all external symbols to be mapped to one case and truncated to six characters, providing a worst-case analysis.

A number of differences arise in the area of character handling: characters in the UNIX system are eight bit ascii, while they are eight bit ebcidic on the IBM, and nine bit ascii on GCOS. Moreover, character strings go from high to low bit positions ("left to right") on GCOS and IBM, and low to high ("right to left") on the PDP-11. This means that code attempting to construct strings out of character constants, or attempting to use characters as indices into arrays, must be looked at with great suspicion. *Lint* is of little help here, except to flag multi-character character constants.

Of course, the word sizes are different! This causes less trouble than might be expected, at least when moving from the UNIX system (16 bit words) to the IBM (32 bits) or GCOS (36 bits). The main problems are likely to arise in shifting or masking. C now supports a bit-field facility, which can be used to write much of this code in a reasonably portable way. Frequently, portability of such code can be enhanced by slight rearrangements in coding style. Many of the incompatibilities seem to have the flavor of writing

```
x &= 0177700 ;
```

to clear the low order six bits of *x*. This suffices on the PDP-11, but fails badly on GCOS and IBM. If the bit field feature cannot be used, the same effect can be obtained by writing

```
x &= ~ 077 ;
```

which will work on all these machines.

The right shift operator is arithmetic shift on the PDP-11, and logical shift on most other machines. To obtain a logical shift on all machines, the left operand can be typed **unsigned**. Characters are considered signed integers on the PDP-11, and unsigned on the other machines. This persistence of the sign bit may be reasonably considered a bug in the PDP-11 hardware which has infiltrated itself into the C language. If there were a good way to discover the programs which would be affected, C could be changed; in any case, *lint* is no help here.

The above discussion may have made the problem of portability seem bigger than it in fact is. The issues involved here are rarely subtle or mysterious, at least to the implementor of the program, although they can involve some work to straighten out. The most serious bar to the portability of UNIX system utilities has been the inability to mimic essential UNIX system functions on the other systems. The inability to seek to a random character position in a text file, or to establish a pipe between processes, has involved far more rewriting and debugging than any of the differences in C compilers. On the other hand, *lint* has been very helpful in moving the UNIX operating system and associated utility programs to other machines.

Shutting up Lint

There are occasions when the programmer is smarter than *lint*. There may be valid reasons for "illegal" type casts, functions with a variable number of arguments, etc. Moreover, as specified above, the flow of control information produced by *lint* often has blind spots, causing occasional spurious messages about perfectly reasonable programs. Thus, some way of communicating with *lint*, typically to shut it up, is desirable.

The form which this mechanism should take is not at all clear. New keywords would require current and old compilers to recognize these keywords, if only to ignore them. This has both philosophical and practical problems. New preprocessor syntax suffers from similar problems. What was finally done was to cause a number of words to be recognized by *lint* when they were embedded in comments. This required minimal preprocessor changes; the preprocessor just had to agree to pass comments through to its output, instead of deleting them as had been previously done. Thus, *lint* directives are invisible to the compilers, and the effect on systems with the older preprocessors is merely that the *lint* directives don't work.

The first directive is concerned with flow of control information; if a particular place in the program cannot be reached, but this is not apparent to *lint*, this can be asserted by the directive

```
/* NOTREACHED */
```

at the appropriate spot in the program. Similarly, if it is desired to turn off strict type checking for the next expression, the directive

```
/* NOSTRICT */
```

can be used; the situation reverts to the previous default after the next expression. The `-v` flag can be turned on for one function by the directive

```
/* ARGSUSED */
```

Complaints about variable number of arguments in calls to a function can be turned off by the directive

```
/* VARARGS */
```

preceding the function definition. In some cases, it is desirable to check the first several arguments, and leave the later arguments unchecked. This can be done by following the `VARARGS` keyword immediately with a digit giving the number of arguments which should be checked; thus,

```
/* VARARGS2 */
```

will cause the first two arguments to be checked, the others unchecked. Finally, the directive

```
/* LINTLIBRARY */
```

at the head of a file identifies this file as a library declaration file; this topic is worth a section by itself.

Library Declaration Files

Lint accepts certain library directives, such as

```
-ly
```

and tests the source files for compatibility with these libraries. This is done by accessing library description files whose names are constructed from the library directives. These files all begin with the directive

```
/* LINTLIBRARY */
```

which is followed by a series of dummy function definitions. The critical parts of these

definitions are the declaration of the function return type, whether the dummy function returns a value, and the number and types of arguments to the function. The VARARGS and ARGSUSED directives can be used to specify features of the library functions.

Lint library files are processed almost exactly like ordinary source files. The only difference is that functions which are defined on a library file, but are not used on a source file, draw no complaints. *Lint* does not simulate a full library search algorithm, and complains if the source files contain a redefinition of a library routine (this is a feature!).

By default, *lint* checks the programs it is given against a standard library file, which contains descriptions of the programs which are normally loaded when a C program is run. When the `-p` flag is in effect, another file is checked containing descriptions of the standard I/O library routines which are expected to be portable across various machines. The `-n` flag can be used to suppress all library checking.

Bugs, etc.

Lint was a difficult program to write, partially because it is closely connected with matters of programming style, and partially because users usually don't notice bugs which cause *lint* to miss errors which it should have caught. (By contrast, if *lint* incorrectly complains about something that is correct, the programmer reports that immediately!)

Several areas remain to be further developed. The checking of structures and arrays is rather inadequate; size incompatibilities go unchecked; no attempt is made to match structure and union declarations across files. Some stricter checking of the use of `typedef` is clearly desirable, but what checking is appropriate, and how to carry it out, is still to be determined.

Lint shares the preprocessor with the C compiler. At some point it may be appropriate for a special version of the preprocessor to be constructed which checks for things such as unused macro definitions, macro arguments which have side effects which are not expanded at all, or are expanded more than once, etc.

The central problem with *lint* is the packaging of the information which it collects. There are many options which serve only to turn off, or slightly modify, certain features. There are pressures to add even more of these options.

In conclusion, it appears that the general notion of having two programs is a good one. The compiler concentrates on quickly and accurately turning the program text into bits which can be run; *lint* concentrates on issues of portability, style, and efficiency. *Lint* can afford to be wrong, since incorrectness and over-conservatism are merely annoying, not fatal. The compiler can be fast since it knows that *lint* will cover its flanks. Finally, the programmer can concentrate at one stage of the programming process solely on the algorithms, data structures, and correctness of the program, and then later retrofit, with the aid of *lint*, the desirable properties of universality and portability.

References

- [1] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey (1978).
- [2] S. C. Johnson. "YACC—Yet Another Compiler-Compiler," Bell Laboratories (July 1975).
- [3] M. E. Lesk. "LEX—A Lexical Analyzer Generator," Bell Laboratories (October 1975).
- [4] S. C. Johnson and D. M. Ritchie. "UNIX Time-Sharing System: Portability of C Programs and the UNIX System," *Bell Sys. Tech. J.* **57**(6), pp. 2021-2048 (1978).
- [5] S. C. Johnson. "A Portable Compiler: Theory and Practice," *Proc. 5th ACM Symp. on Principles of Programming Languages*, pp. 97-104 (January 1978).

Appendix: Current Lint Options

The *lint* command currently has the form

```
lint [-options] files ... library-descriptors ...
```

The version of *lint* included in UNIX Release 3.0 interprets some of its options differently than previous versions. The following options have their interpretation reversed in the new *lint*:

<i>Option</i>	<i>New Interpretation</i>
-a	Suppress complaints about assignments of long values to variables that are not long.
-b	Suppress complaints about break statements that cannot be reached.
-c	Suppress complaints about casts that have questionable portability.
-h	Do not apply heuristics (which attempt to intuit bugs, improve style, and reduce waste).
-x	Do not report variables referred to by external declarations, but never used.

The following options have the same interpretation as in the past:

<i>Option</i>	<i>Interpretation</i>
-n	Do not check for compatibility with either the standard or the portable <i>lint</i> library.
-p	Attempt to check portability to other dialects of C (IBM and Honeywell).
-u	Suppress complaints about functions and external variables used and not defined, or defined and not used.
-v	Suppress complaints about unused arguments in functions.

January 1981

A Portable FORTRAN 77 Compiler

S. I. Feldman

P. J. Weinberger

Bell Laboratories

Murray Hill, New Jersey 07974

ABSTRACT

The Fortran language has just been revised. The new language, known as Fortran 77, became an official American National Standard on April 3, 1978. We report here on a compiler and run-time system for the new extended language. This is believed to be the first complete Fortran 77 system to be implemented. This compiler is designed to be portable, to be correct and complete, and to generate code compatible with calling sequences produced by C compilers. In particular, this Fortran is quite usable on UNIX[†] systems. In this paper, we describe the language compiled, interfaces between procedures, and file formats assumed by the I/O system. An appendix describes the Fortran 77 language.

1. INTRODUCTION

Fortran 77 became an official American National Standard [1] on April 3, 1978. Fortran 77 supplants 1966 Standard Fortran [2]. We report here on a compiler and run-time system for the new extended language. The compiler and computation library were written by S. I. Feldman, the I/O system by P. J. Weinberger. We believe ours to be the first complete Fortran 77 system to be implemented. This compiler is designed to be portable to a number of different machines, to be correct and complete, and to generate code compatible with calling sequences produced by compilers for the C language [3]. In particular, it is in use on UNIX systems. Two families of C compilers are in use at Bell Laboratories, those based on D. M. Ritchie's PDP-11 compiler [4] and those based on S. C. Johnson's portable C compiler [5]. This Fortran compiler can drive the second passes of either family. In this paper, we describe the language compiled, interfaces between procedures, and file formats assumed by the I/O system. We will describe implementation details in companion papers.

1.1. Usage

At present, versions of the compiler run on and compile for the PDP-11, the VAX-11/780, and the Interdata 8/32 UNIX systems. The command to run the compiler is:

f77 flags file . . .

f77 is a general-purpose command for compiling and loading Fortran and Fortran-related files. EFL [6] and Ratfor [7] source files will be preprocessed before being presented to the Fortran compiler. C and assembler source files will be compiled by the appropriate programs. Object files will be loaded. (The *f77* and *cc* commands cause slightly different loading sequences to be generated, since Fortran programs need a few extra libraries and a different startup routine than do C programs.) The following file name suffixes are understood:

[†] UNIX is a trademark of Bell Laboratories.

.f	Fortran source file
.e	EFL source file
.r	Ratfor source file
.c	C source file
.s	Assembler source file
.o	Object file

The following flags are understood:

- S Generate assembler output for each source file, but do not assemble it. Assembler output for a source file *x.f*, *x.e*, *x.r*, or *x.c* is put on file *x.s*.
- c Compile but do not load. Output for *x.f*, *x.e*, *x.r*, *x.c*, or *x.s* is put on file *x.o*.
- m Apply the M4 macro preprocessor to each EFL or Ratfor source file before using the appropriate compiler.
- f Apply the EFL or Ratfor processor to all relevant files, and leave the output from *x.e* or *x.r* on *x.f*. Do not compile the resulting Fortran program.
- p Generate code to produce usage profiles.
- o *f* Put executable module on file *f*. (Default is **a.out**).
- w Suppress all warning messages.
- w66 Suppress warnings about Fortran 66 features used.
- O Invoke the C object code optimizer.
- C Compile code the checks that subscripts are within array bounds.
- onetrip Compile code that performs every **do** loop at least once; see Section 2.10.
- U Do not convert upper case letters to lower case. The default is to convert Fortran programs to lower case.
- u Make the default type of a variable **undefined**; see Section 2.3.
- I2 On machines which support short integers, make the default integer constants and variables short. (**-I4** is the standard value of this option; see Section 2.14.) All logical quantities will be short.
- E The remaining characters in the argument are used as an EFL flag argument.
- R The remaining characters in the argument are used as a Ratfor flag argument.
- F Ratfor and and EFL source programs are pre-processed into Fortran files, but those files are not compiled or removed.

Other flags, all library names (arguments beginning **-l**), and any names not ending with one of the understood suffixes are passed to the loader.

1.2. Documentation Conventions

In running text, we write Fortran keywords and other literal strings in boldface lower case. Examples will be presented in lightface lower case. Names representing a class of values will be printed in italics.

1.3. Implementation Strategy

The compiler and library are written entirely in C. The compiler generates C compiler intermediate code. Since there are C compilers running on a variety of machines, relatively small changes will make this Fortran compiler generate code for any of them. Furthermore, this approach guarantees that the resulting programs are compatible with C usage. The runtime computational library is complete. The mathematical functions are computed to at least 63 bit precision. The runtime I/O library makes use of D. M. Ritchie's Standard C I/O package [8] for transferring data. With the few exceptions described below, only documented calls are used, so it should be relatively easy to modify to run on other operating systems.

2. LANGUAGE EXTENSIONS

Fortran 77 includes almost all of Fortran 66 as a subset. We describe the differences briefly in the Appendix. The most important additions are a character string data type, file-oriented input/output statements, and random access I/O. Also, the language has been cleaned up considerably.

In addition to implementing the language specified in the new Standard, our compiler implements a few extensions described in this section. Most are useful additions to the language. The remainder are extensions to make it easier to communicate with C procedures or to permit compilation of old (1966 Standard) programs.

2.1. Double Complex Data Type

The new type **double complex** is defined. Each datum is represented by a pair of double precision real variables. A double complex version of every **complex** built-in function is provided. The specific function names begin with **z** instead of **c**.

2.2. Internal Files

The Fortran 77 standard introduces "internal files" (memory arrays), but restricts their use to formatted sequential I/O statements. Our I/O system also permits internal files to be used in direct and unformatted reads and writes.

2.3. Implicit Undefined statement

Fortran 66 has a fixed rule that the type of a variable that does not appear in a type statement is **integer** if its first letter is **i, j, k, l, m** or **n**, and **real** otherwise. Fortran 77 has an **implicit** statement for overriding this rule. As an aid to good programming practice, we permit an additional type, **undefined**. The statement

```
implicit undefined(a-z)
```

turns off the automatic data typing mechanism, and the compiler will issue a diagnostic for each variable that is used but does not appear in a type statement. Specifying the **-u** compiler flag is equivalent to beginning each procedure with this statement.

2.4. Recursion

Procedures may call themselves, directly or through a chain of other procedures.

2.5. Automatic Storage

Two new keywords are recognized, **static** and **automatic**. These keywords may appear as "types" in type statements and in **implicit** statements. Local variables are **static** by default; there is exactly one copy of the datum, and its value is retained between calls. There is one copy of each variable declared **automatic** for each invocation of the procedure. Automatic variables may not appear in **equivalence**, **data**, or **save** statements.

2.6. Source Input Format

The Standard expects input to the compiler to be in 72 column format: except in comment lines, the first five characters are the statement number, the next is the continuation character, and the next sixty-six are the body of the line. (If there are fewer than seventy-two characters on a line, the compiler pads it with blanks; characters after the seventy-second are ignored).

In order to make it easier to type Fortran programs, our compiler also accepts input in variable length lines. An ampersand ("&") in the first position of a line indicates a continuation line; the remaining characters form the body of the line. A tab character in one of the first six positions of a line signals the end of the statement number and continuation part of the line; the remaining characters form the body of the line. A tab elsewhere on the line is treated as another kind of blank by the compiler.

In the Standard, there are only 26 letters — Fortran is a one-case language. Consistent with ordinary UNIX system usage, our compiler expects lower case input. By default, the compiler converts all upper case characters to lower case except those inside character constants. However, if the `-U` compiler flag is specified, upper case letters are not transformed. In this mode, it is possible to specify external names with upper case letters in them, and to have distinct variables differing only in case. Regardless of the setting of the flag, keywords will only be recognized in lower case.

2.7. Include Statement

The statement

```
include 'stuff'
```

is replaced by the contents of the file `stuff`. `includes` may be nested to a reasonable depth, currently ten.

2.8. Binary Initialization Constants

A **logical**, **real**, or **integer** variable may be initialized in a **data** statement by a binary constant, denoted by a letter followed by a quoted string. If the letter is **b**, the string is binary, and only zeroes and ones are permitted. If the letter is **o**, the string is octal, with digits **0–7**. If the letter is **z** or **x**, the string is hexadecimal, with digits **0–9**, **a–f**. Thus, the statements

```
integer a(3)
data a / b'1010', o'12', z'a' /
```

initialize all three elements of **a** to ten.

2.9. Character Strings

For compatibility with C usage, the following backslash escapes are recognized:

<code>\n</code>	new-line
<code>\t</code>	tab
<code>\b</code>	backspace
<code>\f</code>	form feed
<code>\0</code>	null
<code>\'</code>	apostrophe (does not terminate a string)
<code>\"</code>	quotation mark (does not terminate a string)
<code>\\</code>	<code>\</code>
<code>\x</code>	<code>x</code> , where <code>x</code> is any other character

Fortran 77 only has one quoting character, the apostrophe. Our compiler and I/O system recognize both the apostrophe (`'`) and the double-quote (`"`). If a string begins with one variety of quote mark, the other may be embedded within it without using the repeated quote or backslash escapes.

Every unequivalenced scalar local character variable and every character string constant is aligned on an **integer** word boundary. Each character string constant appearing outside a **data** statement is followed by a null character to ease communication with C routines.

2.10. Hollerith

Fortran 77 does not have the old Hollerith (n h) notation, though the new Standard recommends implementing the old Hollerith feature in order to improve compatibility with old programs. In our compiler, Hollerith data may be used in place of character string constants, and may also be used to initialize non-character variables in **data** statements.

2.11. Equivalence Statements

As a very special and peculiar case, Fortran 66 permits an element of a multiply-dimensioned array to be represented by a singly-subscripted reference in **equivalence** statements. Fortran 77 does not permit this usage, since subscript lower bounds may now be different from 1. Our compiler permits single subscripts in **equivalence** statements, under the interpretation that all missing subscripts are equal to 1. A warning message is printed for each such incomplete subscript.

2.12. One-Trip DO Loops

The Fortran 77 Standard requires that the range of a **do** loop not be performed if the initial value is already past the limit value, as in

```
do 10 i = 2, 1
```

The 1966 Standard stated that the effect of such a statement was undefined, but it was common practice that the range of a **do** loop would be performed at least once. In order to accommodate old programs, though they were in violation of the 1966 Standard, the **-onetrip** compiler flag causes non-standard loops to be generated.

2.13. Commas in Formatted Input

The I/O system attempts to be more lenient than the Standard when it seems worthwhile. When doing a formatted read of non-character variables, commas may be used as value separators in the input record, overriding the field lengths given in the format statement. Thus, the format

```
(i10, f20.10, i4)
```

will read the record

```
-345,.05e-3,12
```

correctly.

2.14. Short Integers

On machines that support halfword integers, the compiler accepts declarations of type **integer*2**. (Ordinary integers follow the Fortran rules about occupying the same space as a REAL variable; they are assumed to be of C type **long int**; halfword integers are of C type **short int**.) An expression involving only objects of type **integer*2** is of that type. Generic functions return short or long integers depending on the actual types of their arguments. If a procedure is compiled using the **-I2** flag, all small integer constants will be of type **integer*2**. If the precision of an integer-valued intrinsic function is not determined by the generic function rules, one will be chosen that returns the prevailing length (**integer*2** when the **-I2** command flag is in effect). When the **-I2** option is in effect, all quantities of type **logical** will be short. Note that these short integer and logical quantities do not obey the standard rules for storage association.

2.15. Additional Intrinsic Functions

This compiler supports all of the intrinsic functions specified in the Fortran 77 Standard. In addition, there are functions for performing bitwise Boolean operations (**or**, **and**, **xor**, and **not**) and for accessing the UNIX command arguments (**getarg** and **iargc**).

3. VIOLATIONS OF THE STANDARD

We know only three ways in which our Fortran system violates the new standard:

3.1. Double Precision Alignment

The Fortran standards (both 1966 and 1977) permit **common** or **equivalence** statements to force a double precision quantity onto an odd word boundary, as in the following example:

```
real a(4)
double precision b,c
equivalence (a(1),b), (a(4),c)
```

Some machines (e.g., Honeywell 6000, IBM 360) require that double precision quantities be on double word boundaries; other machines (e.g., IBM 370), run inefficiently if this alignment rule is not observed. It is possible to tell which equivalenced and common variables suffer from a forced odd alignment, but every double precision argument would have to be assumed on a bad boundary. To load such a quantity on some machines, it would be necessary to use separate operations to move the upper and lower halves into the halves of an aligned temporary, then to load that double precision temporary; the reverse would be needed to store a result. We have chosen to require that all double precision real and complex quantities fall on even word boundaries on machines with corresponding hardware requirements, and to issue a diagnostic if the source code demands a violation of the rule.

3.2. Dummy Procedure Arguments

If any argument of a procedure is of type character, all dummy procedure arguments of that procedure must be declared in an **external** statement. This requirement arises as a subtle corollary of the way we represent character string arguments and of the one-pass nature of the compiler. A warning is printed if a dummy procedure is not declared **external**. Code is correct if there are no **character** arguments.

3.3. T and TL Formats

The implementation of the **t** (absolute tab) and **tl** (leftward tab) format codes is defective. These codes allow rereading or rewriting part of the record which has already been processed. (Section 6.3.2 in the Appendix.) The implementation uses seeks, so if the unit is not one which allows seeks, such as a terminal, the program is in error. (People who can make a case for using **tl** should let us know.) A benefit of the implementation chosen is that there is no upper limit on the length of a record, nor is it necessary to predeclare any record lengths except where specifically required by Fortran or the operating system.

4. INTER-PROCEDURE INTERFACE

To be able to write C procedures that call or are called by Fortran procedures, it is necessary to know the conventions for procedure names, data representation, return values, and argument lists that the compiled code obeys.

4.1. Procedure Names

On UNIX systems, the name of a common block or a Fortran procedure has an underscore appended to it by the compiler to distinguish it from a C procedure or external variable with the same user-assigned name. Fortran library procedure names have embedded underscores to avoid clashes with user-assigned subroutine names.

4.2. Data Representations

The following is a table of corresponding Fortran and C declarations:

Fortran	C
integer*2 x	short int x;
integer x	long int x;
logical x	long int x;
real x	float x;
double precision x	double x;
complex x	struct { float r, i; } x;
double complex x	struct { double dr, di; } x;
character*6 x	char x[6];

(By the rules of Fortran, **integer**, **logical**, and **real** data occupy the same amount of memory).

4.3. Return Values

A function of type **integer**, **logical**, **real**, or **double precision** declared as a C function that returns the corresponding type. A **complex** or **double complex** function is equivalent to a C routine with an additional initial argument that points to the place where the return value is to be stored. Thus,

```
complex function f( . . . )
```

is equivalent to

```
f_(temp, . . . )
struct { float r, i; } *temp;
. . .
```

A character-valued function is equivalent to a C routine with two extra initial arguments: a data address and a length. Thus,

```
character*15 function g( . . . )
```

is equivalent to

```
g_(result, length, . . . )
char result[ ];
long int length;
. . .
```

and could be invoked in C by

```
char chars[15];
. . .
g_(chars, 15L, . . . );
```

Subroutines are invoked as if they were **integer**-valued functions whose value specifies which alternate return to use. Alternate return arguments (statement labels) are not passed to the function, but are used to do an indexed branch in the calling procedure. (If the subroutine has no entry points with alternate return arguments, the returned value is undefined.) The statement:

```
call nret(*1, *2, *3)
```

is treated exactly as if it were the computed `goto`

```
goto (1, 2, 3), nret( )
```

4.4. Argument Lists

All Fortran arguments are passed by address. In addition, for every argument that is of type character or that is a dummy procedure, an argument giving the length of the value is passed. (The string lengths are **long int** quantities passed by value). The order of arguments is then:

```
Extra arguments for complex and character functions
Address for each datum or function
A long int for each character or procedure argument
```

Thus, the call in

```
external f
character*7 s
integer b(3)
...
call sam(f, b(2), s)
```

is equivalent to that in

```
int f();
char s[7];
long int b[3];
...
sam_(f, &b[1], s, 0L, 7L);
```

Note that the first element of a C array always has subscript zero, but Fortran arrays begin at 1 by default. Fortran arrays are stored in column-major order, C arrays are stored in row-major order.

5. FILE FORMATS

5.1. Structure of Fortran Files

Fortran requires four kinds of external files: sequential formatted and unformatted, and direct formatted and unformatted. On UNIX systems, these are all implemented as ordinary files which are assumed to have the proper internal structure.

Fortran I/O is based on "records". When a direct file is opened in a Fortran program, the record length of the records must be given, and this is used by the Fortran I/O system to make the file look as if it is made up of records of the given length. In the special case that the record length is given as 1, the files are not considered to be divided into records, but are treated as byte-addressable byte strings; that is, as ordinary UNIX file system files. (A read or write request on such a file keeps consuming bytes until satisfied, rather than being restricted to a single record.)

The peculiar requirements on sequential unformatted files make it unlikely that they will ever be read or written by any means except Fortran I/O statements. Each record is preceded and followed by an integer containing the record's length in bytes.

The Fortran I/O system breaks sequential formatted files into records while reading by using each new-line as a record separator. The result of reading off the end of a record is undefined according to the Standard. The I/O system is permissive and treats the record as being extended by blanks. On output, the I/O system will write a new-line at the end of each

record. It is also possible for programs to write new-lines for themselves. This is an error, but the only effect will be that the single record the user thought he wrote will be treated as more than one record when being read or backspaced over.

5.2. Portability Considerations

The Fortran I/O system uses only the facilities of the standard C I/O library, a widely available and fairly portable package, with the following two nonstandard features: The I/O system needs to know whether a file can be used for direct I/O, and whether or not it is possible to backspace. Both of these facilities are implemented using the `fseek` routine, so there is a routine `canseek` which determines if `fseek` will have the desired effect. Also, the `inquire` statement provides the user with the ability to find out if two files are the same, and to get the name of an already opened file in a form which would enable the program to reopen it. (The UNIX operating system implementation attempts to determine the full pathname.) Therefore there are two routines which depend on facilities of the operating system to provide these two services. In any case, the I/O system runs on the PDP-11, VAX-11/780, and Interdata 8/32 UNIX systems.

5.3. Pre-Connected Files and File Positions

Units 5, 6, and 0 are preconnected when the program starts. Unit 5 is connected to the standard input, unit 6 is connected to the standard output, and unit 0 is connected to the standard error unit. All are connected for sequential formatted I/O.

All the other units are also preconnected when execution begins. Unit n is connected to a file named `fort.n`. These files need not exist, nor will they be created unless their units are used without first executing an `open`. The default connection is for sequential formatted I/O.

The Standard does not specify where a file which has been explicitly `opened` for sequential I/O is initially positioned. In fact, the I/O system attempts to position the file at the end, so a `write` will append to the file and a `read` will result in an end-of-file indication. To position a file to its beginning, use a `rewind` statement. The preconnected units 0, 5, and 6 are positioned as they come from the program's parent process.

REFERENCES

1. *SIGPLAN Notices* 11,3 (1976), as amended in X3J3 internal documents through "/90.1".
2. *USA Standard FORTRAN, USAS X3.9-1966*, New York: United States of America Standards Institute, March 7, 1966. Clarified in *CACM* 12:289 (1969) and *CACM* 14:628 (1971).
3. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Englewood Cliffs: Prentice-Hall (1978).
4. D. M. Ritchie, private communication.
5. S. C. Johnson, "A Portable Compiler: Theory and Practice," *Proc. 5th ACM Symp. on Principles of Programming Languages* (Jan. 1978).
6. S. I. Feldman, "An Informal Description of EFL," internal memorandum, Bell Laboratories.
7. B. W. Kernighan, *RATFOR—A Preprocessor for a Rational Fortran*, Bell Laboratories (Jan. 1977).
8. D. M. Ritchie, private communication.

Appendix: DIFFERENCES BETWEEN FORTRAN 66 AND FORTRAN 77

The following is a very brief description of the differences between the 1966 [2] and the 1977 [1] Standard languages. We assume that the reader is familiar with Fortran 66. We do not pretend to be complete, precise, or unbiased, but plan to describe what we feel are the most important aspects of the new language. At present the only current information on the 1977 Standard is in publications of the X3J3 Subcommittee of the American National Standards Institute. The following information is from the "/92" document. This draft Standard is written in English rather than a meta-language, but it is forbidding and legalistic. No tutorials or textbooks are available yet.

1. FEATURES DELETED FROM FORTRAN 66

1.1. Hollerith

All notions of "Hollerith" (*nh*) as data have been officially removed, although our compiler, like almost all in the foreseeable future, will continue to support this archaism.

1.2. Extended Range

In Fortran 66, under a set of very restrictive and rarely-understood conditions, it is permissible to jump out of the range of a **do** loop, then jump back into it. Extended range has been removed in the Fortran 77 language. The restrictions are so special, and the implementation of extended range is so unreliable in many compilers, that this change really counts as no loss.

2. PROGRAM FORM

2.1. Blank Lines

Completely blank lines are now legal comment lines.

2.2. Program and Block Data Statements

A main program may now begin with a statement that gives that program an external name:

```
program work
```

Block data procedures may also have names.

```
block data stuff
```

There is now a rule that only *one* unnamed block data procedure may appear in a program. (This rule is not enforced by our system.) The Standard does not specify the effect of the program and block data names, but they are clearly intended to aid conventional loaders.

2.3. ENTRY Statement

Multiple entry points are now legal. Subroutine and function subprograms may have additional entry points, declared by an **entry** statement with an optional argument list.

```
entry extra(a, b, c)
```

Execution begins at the first statement following the **entry** line. All variable declarations must precede all executable statements in the procedure. If the procedure begins with a **subroutine** statement, all entry points are subroutine names. If it begins with a **function** statement, each entry is a function entry point, with type determined by the type declared for the entry name. If any entry is a character-valued function, then all entries must be. In a function, an entry name of the same type as that where control entered must be assigned a value. Arguments do not retain their values between calls. (The ancient trick

of calling one entry point with a large number of arguments to cause the procedure to "remember" the locations of those arguments, then invoking an entry with just a few arguments for later calculation, is still illegal. Furthermore, the trick doesn't work in our implementation, since arguments are not kept in static storage.)

2.4. DO Loops

do variables and range parameters may now be of integer, real, or double precision types. (The use of floating point **do** variables is very dangerous because of the possibility of unexpected roundoff, and we strongly recommend against their use). The action of the **do** statement is now defined for all values of the **do** parameters. The statement

```
do 10 i = 1, u, d
```

performs $\max(0, \lfloor (u-1)/d \rfloor + 1)$ iterations. The **do** variable has a predictable value when exiting a loop: the value at the time a **goto** or **return** terminates the loop; otherwise the value that failed the limit test.

2.5. Alternate Returns

In a **subroutine** or subroutine **entry** statement, some of the arguments may be noted by an asterisk, as in

```
subroutine s(a, *, b, *)
```

The meaning of the "alternate returns" is described in section 5.2 of the Appendix.

3. DECLARATIONS

3.1. CHARACTER Data Type

One of the biggest improvements to the language is the addition of a character-string data type. Local and common character variables must have a length denoted by a constant expression:

```
character*17 a, b(3,4)
character*(6+3) c
```

If the length is omitted entirely, it is assumed equal to 1. A character string argument may have a constant length, or the length may be declared to be the same as that of the corresponding actual argument at run time by a statement like

```
character*(*) a
```

(There is an intrinsic function **len** that returns the actual length of a character string). Character arrays and common blocks containing character variables must be packed: in an array of character variables, the first character of one element must follow the last character of the preceding element, without holes.

3.2. IMPLICIT Statement

The traditional implied declaration rules still hold: a variable whose name begins with **i**, **j**, **k**, **l**, **m**, or **n** is of type **integer**, other variables are of type **real**, unless otherwise declared. This general rule may be overridden with an **implicit** statement:

```
implicit real(a-c,g), complex(w-z), character*(17) (s)
```

declares that variables whose name begins with an **a**, **b**, **c**, or **g** are **real**, those beginning with **w**, **x**, **y**, or **z** are assumed **complex**, and so on. It is still poor practice to depend on implicit typing, but this statement is an industry standard.

3.3. PARAMETER Statement

It is now possible to give a constant a symbolic name, as in

```
parameter (x=17, y=x/3, pi=3.14159d0, s='hello')
```

The type of each parameter name is governed by the same implicit and explicit rules as for a variable. The right side of each equal sign must be a constant expression (an expression made up of constants, operators, and already defined parameters).

3.4. Array Declarations

Arrays may now have as many as seven dimensions. (Only three were permitted in 1966). The lower bound of each dimension may be declared to be other than 1 by using a colon. Furthermore, an adjustable array bound may be an integer expression involving constants, arguments, and variables in **common**.

```
real a(-5:3, 7, m:n), b(n+1:2*n)
```

The upper bound on the last dimension of an array argument may be denoted by an asterisk to indicate that the upper bound is not specified:

```
integer a(5, *), b(*), c(0:1, -2:*)
```

3.5. SAVE Statement

A poorly known rule of Fortran 66 is that local variables in a procedure do not necessarily retain their values between invocations of that procedure. At any instant in the execution of a program, if a common block is declared neither in the currently executing procedure nor in any of the procedures in the chain of callers, all of the variables in that common block also become undefined. (The only exceptions are variables that have been defined in a **data** statement and never changed). These rules permit overlay and stack implementations for the affected variables. Fortran 77 permits one to specify that certain variables and common blocks are to retain their values between invocations. The declaration

```
save a, /b/, c
```

leaves the values of the variables **a** and **c** and all of the contents of common block **b** unaffected by a return. The simple declaration

```
save
```

has this effect on all variables and common blocks in the procedure. A common block must be **saved** in every procedure in which it is declared if the desired effect is to occur.

3.6. INTRINSIC Statement

All of the functions specified in the Standard are in a single category, "intrinsic functions", rather than being divided into "intrinsic" and "basic external" functions. If an intrinsic function is to be passed to another procedure, it must be declared **intrinsic**. Declaring it **external** (as in Fortran 66) causes a function other than the built-in one to be passed.

4. EXPRESSIONS

4.1. Character Constants

Character string constants are marked by strings surrounded by apostrophes. If an apostrophe is to be included in a constant, it is repeated:

```
'abc'  
'ain''t'
```

There are no null (zero-length) character strings in Fortran 77. Our compiler has two different quotation marks, (') and ("). (See Section 2.9 in the main text.)

4.2. Concatenation

One new operator has been added, character string concatenation, marked by a double slash ("//"). The result of a concatenation is the string containing the characters of the left operand followed by the characters of the right operand. The strings

```
'ab' // 'cd'
'abcd'
```

are equal. The strings being concatenated must be of constant length in all concatenations that are not the right sides of assignments. (The only concatenation expressions in which a character string declared adjustable with a "(*)" modifier or a substring denotation with nonconstant position values may appear are the right sides of assignments).

4.3. Character String Assignment

The left and right sides of a character assignment may not share storage. (The assumed implementation of character assignment is to copy characters from the right to the left side.) If the left side is longer than the right, it is padded with blanks. If the left side is shorter than the right, trailing characters are discarded.

4.4. Substrings

It is possible to extract a substring of a character variable or character array element, using the colon notation:

$$a(i,j) (m:n)$$

is the string of $(n-m+1)$ characters beginning at the m^{th} character of the character array element a_{ij} . Results are undefined unless $m \leq n$. Substrings may be used on the left sides of assignments and as procedure actual arguments.

4.5. Exponentiation

It is now permissible to raise real quantities to complex powers, or complex quantities to real or complex powers. (The principal part of the logarithm is used). Also, multiple exponentiation is now defined:

$$a^{b^{c}} = a^{(b^{c})}$$

4.6. Relaxation of Restrictions

Mixed mode expressions are now permitted. (For instance, it is permissible to combine integer and complex quantities in an expression.)

Constant expressions are permitted where a constant is allowed, except in **data** statements. (A constant expression is made up of explicit constants and **parameters** and the Fortran operators, except for exponentiation to a floating-point power). An adjustable dimension may now be an integer expression involving constants, arguments, and variables in **B** common..

Subscripts may now be general integer expressions; the old $cv \pm c'$ rules have been removed. **do** loop bounds may be general integer, real, or double precision expressions. Computed **goto** expressions and I/O unit numbers may be general integer expressions.

5. EXECUTABLE STATEMENTS

5.1. IF-THEN-ELSE

At last, the if-then-else branching structure has been added to Fortran. It is called a "Block If". A Block If begins with a statement of the form

```
if ( . . . ) then
```

and ends with an

```
end if
```

statement. Two other new statements may appear in a Block If. There may be several

```
else if( . . ) then
```

statements, followed by at most one

```
else
```

statement. If the logical expression in the Block If statement is true, the statements following it up to the next **elseif**, **else**, or **endif** are executed. Otherwise, the next **elseif** statement in the group is executed. If none of the **elseif** conditions are true, control passes to the statements following the **else** statement, if any. (The **else** must follow all **elseif**s in a Block If. Of course, there may be Block Ifs embedded inside of other Block If structures). A case construct may be rendered

```
if (s .eq. 'ab') then
```

```
...
```

```
else if (s .eq. 'cd') then
```

```
...
```

```
else
```

```
...
```

```
end if
```

5.2. Alternate Returns

Some of the arguments of a subroutine call may be statement labels preceded by an asterisk, as in

```
call joe(j, *10, m, *2)
```

A **return** statement may have an integer expression, such as

```
return k
```

If the entry point has n alternate return (asterisk) arguments and if $1 \leq k \leq n$, the return is followed by a branch to the corresponding statement label; otherwise the usual return to the statement following the **call** is executed.

6. INPUT/OUTPUT

6.1. Format Variables

A format may be the value of a character expression (constant or otherwise), or be stored in a character array, as in

```
write(6, '(i5)') x
```

6.2. END=, ERR=, and IOSTAT= Clauses

A `read` or `write` statement may contain `end=`, `err=`, and `iostat=` clauses, as in

```
write(6, 101, err=20, iostat=a(4))
read(5, 101, err=20, end=30, iostat=x)
```

Here 5 and 6 are the *units* on which the I/O is done, 101 is the statement number of the associated format, 20 and 30 are statement numbers, and `a` and `x` are integers. If an error occurs during I/O, control returns to the program at statement 20. If the end of the file is reached, control returns to the program at statement 30. In any case, the variable referred to in the `iostat=` clause is given a value when the I/O statement finishes. (Yes, the value is assigned to the name on the right side of the equal sign.) This value is zero if all went well, negative for end of file, and some positive value for errors.

6.3. Formatted I/O

6.3.1. Character Constants

Character constants in formats are copied literally to the output. Character constants cannot be read into.

```
write(6, '(i2, " isn't ", i1)') 7, 4
```

produces

```
7 isn't 4
```

Here the format is the character constant

```
(i2, ' isn't ', i1)
```

and the character constant "isn't" is copied into the output.

6.3.2. Positional Editing Codes

`t`, `tl`, `tr`, and `x` codes control where the next character is in the record. `trn` or `nx` specifies that the next character is `n` to the right of the current position. `tl` specifies that the next character is `n` to the left of the current position, allowing parts of the record to be reconsidered. `tn` says that the next character is to be character number `n` in the record. (See section 3.4 in the main text.)

6.3.3. Colon

A colon in the format terminates the I/O operation if there are no more data items in the I/O list, otherwise it has no effect. In the fragment

```
x=('hello', :, " there", i4)
write(6, x) 12
write(6, x)
```

the first `write` statement prints **hello there 12**, while the second only prints **hello**.

6.3.4. Optional Plus Signs

According to the Standard, each implementation has the option of putting plus signs in front of non-negative numeric output. The `sp` format code may be used to make the optional plus signs actually appear for all subsequent items while the format is active. The `ss` format code guarantees that the I/O system will not insert the optional plus signs, and the `s` format code restores the default behavior of the I/O system. (Since we never put out optional plus signs, `ss` and `s` codes have the same effect in our implementation.)

6.3.5. Blanks on Input

Blanks in numeric input fields, other than leading blanks will be ignored following a **bn** code in a format statement, and will be treated as zeros following a **bz** code in a format statement. The default for a unit may be changed by using the **open** statement. (Blanks are ignored by default.)

6.3.6. Unrepresentable Values

The Standard requires that if a numeric item cannot be represented in the form required by a format code, the output field must be filled with asterisks. (We think this should have been an option.)

6.3.7. *Iw.m*

There is a new integer output code, *iw.m*. It is the same as *iw*, except that there will be at least *m* digits in the output field, including, if necessary, leading zeros. The case *iw.0* is special, in that if the value being printed is 0, the output field is entirely blank. *iw.1* is the same as *iw*.

6.3.8. Floating Point

On input, exponents may start with the letter **E**, **D**, **e**, or **d**. All have the same meaning. On output we always use **e**. The **e** and **d** format codes also have identical meanings. A leading zero before the decimal point in **e** output without a scale factor is optional with the implementation. (We do not print it.) There is a *gw.d* format code which is the same as *ew.d* and *fw.d* on input, but which chooses **f** or **e** formats for output depending on the size of the number and of *d*.

6.3.9. "A" Format Code

A codes are used for character values. **aw** use a field width of *w*, while a plain **a** uses the length of the character item.

6.4. Standard Units

There are default formatted input and output units. The statement

```
read 10, a, b
```

reads from the standard unit using format statement 10. The default unit may be explicitly specified by an asterisk, as in

```
read(*, 10) a,b
```

Similarly, the standard output units is specified by a **print** statement or an asterisk unit:

```
print 10
write(*, 10)
```

6.5. List-Directed Formatting

List-directed I/O is a kind of free form input for sequential I/O. It is invoked by using an asterisk as the format identifier, as in

```
read(6, *) a,b,c
```

On input, values are separated by strings of blanks and possibly a comma. Values, except for character strings, cannot contain blanks. End of record counts as a blank, except in character strings, where it is ignored. Complex constants are given as two real constants separated by a comma and enclosed in parentheses. A null input field, such as between two consecutive commas, means the corresponding variable in the I/O list is not changed. Values may be preceded by repetition counts, as in

```
4*(3.,2.) 2*, 4*'hello'
```

which stands for 4 complex constants, 2 null values, and 4 string constants.

For output, suitable formats are chosen for each item. The values of character strings are printed; they are not enclosed in quotes, so they cannot be read back using list-directed input.

6.6. Direct I/O

A file connected for direct access consists of a set of equal-sized records each of which is uniquely identified by a positive integer. The records may be written or read in any order, using direct access I/O statements.

Direct access **read** and **write** statements have an extra argument, **rec=**, which gives the record number to be read or written.

```
read(2, rec=13, err=20) (a(i), i=1, 203)
```

reads the thirteenth record into the array **a**.

The size of the records must be given by an **open** statement (see below). Direct access files may be connected for either formatted or unformatted I/O.

6.7. Internal Files

Internal files are character string objects, such as variables or substrings, or arrays of type character. In the former cases there is only a single record in the file, in the latter case each array element is a record. The Standard includes only sequential formatted I/O on internal files. (I/O is not a very precise term to use here, but internal files are dealt with using **read** and **write**). There is no list-directed I/O on internal files. Internal files are used by giving the name of the character object in place of the unit number, as in

```
character*80 x
read(5,"(a)") x
read(x,"(i3,i4)") n1,n2
```

which reads a card image into **x** and then reads two integers from the front of it. A sequential **read** or **write** always starts at the beginning of an internal file.

(We also support a compatible extension, direct I/O on internal files. This is like direct I/O on external files, except that the number of records in the file cannot be changed.)

6.8. OPEN, CLOSE, and INQUIRE Statements

These statements are used to connect and disconnect units and files, and to gather information about units and files.

6.8.1. OPEN

The **open** statement is used to connect a file with a unit, or to alter some properties of the connection. The following is a minimal example.

```
open(1, file='fort.junk')
```

open takes a variety of arguments with meanings described below.

unit= a small non-negative integer which is the unit to which the file is to be connected. We allow, at the time of this writing, 0 through 9. If this parameter is the first one in the **open** statement, the **unit**= can be omitted.

iostat= is the same as in **read** or **write**.

err= is the same as in **read** or **write**.

file= a character expression, which when stripped of trailing blanks, is the name of the file to be connected to the unit. The filename should not be given if the **status**=**scratch**.

status= one of **old**, **new**, **scratch**, or **unknown**. If this parameter is not given, **unknown** is assumed. If **scratch** is given, a temporary file will be created. Temporary files are destroyed at the end of execution. If **new** is given, the file will be created if it doesn't exist, or truncated if it does. The meaning of **unknown** is processor dependent; our system treats it as synonymous with **old**.

access= **sequential** or **direct**, depending on whether the file is to be opened for sequential or direct I/O.

form= **formatted** or **unformatted**.

recl= a positive integer specifying the record length of the direct access file being opened. We measure all record lengths in bytes. On UNIX systems a record length of 1 has the special meaning explained in section 5.1 of the text.

blank= **null** or **zero**. This parameter has meaning only for formatted I/O. The default value is **null**. **zero** means that blanks, other than leading blanks, in numeric input fields are to be treated as zeros.

Opening a new file on a unit which is already connected has the effect of first closing the old file.

6.8.2. CLOSE

close severs the connection between a unit and a file. The unit number must be given. The optional parameters are **iostat**= and **err**= with their usual meanings, and **status**= either **keep** or **delete**. Scratch files cannot be kept, otherwise **keep** is the default. **delete** means the file will be removed. A simple example is

```
close(3, err=17)
```

6.8.3. INQUIRE

The **inquire** statement gives information about a unit ("inquire by unit") or a file ("inquire by file"). Simple examples are:

```
inquire(unit=3, namexx)
inquire(file='junk', number=n, exist=1)
```

file= a character variable specifies the file the **inquire** is about. Trailing blanks in the file name are ignored.

unit= an integer variable specifies the unit the **inquire** is about. Exactly one of **file**= or **unit**= must be used.

iostat=, **err**= are as before.

exist= a logical variable. The logical variable is set to **.true.** if the file or unit exists and is set to **.false.** otherwise.

opened= a logical variable. The logical variable is set to **.true.** if the file is connected to a unit or if the unit is connected to a file, and it is set to **.false.** otherwise.

- number**= an integer variable to which is assigned the number of the unit connected to the file, if any.
- named**= a logical variable to which is assigned **.true.** if the file has a name, or **.false.** otherwise.
- name**= a character variable to which is assigned the name of the file (inquire by file) or the name of the file connected to the unit (inquire by unit). The name will be the full name of the file.
- access**= a character variable to which will be assigned the value **'sequential'** if the connection is for sequential I/O, **'direct'** if the connection is for direct I/O. The value becomes undefined if there is no connection.
- sequential**= a character variable to which is assigned the value **'yes'** if the file could be connected for sequential I/O, **'no'** if the file could not be connected for sequential I/O, and **'unknown'** if we can't tell.
- direct**= a character variable to which is assigned the value **'yes'** if the file could be connected for direct I/O, **'no'** if the file could not be connected for direct I/O, and **'unknown'** if we can't tell.
- form**= a character variable to which is assigned the value **'formatted'** if the file is connected for formatted I/O, or **'unformatted'** if the file is connected for unformatted I/O.
- formatted**= a character variable to which is assigned the value **'yes'** if the file could be connected for formatted I/O, **'no'** if the file could not be connected for formatted I/O, and **'unknown'** if we can't tell.
- unformatted**= a character variable to which is assigned the value **'yes'** if the file could be connected for unformatted I/O, **'no'** if the file could not be connected for unformatted I/O, and **'unknown'** if we can't tell.
- recl**= an integer variable to which is assigned the record length of the records in the file if the file is connected for direct access.
- nextrec**= an integer variable to which is assigned one more than the number of the the last record read from a file connected for direct access.
- blank**= a character variable to which is assigned the value **'null'** if null blank control is in effect for the file connected for formatted I/O, **'zero'** if blanks are being converted to zeros and the file is connected for formatted I/O.

The gentle reader will remember that the people who wrote the standard probably weren't thinking of his needs. Here is an example. The declarations are omitted.

```
open(1, file="/dev/console")
```

On a UNIX system this statement opens the console for formatted sequential I/O. An **inquire** statement for either unit 1 or file **"/dev/console"** would reveal that the file exists, is connected to unit 1, has a name, namely **"/dev/console"**, is opened for sequential I/O, could be connected for sequential I/O, could not be connected for direct I/O (can't seek), is connected for formatted I/O, could be connected for formatted I/O, could not be connected for unformatted I/O (can't seek), has neither a record length nor a next record number, and is ignoring blanks in numeric fields.

In the UNIX system environment, the only way to discover what permissions you have for a file is to open it and try to read and write it. The **err=** parameter will return system error numbers. The **inquire** statement does not give a way of determining permissions.

January 1981

RATFOR — A Preprocessor for a Rational FORTRAN

Brian W. Kernighan

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Although Fortran is not a pleasant language to use, it does have the advantages of universality and (usually) relative efficiency. The Ratfor language attempts to conceal the main deficiencies of Fortran while retaining its desirable qualities, by providing decent control flow statements:

- statement grouping
- **if-else** and **switch** for decision-making
- **while**, **for**, **do**, and **repeat-until** for looping
- **break** and **next** for controlling loop exits

and some "syntactic sugar":

- free form input (multiple statements/line, automatic continuation)
- unobtrusive comment convention
- translation of $>$, $>=$, etc., into $.GT.$, $.GE.$, etc.
- **return(expression)** statement for functions
- **define** statement for symbolic parameters
- **include** statement for including source files

Ratfor is implemented as a preprocessor which translates this language into Fortran.

Once the control flow and cosmetic deficiencies of Fortran are hidden, the resulting language is remarkably pleasant to use. Ratfor programs are markedly easier to write, and to read, and thus easier to debug, maintain and modify than their Fortran equivalents.

It is readily possible to write Ratfor programs which are portable to other environments. Ratfor is written in itself in this way, so it is also portable; versions of Ratfor are now running on at least two dozen different types of computers at over five hundred locations.

This paper discusses design criteria for a Fortran preprocessor, the Ratfor language and its implementation, and user experience.

1. INTRODUCTION

Most programmers will agree that Fortran is an unpleasant language to program in, yet there are many occasions when they are forced to use it. For example, Fortran is often the only language thoroughly supported on the local computer. Indeed, it is the closest thing to a

universal programming language currently available: with care it is possible to write large, truly portable Fortran programs [1]. Finally, Fortran is often the most "efficient" language available, particularly for programs requiring much computation.

But Fortran *is* unpleasant. Perhaps the worst deficiency is in the control flow statements — conditional branches and loops — which express the logic of the program. The conditional statements in Fortran are primitive. The Arithmetic IF forces the user into at least two statement numbers and two (implied) GOTO's; it leads to unintelligible code, and is eschewed by good programmers. The Logical IF is better, in that the test part can be stated clearly, but hopelessly restrictive because the statement that follows the IF can only be one Fortran statement (with some *further* restrictions!). And of course there can be no ELSE part to a Fortran IF: there is no way to specify an alternative action if the IF is not satisfied.

The Fortran DO restricts the user to going forward in an arithmetic progression. It is fine for "1 to N in steps of 1 (or 2 or ...)", but there is no direct way to go backwards, or even (in ANSI Fortran [2]) to go from 1 to N-1. And of course the DO is useless if one's problem doesn't map into an arithmetic progression.

The result of these failings is that Fortran programs must be written with numerous labels and branches. The resulting code is particularly difficult to read and understand, and thus hard to debug and modify.

When one is faced with an unpleasant language, a useful technique is to define a new language that overcomes the deficiencies, and to translate it into the unpleasant one with a preprocessor. This is the approach taken with Ratfor. (The preprocessor idea is of course not new, and preprocessors for Fortran are especially popular today. A recent listing [3] of preprocessors shows more than 50, of which at least half a dozen are widely available.)

2. LANGUAGE DESCRIPTION

Design

Ratfor attempts to retain the merits of Fortran (universality, portability, efficiency) while hiding the worst Fortran inadequacies. The language *is* Fortran except for two aspects. First, since control flow is central to any program, regardless of the specific application, the primary task of Ratfor is to conceal this part of Fortran from the user, by providing decent control flow structures. These structures are sufficient and comfortable for structured programming in the narrow sense of programming without GOTO's. Second, since the preprocessor must examine an entire program to translate the control structure, it is possible at the same time to clean up many of the "cosmetic" deficiencies of Fortran, and thus provide a language which is

easier and more pleasant to read and write.

Beyond these two aspects — control flow and cosmetics — Ratfor does nothing about the host of other weaknesses of Fortran. Although it would be straightforward to extend it to provide character strings, for example, they are not needed by everyone, and of course the preprocessor would be harder to implement. Throughout, the design principle which has determined what should be in Ratfor and what should not has been *Ratfor doesn't know any Fortran*. Any language feature which would require that Ratfor really understand Fortran has been omitted. We will return to this point in the section on implementation.

Even within the confines of control flow and cosmetics, we have attempted to be selective in what features to provide. The intent has been to provide a small set of the most useful constructs, rather than to throw in everything that has ever been thought useful by someone.

The rest of this section contains an informal description of the Ratfor language. The control flow aspects will be quite familiar to readers used to languages like Algol, PL/I, Pascal, etc., and the cosmetic changes are equally straightforward. We shall concentrate on showing what the language looks like.

Statement Grouping

Fortran provides no way to group statements together, short of making them into a subroutine. The standard construction "if a condition is true, do this group of things," for example,

```
if (x > 100)
  { call error("x>100"); err = 1; return }
```

cannot be written directly in Fortran. Instead a programmer is forced to translate this relatively clear thought into murky Fortran, by stating the negative condition and branching around the group of statements:

```
if (x .le. 100) goto 10
  call error(5hx>100)
  err = 1
  return
```

10 ...

When the program 'doesn't work, or when it must be modified, this must be translated back into a clearer form before one can be sure what it does.

Ratfor eliminates this error-prone and confusing back-and-forth translation; the first form *is* the way the computation is written in Ratfor. A group of statements can be treated as a unit by enclosing them in the braces { and }.

This is true throughout the language: wherever a single Ratfor statement can be used, there can be several enclosed in braces. (Braces seem clearer and less obtrusive than **begin** and **end** or **do** and **end**, and of course **do** and **end** already have Fortran meanings.)

Cosmetics contribute to the readability of code, and thus to its understandability. The character ">" is clearer than ".GT.", so Ratfor translates it appropriately, along with several other similar shorthands. Although many Fortran compilers permit character strings in quotes (like "x>100"), quotes are not allowed in ANSI Fortran, so Ratfor converts it into the right number of H's: computers count better than people do.

Ratfor is a free-form language: statements may appear anywhere on a line, and several may appear on one line if they are separated by semicolons. The example above could also be written as

```
if (x > 100) {
    call error("x>100")
    err = 1
    return
}
```

In this case, no semicolon is needed at the end of each line because Ratfor assumes there is one statement per line unless told otherwise.

Of course, if the statement that follows the **if** is a single statement (Ratfor or otherwise), no braces are needed:

```
if (y <= 0.0 & z <= 0.0)
    write(6, 20) y, z
```

No continuation need be indicated because the statement is clearly not finished on the first line. In general Ratfor continues lines when it seems obvious that they are not yet done. (The continuation convention is discussed in detail later.)

Although a free-form language permits wide latitude in formatting styles, it is wise to pick one that is readable, then stick to it. In particular, proper indentation is vital, to make the logical structure of the program obvious to the reader.

The "else" Clause

Ratfor provides an **else** statement to handle the construction "if a condition is true, do this thing, *otherwise* do that thing."

```
if (a <= b)
    { sw = 0; write(6, 1) a, b }
else
    { sw = 1; write(6, 1) b, a }
```

This writes out the smaller of **a** and **b**, then the

larger, and sets **sw** appropriately.

The Fortran equivalent of this code is circuitous indeed:

```
if (a .gt. b) goto 10
    sw = 0
    write(6, 1) a, b
    goto 20
10  sw = 1
    write(6, 1) b, a
20  ...
```

This is a mechanical translation; shorter forms exist, as they do for many similar situations. But all translations suffer from the same problem: since they are translations, they are less clear and understandable than code that is not a translation. To understand the Fortran version, one must scan the entire program to make sure that no other statement branches to statements 10 or 20 before one knows that indeed this is an **if-else** construction. With the Ratfor version, there is no question about how one gets to the parts of the statement. The **if-else** is a single unit, which can be read, understood, and ignored if not relevant. The program says what it means.

As before, if the statement following an **if** or an **else** is a single statement, no braces are needed:

```
if (a <= b)
    sw = 0
else
    sw = 1
```

The syntax of the **if** statement is

```
if (legal Fortran condition)
    Ratfor statement
else
    Ratfor statement
```

where the **else** part is optional. The *legal Fortran condition* is anything that can legally go into a Fortran Logical IF. Ratfor does not check this clause, since it does not know enough Fortran to know what is permitted. The *Ratfor statement* is any Ratfor or Fortran statement, or any collection of them in braces.

Nested if's

Since the statement that follows an **if** or an **else** can be any Ratfor statement, this leads immediately to the possibility of another **if** or **else**. As a useful example, consider this problem: the variable **f** is to be set to -1 if **x** is less than zero, to +1 if **x** is greater than 100, and to 0 otherwise. Then in Ratfor, we write

```

if (x < 0)
    f = -1
else if (x > 100)
    f = +1
else
    f = 0

```

Here the statement after the first **else** is another **if-else**. Logically it is just a single statement, although it is rather complicated.

This code says what it means. Any version written in straight Fortran will necessarily be indirect because Fortran does not let you say what you mean. And as always, clever shortcuts may turn out to be too clever to understand a year from now.

Following an **else** with an **if** is one way to write a multi-way branch in Ratfor. In general the structure

```

if (...)
    ---
else if (...)
    ---
else if (...)
    ---
...
else
    ---

```

provides a way to specify the choice of exactly one of several alternatives. (Ratfor also provides a **switch** statement which does the same job in certain special cases; in more general situations, we have to make do with spare parts.) The tests are laid out in sequence, and each one is followed by the code associated with it. Read down the list of decisions until one is found that is satisfied. The code associated with this condition is executed, and then the entire structure is finished. The trailing **else** part handles the "default" case, where none of the other conditions apply. If there is no default action, this final **else** part is omitted:

```

if (x < 0)
    x = 0
else if (x > 100)
    x = 100

```

if-else ambiguity

There is one thing to notice about complicated structures involving nested **if**'s and **else**'s. Consider

```

if (x > 0)
    if (y > 0)
        write(6, 1) x, y
    else
        write(6, 2) y

```

There are two **if**'s and only one **else**. Which **if** does the **else** go with?

This is a genuine ambiguity in Ratfor, as it is in many other programming languages. The ambiguity is resolved in Ratfor (as elsewhere) by saying that in such cases the **else** goes with the closest previous un-**else**'ed **if**. Thus in this case, the **else** goes with the inner **if**, as we have indicated by the indentation.

It is a wise practice to resolve such cases by explicit braces, just to make your intent clear. In the case above, we would write

```

if (x > 0) {
    if (y > 0)
        write(6, 1) x, y
    else
        write(6, 2) y
}

```

which does not change the meaning, but leaves no doubt in the reader's mind. If we want the other association, we *must* write

```

if (x > 0) {
    if (y > 0)
        write(6, 1) x, y
}
else
    write(6, 2) y

```

The "switch" Statement

The **switch** statement provides a clean way to express multi-way branches which branch on the value of some integer-valued expression. The syntax is

```

switch (expression) {
    case expr1 :
        statements
    case expr2, expr3 :
        statements
    ...
    default:
        statements
}

```

Each **case** is followed by a list of comma-separated integer expressions. The *expression* inside **switch** is compared against the case expressions *expr1*, *expr2*, and so on in turn until one matches, at which time the statements following that **case** are executed. If no cases match

expression, and there is a **default** section, the statements with it are done; if there is no **default**, nothing is done. In all situations, as soon as some block of statements is executed, the entire **switch** is exited immediately. (Readers familiar with C [4] should beware that this behavior is not the same as the C **switch**.)

The "do" Statement

The **do** statement in Ratfor is quite similar to the **DO** statement in Fortran, except that it uses no statement number. The statement number, after all, serves only to mark the end of the **DO**, and this can be done just as easily with braces. Thus

```
do i = 1, n {
    x(i) = 0.0
    y(i) = 0.0
    z(i) = 0.0
}
```

is the same as

```
do 10 i = 1, n
    x(i) = 0.0
    y(i) = 0.0
    z(i) = 0.0
10 continue
```

The syntax is:

```
do legal-Fortran-DO-text
    Ratfor statement
```

The part that follows the keyword **do** has to be something that can legally go into a Fortran **DO** statement. Thus if a local version of Fortran allows **DO** limits to be expressions (which is not currently permitted in ANSI Fortran), they can be used in a Ratfor **do**.

The *Ratfor statement* part will often be enclosed in braces, but as with the **if**, a single statement need not have braces around it. This code sets an array to zero:

```
do i = 1, n
    x(i) = 0.0
```

Slightly more complicated,

```
do i = 1, n
    do j = 1, n
        m(i, j) = 0
```

sets the entire array **m** to zero, and

```
do i = 1, n
    do j = 1, n
        if (i < j)
            m(i, j) = -1
        else if (i == j)
            m(i, j) = 0
        else
            m(i, j) = +1
```

sets the upper triangle of **m** to -1 , the diagonal to zero, and the lower triangle to $+1$. (The operator **==** is "equals", that is, "EQ.") In each case, the statement that follows the **do** is logically a *single* statement, even though complicated, and thus needs no braces.

"break" and "next"

Ratfor provides a statement for leaving a loop early, and one for beginning the next iteration. **break** causes an immediate exit from the **do**; in effect it is a branch to the statement *after* the **do**. **next** is a branch to the bottom of the loop, so it causes the next iteration to be done. For example, this code skips over negative values in an array:

```
do i = 1, n {
    if (x(i) < 0.0)
        next
    process positive element
}
```

break and **next** also work in the other Ratfor looping constructions that we will talk about in the next few sections.

break and **next** can be followed by an integer to indicate breaking or iterating that level of enclosing loop; thus

```
break 2
```

exits from two levels of enclosing loops, and **break 1** is equivalent to **break**. **next 2** iterates the second enclosing loop. (Realistically, multi-level **break**'s and **next**'s are not likely to be much used because they lead to code that is hard to understand and somewhat risky to change.)

The "while" Statement

One of the problems with the Fortran **DO** statement is that it generally insists upon being done once, regardless of its limits. If a loop begins

```
DO I = 2, 1
```

this will typically be done once with **I** set to 2, even though common sense would suggest that perhaps it shouldn't be. Of course a Ratfor **do** can easily be preceded by a test

```

if (j <= k)
  do i = j, k {
    -----
  }

```

but this has to be a conscious act, and is often overlooked by programmers.

A more serious problem with the DO statement is that it encourages that a program be written in terms of an arithmetic progression with small positive steps, even though that may not be the best way to write it. If code has to be contorted to fit the requirements imposed by the Fortran DO, it is that much harder to write and understand.

To overcome these difficulties, Ratfor provides a **while** statement, which is simply a loop: "while some condition is true, repeat this group of statements". It has no preconceptions about why one is looping. For example, this routine to compute $\sin(x)$ by the Maclaurin series combines two termination criteria.

```

real function sin(x, e)
  # returns sin(x) to accuracy e, by
  #  $\sin(x) = x - x**3/3! + x**5/5! - \dots$ 

  sin = x
  term = x

  i = 3
  while (abs(term)>e & i<100) {
    term = -term * x**2 / float(i*(i-1))
    sin = sin + term
    i = i + 2
  }

  return
end

```

Notice that if the routine is entered with **term** already smaller than **e**, the loop will be done *zero times*, that is, no attempt will be made to compute $x**3$ and thus a potential underflow is avoided. Since the test is made at the top of a **while** loop instead of the bottom, a special case disappears — the code works at one of its boundaries. (The test $i<100$ is the other boundary — making sure the routine stops after some maximum number of iterations.)

As an aside, a sharp character "**#**" in a line marks the beginning of a comment; the rest of the line is comment. Comments and code can co-exist on the same line — one can make marginal remarks, which is not possible with Fortran's "C in column 1" convention. Blank lines are also permitted anywhere (they are not in Fortran); they should be used to emphasize the natural divisions of a program.

The syntax of the **while** statement is

```

while (legal Fortran condition)
  Ratfor statement

```

As with the **if**, *legal Fortran condition* is something that can go into a Fortran Logical IF, and *Ratfor statement* is a single statement, which may be multiple statements in braces.

The **while** encourages a style of coding not normally practiced by Fortran programmers. For example, suppose **nextch** is a function which returns the next input character both as a function value and in its argument. Then a loop to find the first non-blank character is just

```

while (nextch(ich) == iblank)
  ;

```

A semicolon by itself is a null statement, which is necessary here to mark the end of the **while**; if it were not present, the **while** would control the next statement. When the loop is broken, **ich** contains the first non-blank. Of course the same code can be written in Fortran as

```

100 if (nextch(ich) .eq. iblank) goto 100

```

but many Fortran programmers (and a few compilers) believe this line is illegal. The language at one's disposal strongly influences how one thinks about a problem.

The "for" Statement

The **for** statement is another Ratfor loop, which attempts to carry the separation of loop-body from reason-for-looping a step further than the **while**. A **for** statement allows explicit initialization and increment steps as part of the statement. For example, a DO loop is just

```

for (i = 1; i <= n; i = i + 1) ...

```

This is equivalent to

```

i = 1
while (i <= n) {
  ...
  i = i + 1
}

```

The initialization and increment of **i** have been moved into the **for** statement, making it easier to see at a glance what controls the loop.

The **for** and **while** versions have the advantage that they will be done zero times if **n** is less than 1; this is not true of the **do**.

The loop of the sine routine in the previous section can be re-written with a **for** as


```

for (i=3; abs(term) > e & i < 100; i=i+2) {
    term = -term * x**2 / float(i*(i-1))
    sin = sin + term
}

```

The syntax of the **for** statement is

```

for ( init ; condition ; increment )
    Ratfor statement

```

init is any single Fortran statement, which gets done once before the loop begins. *increment* is any single Fortran statement, which gets done at the end of each pass through the loop, before the test. *condition* is again anything that is legal in a logical IF. Any of *init*, *condition*, and *increment* may be omitted, although the semicolons *must* always be present. A non-existent *condition* is treated as always true, so **for(;;)** is an indefinite repeat. (But see the **repeat-until** in the next section.)

The **for** statement is particularly useful for backward loops, chaining along lists, loops that might be done zero times, and similar things which are hard to express with a **DO** statement, and obscure to write out with **IF**'s and **GOTO**'s. For example, here is a backwards **DO** loop to find the last non-blank character on a card:

```

for (i = 80; i > 0; i = i - 1)
    if (card(i) != blank)
        break

```

("!=" is the same as ".NE."). The code scans the columns from 80 through to 1. If a non-blank is found, the loop is immediately broken. (**break** and **next** work in **for**'s and **while**'s just as in **do**'s). If *i* reaches zero, the card is all blank.

This code is rather nasty to write with a regular Fortran **DO**, since the loop must go forward, and we must explicitly set up proper conditions when we fall out of the loop. (Forgetting this is a common error.) Thus:

```

DO 10 J = 1, 80
    I = 81 - J
    IF (CARD(I) .NE. BLANK) GO TO 11
10 CONTINUE
    I = 0
11 ...

```

The version that uses the **for** handles the termination condition properly for free; *i* is zero when we fall out of the **for** loop.

The increment in a **for** need not be an arithmetic progression; the following program walks along a list (stored in an integer array **ptr**) until a zero pointer is found, adding up elements from a parallel array of values:

```

sum = 0.0
for (i = first; i > 0; i = ptr(i))
    sum = sum + value(i)

```

Notice that the code works correctly if the list is empty. Again, placing the test at the top of a loop instead of the bottom eliminates a potential boundary error.

The "repeat-until" statement

In spite of the dire warnings, there are times when one really needs a loop that tests at the bottom after one pass through. This service is provided by the **repeat-until**:

```

repeat
    Ratfor statement
until (legal Fortran condition)

```

The *Ratfor statement* part is done once, then the condition is evaluated. If it is true, the loop is exited; if it is false, another pass is made.

The **until** part is optional, so a bare **repeat** is the cleanest way to specify an infinite loop. Of course such a loop must ultimately be broken by some transfer of control such as **stop**, **return**, or **break**, or an implicit stop such as running out of input with a **READ** statement.

As a matter of observed fact [8], the **repeat-until** statement is *much* less used than the other looping constructions; in particular, it is typically outnumbered ten to one by **for** and **while**. Be cautious about using it, for loops that test only at the bottom often don't handle null cases well.

More on break and next

break exits immediately from **do**, **while**, **for**, and **repeat-until**. **next** goes to the test part of **do**, **while** and **repeat-until**, and to the increment step of a **for**.

"return" Statement

The standard Fortran mechanism for returning a value from a function uses the name of the function as a variable which can be assigned to; the last value stored in it is the function value upon return. For example, here is a routine **equal** which returns 1 if two arrays are identical, and zero if they differ. The array ends are marked by the special value -1.

```
# equal _ compare str1 to str2;
# return 1 if equal, 0 if not
integer function equal(str1, str2)
integer str1(100), str2(100)
integer i

for (i = 1; str1(i) == str2(i); i = i + 1)
  if (str1(i) == -1) {
    equal = 1
    return
  }
equal = 0
return
end
```

In many languages (e.g., PL/I) one instead says

```
return (expression)
```

to return a value from a function. Since this is often clearer, Ratfor provides such a **return** statement — in a function *F*, **return(expression)** is equivalent to

```
{ F = expression; return }
```

For example, here is **equal** again:

```
# equal _ compare str1 to str2;
# return 1 if equal, 0 if not
integer function equal(str1, str2)
integer str1(100), str2(100)
integer i

for (i = 1; str1(i) == str2(i); i = i + 1)
  if (str1(i) == -1)
    return(1)
return(0)
end
```

If there is no parenthesized expression after **return**, a normal RETURN is made. (Another version of **equal** is presented shortly.)

Cosmetics

As we said above, the visual appearance of a language has a substantial effect on how easy it is to read and understand programs. Accordingly, Ratfor provides a number of cosmetic facilities which may be used to make programs more readable.

Free-form Input

Statements can be placed anywhere on a line; long statements are continued automatically, as are long conditions in **if**, **while**, **for**, and **until**. Blank lines are ignored. Multiple statements may appear on one line, if they are separated by semicolons. No semicolon is needed at the end of a line, if Ratfor can make

some reasonable guess about whether the statement ends there. Lines ending with any of the characters

```
= + - * , | & ( _
```

are assumed to be continued on the next line. Underscores are discarded wherever they occur; all others remain as part of the statement.

Any statement that begins with an all-numeric field is assumed to be a Fortran label, and placed in columns 1-5 upon output. Thus

```
write(6, 100); 100 format("hello")
```

is converted into

```
write(6, 100)
100 format(5hello)
```

Translation Services

Text enclosed in matching single or double quotes is converted to **nH...** but is otherwise unaltered (except for formatting — it may get split across card boundaries during the reformatting process). Within quoted strings, the backslash '****' serves as an escape character: the next character is taken literally. This provides a way to get quotes (and of course the backslash itself) into quoted strings:

```
"\\\""
```

is a string containing a backslash and an apostrophe. (This is *not* the standard convention of doubled quotes, but it is easier to use and more general.)

Any line that begins with the character '**%**' is left absolutely unaltered except for stripping off the '**%**' and moving the line one position to the left. This is useful for inserting control cards, and other things that should not be transmogrified (like an existing Fortran program). Use '**%**' only for ordinary statements, not for the condition parts of **if**, **while**, etc., or the output may come out in an unexpected place.

The following character translations are made, except within single or double quotes or on a line beginning with a '**%**'.

=	=	.eq.	!=	.ne.
>		.gt.	>=	.ge.
<		.lt.	<=	.le.
&		.and.		.or.
!		.not.	-	.not.

In addition, the following translations are provided for input devices with restricted character sets.

[{]	}
\$({	\$)	}

“define” Statement

Any string of alphanumeric characters can be defined as a name; thereafter, whenever that name occurs in the input (delimited by non-alphanumerics) it is replaced by the rest of the definition line. (Comments and trailing white spaces are stripped off). A defined name can be arbitrarily long, and must begin with a letter.

define is typically used to create symbolic parameters:

```
define ROWS 100
define COLS 50

dimension a(ROWS), b(ROWS, COLS)
    if (i > ROWS | j > COLS) ...
```

Alternately, definitions may be written as

```
define(ROWS, 100)
```

In this case, the defining text is everything after the comma up to the balancing right parenthesis; this allows multi-line definitions.

It is generally a wise practice to use symbolic parameters for most constants, to help make clear the function of what would otherwise be mysterious numbers. As an example, here is the routine **equal** again, this time with symbolic constants.

```
define YES 1
define NO 0
define EOS -1
define ARB 100

# equal _ compare str1 to str2;
# return YES if equal, NO if not
integer function equal(str1, str2)
integer str1(ARB), str2(ARB)
integer i

for (i = 1; str1(i) == str2(i); i = i + 1)
    if (str1(i) == EOS)
        return(YES)
return(NO)
end
```

“include” Statement

The statement

```
include file
```

inserts the file found on input stream *file* into the Ratfor input in place of the **include** statement. The standard usage is to place **COMMON** blocks on a file, and **include** that file whenever a copy is needed:

```
subroutine x
    include commonblocks
    ...
end

suroutine y
    include commonblocks
    ...
end
```

This ensures that all copies of the **COMMON** blocks are identical

Pitfalls, Botches, Blemishes and other Failings

Ratfor catches certain syntax errors, such as missing braces, **else** clauses without an **if**, and most errors involving missing parentheses in statements. Beyond that, since Ratfor knows no Fortran, any errors you make will be reported by the Fortran compiler, so you will from time to time have to relate a Fortran diagnostic back to the Ratfor source.

Keywords are reserved — using **if**, **else**, etc., as variable names will typically wreak havoc. Don't leave spaces in keywords. Don't use the Arithmetic IF.

The Fortran **nH** convention is not recognized anywhere by Ratfor; use quotes instead.

3. IMPLEMENTATION

Ratfor was originally written in C [4] on the UNIX† operating system [5]. The language is specified by a context free grammar and the compiler constructed using the YACC compiler-compiler [6].

The Ratfor grammar is simple and straightforward, being essentially:

```
prog : stat
    | prog stat
stat : if (...) stat
    | if (...) stat else stat
    | while (...) stat
    | for (...; ...; ...) stat
    | do ... stat
    | repeat stat
    | repeat stat until (...)
    | switch (...) { case ...: prog ...
                    default: prog }
    | return
    | break
    | next
    | digits stat
    | { prog }
    | anything unrecognizable
```

† UNIX is a trademark of Bell Laboratories.

The observation that Ratfor knows no Fortran follows directly from the rule that says a statement is "anything unrecognizable". In fact most of Fortran falls into this category, since any statement that does not begin with one of the keywords is by definition "unrecognizable."

Code generation is also simple. If the first thing on a source line is not a keyword (like *if*, *else*, etc.) the entire statement is simply copied to the output with appropriate character translation and formatting. (Leading digits are treated as a label.) Keywords cause only slightly more complicated actions. For example, when *if* is recognized, two consecutive labels *L* and *L+1* are generated and the value of *L* is stacked. The condition is then isolated, and the code

```
if (.not. (condition)) goto L
```

is output. The *statement* part of the *if* is then translated. When the end of the statement is encountered (which may be some distance away and include nested *if*'s, of course), the code

```
L    continue
```

is generated, unless there is an *else* clause, in which case the code is

```
      goto L+1
L    continue
```

In this latter case, the code

```
L+1 continue
```

is produced after the *statement* part of the *else*. Code generation for the various loops is equally simple.

One might argue that more care should be taken in code generation. For example, if there is no trailing *else*,

```
if (i > 0) x = a
```

should be left alone, not converted into

```
if (.not. (i .gt. 0)) goto 100
x = a
100 continue
```

But what are optimizing compilers for, if not to improve code? It is a rare program indeed where this kind of "inefficiency" will make even a measurable difference. In the few cases where it is important, the offending lines can be protected by '%'.

The use of a compiler-compiler is definitely the preferred method of software development. The language is well-defined, with few syntactic irregularities. Implementation is quite simple; the original construction took under a week. The language is sufficiently simple, however, that an *ad hoc* recognizer can be

readily constructed to do the same job if no compiler-compiler is available.

The C version of Ratfor is used on UNIX and on the Honeywell GCOS systems. C compilers are not as widely available as Fortran, however, so there is also a Ratfor written in itself and originally bootstrapped with the C version. The Ratfor version was written so as to translate into the portable subset of Fortran described in [1], so it is portable, having been run essentially without change on at least twelve distinct machines. (The main restrictions of the portable subset are: only one character per machine word; subscripts in the form $c*v\pm c$; avoiding expressions in places like DO loops; consistency in subroutine argument usage, and in COMMON declarations. Ratfor itself will not gratuitously generate non-standard Fortran.)

The Ratfor version is about 1500 lines of Ratfor (compared to about 1000 lines of C); this compiles into 2500 lines of Fortran. This expansion ratio is somewhat higher than average, since the compiled code contains unnecessary occurrences of COMMON declarations. The execution time of the Ratfor version is dominated by two routines that read and write cards. Clearly these routines could be replaced by machine coded local versions; unless this is done, the efficiency of other parts of the translation process is largely irrelevant.

4. EXPERIENCE

Good Things

"It's so much better than Fortran" is the most common response of users when asked how well Ratfor meets their needs. Although cynics might consider this to be vacuous, it does seem to be true that decent control flow and cosmetics converts Fortran from a bad language into quite a reasonable one, assuming that Fortran data structures are adequate for the task at hand.

Although there are no quantitative results, users feel that coding in Ratfor is at least twice as fast as in Fortran. More important, debugging and subsequent revision are much faster than in Fortran. Partly this is simply because the code can be *read*. The looping statements which test at the top instead of the bottom seem to eliminate or at least reduce the occurrence of a wide class of boundary errors. And of course it is easy to do structured programming in Ratfor; this self-discipline also contributes markedly to reliability.

One interesting and encouraging fact is that programs written in Ratfor tend to be as readable as programs written in more modern

languages like Pascal. Once one is freed from the shackles of Fortran's clerical detail and rigid input format, it is easy to write code that is readable, even esthetically pleasing. For example, here is a Ratfor implementation of the linear table search discussed by Knuth [7]:

```
A(m+1) = x
for (i = 1; A(i) != x; i = i + 1)
;
if (i > m) {
    m = i
    B(i) = 1
}
else
    B(i) = B(i) + 1
```

A large corpus (5400 lines) of Ratfor, including a subset of the Ratfor preprocessor itself, can be found in [8].

Bad Things

The biggest single problem is that many Fortran syntax errors are not detected by Ratfor but by the local Fortran compiler. The compiler then prints a message in terms of the generated Fortran, and in a few cases this may be difficult to relate back to the offending Ratfor line, especially if the implementation conceals the generated Fortran. This problem could be dealt with by tagging each generated line with some indication of the source line that created it, but this is inherently implementation-dependent, so no action has yet been taken. Error message interpretation is actually not so arduous as might be thought. Since Ratfor generates no variables, only a simple pattern of IF's and GOTO's, data-related errors like missing DIMENSION statements are easy to find in the Fortran. Furthermore, there has been a steady improvement in Ratfor's ability to catch trivial syntactic errors like unbalanced parentheses and quotes.

There are a number of implementation weaknesses that are a nuisance, especially to new users. For example, keywords are reserved. This rarely makes any difference, except for those hardy souls who want to use an Arithmetic IF. A few standard Fortran constructions are not accepted by Ratfor, and this is perceived as a problem by users with a large corpus of existing Fortran programs. Protecting every line with a '%' is not really a complete solution, although it serves as a stop-gap. The best long-term solution is provided by the program Struct [9], which converts arbitrary Fortran programs into Ratfor.

Users who export programs often complain that the generated Fortran is "unreadable" because it is not tastefully formatted and contains extraneous CONTINUE statements. To some

extent this can be ameliorated (Ratfor now has an option to copy Ratfor comments into the generated Fortran), but it has always seemed that effort is better spent on the input language than on the output esthetics.

One final problem is partly attributable to success — since Ratfor is relatively easy to modify, there are now several dialects of Ratfor. Fortunately, so far most of the differences are in character set, or in invisible aspects like code generation.

5. CONCLUSIONS

Ratfor demonstrates that with modest effort it is possible to convert Fortran from a bad language into quite a good one. A preprocessor is clearly a useful way to extend or ameliorate the facilities of a base language.

When designing a language, it is important to concentrate on the essential requirement of providing the user with the best language possible for a given effort. One must avoid throwing in "features" — things which the user may trivially construct within the existing framework.

One must also avoid getting sidetracked on irrelevancies. For instance it seems pointless for Ratfor to prepare a neatly formatted listing of either its input or its output. The user is presumably capable of the self-discipline required to prepare neat input that reflects his thoughts. It is much more important that the language provide free-form input so he *can* format it neatly. No one should read the output anyway except in the most dire circumstances.

Acknowledgements

C. A. R. Hoare once said that "One thing [the language designer] should not do is to include untried ideas of his own." Ratfor follows this precept very closely — everything in it has been stolen from someone else. Most of the control flow structures are taken directly from the language C [4] developed by Dennis Ritchie; the comment and continuation conventions are adapted from Altran [10].

I am grateful to Stuart Feldman, whose patient simulation of an innocent user during the early days of Ratfor led to several design improvements and the eradication of bugs. He also translated the C parse-tables and YACC parser into Fortran for the first Ratfor version of Ratfor.

Appendix: Usage on UNIX.

Beware — local customs vary. Check with a native before going into the jungle.

The program `ratfor` is the basic translator; it takes either a list of file names or the standard input and writes Fortran on the standard output. Options include `-6x`, which uses `x` as a continuation character in column 6 (UNIX uses `&` in column 1), and `-C`, which causes Ratfor comments to be copied into the generated Fortran.

The program `rc` provides an interface to the `ratfor` command which is much the same as `cc`. Thus

`rc [options] files`

compiles the files specified by `files`. Files with names ending in `.r` are Ratfor source; other files are assumed to be for the loader. The flags `-C` and `-6x` described above are recognized, as are

- `-c` compile only; don't load
- `-f` save intermediate Fortran `.f` files
- `-r` Ratfor only; implies `-c` and `-f`
- `-2` use big Fortran compiler
(for large programs)
- `-U` flag undeclared variables
(not universally available)

Other flags are passed on to the loader.

References

- [1] B. G. Ryder, "The PFORT Verifier," *Software—Practice & Experience*, October 1974.
- [2] American National Standard Fortran. American National Standards Institute, New York, 1966.
- [3] *For-word: Fortran Development Newsletter*, August 1975.
- [4] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Inc., 1978.
- [5] D. M. Ritchie and K. L. Thompson, "The UNIX Time-sharing System." *CACM*, July 1974.
- [6] S. C. Johnson, "YACC — Yet Another Compiler-Compiler." Bell Laboratories, 1978.
- [7] D. E. Knuth, "Structured Programming with goto Statements." *Computing Surveys*, December 1974.
- [8] B. W. Kernighan and P. J. Plauger, *Software Tools*, Addison-Wesley, 1976.
- [9] B. S. Baker, "Struct — A Program which Structures Fortran", Bell Laboratories internal memorandum, December 1975.
- [10] A. D. Hall, "The Altran System for Rational Function Manipulation — A Survey." *CACM*, August 1971.

January 1981

The Programming Language EFL

S. I. Feldman

Bell Laboratories
Murray Hill, New Jersey 07974

1. INTRODUCTION

1.1. Purpose

EFL is a clean, general purpose computer language intended to encourage portable programming. It has a uniform and readable syntax and good data and control flow structuring. EFL programs can be translated into efficient Fortran code, so the EFL programmer can take advantage of the ubiquity of Fortran, the valuable libraries of software written in that language, and the portability that comes with the use of a standardized language, without suffering from Fortran's many failings as a language. It is especially useful for numeric programs. Thus, the EFL language permits the programmer to express complicated ideas in a comprehensible way, while permitting access to the power of the Fortran environment.

1.2. History

EFL can be viewed as a descendant of B. W. Kernighan's Ratfor [1]; the name originally stood for "Extended Fortran Language". A. D. Hall designed the initial version of the language and wrote a preliminary version of a compiler. I extended and modified the language and wrote a full compiler (in C) for it. The current compiler is much more than a simple preprocessor: it attempts to diagnose all syntax errors, to provide readable Fortran output, and to avoid a number of niggling restrictions. To achieve this goal, a sizable two-pass translator is needed.

1.3. Notation

In examples and syntax specifications, **boldface** type is used to indicate literal words and punctuation, such as **while**. Words in *italic* type indicate an item in a category, such as an *expression*. A construct surrounded by double brackets represents a list of one or more of those items, separated by commas. Thus, the notation

[*item*]

could refer to any of the following:

item
item, item
item, item, item

The reader should have a fair degree of familiarity with some procedural language. There will be occasional references to Ratfor and to Fortran which may be ignored if the reader is unfamiliar with those languages.

2. LEXICAL FORM

2.1. Character Set

The following characters are legal in an EFL program:

<i>letters</i>	a b c d e f g h i j k l m n o p q r s t u v w x y z
<i>digits</i>	0 1 2 3 4 5 6 7 8 9
<i>white space</i>	<i>blank tab</i>
<i>quotes</i>	' "
<i>sharp</i>	#
<i>continuation</i>	_
<i>braces</i>	{ }
<i>parentheses</i>	()
<i>other</i>	, ; : . + - * / = < > & ~ \$

Letter case (upper or lower) is ignored except within strings, so "a" and "A" are treated as the same character. All of the examples below are printed in lower case. An exclamation mark ("!") may be used in place of a tilde ("~"). Square brackets ("[" and "]") may be used in place of braces ("{" and "}").

2.2. Lines

EFL is a line-oriented language. Except in special cases (discussed below), the end of a line marks the end of a token and the end of a statement. The trailing portion of a line may be used for a comment. There is a mechanism for diverting input from one source file to another, so a single line in the program may be replaced by a number of lines from the other file. Diagnostic messages are labeled with the line number of the file on which they are detected.

2.2.1. White Space

Outside of a character string or comment, any sequence of one or more spaces or tab characters acts as a single space. Such a space terminates a token.

2.2.2. Comments

A comment may appear at the end of any line. It is introduced by a sharp (#) character, and continues to the end of the line. (A sharp inside of a quoted string does not mark a comment.) The sharp and succeeding characters on the line are discarded. A blank line is also a comment. Comments have no effect on execution.

2.2.3. Include Files

It is possible to insert the contents of a file at a point in the source text, by referencing it in a line like

```
include joe
```

No statement or comment may follow an **include** on a line. In effect, the **include** line is replaced by the lines in the named file, but diagnostics refer to the line number in the included file. **Includes** may be nested at least ten deep.

2.2.4. Continuation

Lines may be continued explicitly by using the underscore (_) character. If the last character of a line (after comments and trailing white space have been stripped) is an underscore, the end of line and the initial blanks on the next line are ignored. Underscores are ignored in other contexts (except inside of quoted strings). Thus

1_000_000_
000

equals 10^9 .

There are also rules for continuing lines automatically: the end of line is ignored whenever it is obvious that the statement is not complete. To be specific, a statement is continued if the last token on a line is an operator, comma, left brace, or left parenthesis. (A statement is not continued just because of unbalanced braces or parentheses.) Some compound statements are also continued automatically; these points are noted in the sections on executable statements.

2.2.5. Multiple Statements on a Line

A semicolon terminates the current statement. Thus, it is possible to write more than one statement on a line. A line consisting only of a semicolon, or a semicolon following a semicolon, forms a null statement.

2.3. Tokens

A program is made up of a sequence of tokens. Each token is a sequence of characters. A blank terminates any token other than a quoted string. End of line also terminates a token unless explicit continuation (see above) is signaled by an underscore.

2.3.1. Identifiers

An identifier is a letter or a letter followed by letters or digits. The following is a list of the reserved words that have special meaning in EFL. They will be discussed later.

array	exit	precision
automatic	external	procedure
break	false	read
call	field	readbin
case	for	real
character	function	repeat
common	go	return
complex	goto	select
continue	if	short
debug	implicit	sizeof
default	include	static
define	initial	struct
dimension	integer	subroutine
do	internal	true
double	lengthof	until
doubleprecision	logical	value
else	long	while
end	next	write
equivalence	option	writebin

The use of these words is discussed below. These words may not be used for any other purpose.

2.3.2. Strings

A character string is a sequence of characters surrounded by quotation marks. If the string is bounded by single-quote marks ('), it may contain double quote marks ("), and vice versa. A quoted string may not be broken across a line boundary.

```
'hello there'
'ain't misbehavin''
```

2.3.3. Integer Constants

An integer constant is a sequence of one or more digits.

```
0
57
123456
```

2.3.4. Floating Point Constants

A floating point constant contains a dot and/or an exponent field. An *exponent field* is a letter *d* or *e* followed by an optionally signed integer constant. If *I* and *J* are integer constants and *E* is an exponent field, then a floating constant has one of the following forms:

```
.I
I.
I.J
IE
I.E
.JE
I.JE
```

2.3.5. Punctuation

Certain characters are used to group or separate objects in the language. These are

```
parentheses ( )
braces      { }
comma       ,
semicolon   ;
colon       :
end-of-line
```

The end-of-line is a token (statement separator) when the line is neither blank nor continued.

2.3.6. Operators

The EFL operators are written as sequences of one or more non-alphanumeric characters.

```
+ - * / **
< <= > >= == ~=
&& || & |
+= -= /= **=
&&= ||= &= |=
-> . $
```

A dot (".") is an operator when it qualifies a structure element name, but not when it acts as a decimal point in a numeric constant. There is a special mode (see the Atavisms section) in which some of the operators may be represented by a string consisting of a dot, an identifier, and a dot (e.g., `.lt.`).

2.4. Macros

EFL has a simple macro substitution facility. An identifier may be defined to be equal to a string of tokens; whenever that name appears as a token in the program, the string replaces it. A macro name is given a value in a **define** statement like

```
define count    n += 1
```

Any time the name **count** appears in the program, it is replaced by the statement

```
n += 1
```

A **define** statement must appear alone on a line; the form is

```
define name rest-of-line
```

Trailing comments are part of the string.

3. PROGRAM FORM

3.1. Files

A *file* is a sequence of lines. A file is compiled as a single unit. It may contain one or more procedures. Declarations and options that appear outside of a procedure affect the succeeding procedures on that file.

3.2. Procedures

Procedures are the largest grouping of statements in EFL. Each procedure has a name by which it is invoked. (The first procedure invoked during execution, known as the *main* procedure, has the null name.) Procedure calls and argument passing are discussed in Section 8.

3.3. Blocks

Statements may be formed into groups inside of a procedure. To describe the scope of names, it is convenient to introduce the ideas of *block* and of *nesting level*. The beginning of a program file is at nesting level zero. Any options, macro definitions, or variable declarations there are also at level zero. The text immediately following a **procedure** statement is at level 1. After the declarations, a left brace marks the beginning of a new block and increases the nesting level by 1; a right brace drops the level by 1. (Braces inside declarations do not mark blocks.) (See Section 7.2). An **end** statement marks the end of the procedure, level 1, and the return to level 0. A name (variable or macro) that is defined at level *k* is defined throughout that block and in all deeper nested levels in which that name is not redefined or redeclared. Thus, a procedure might look like the following:

```
# block 0
procedure george
real x
x = 2
...
if(x > 2)
    {           # new block
integer x     # a different variable
do x = 1,7
                write(,x)
    ...
    }           # end of block
end           # end of procedure, return to block 0
```

3.4. Statements

A statement is terminated by end of line or by a semicolon. Statements are of the following types:

Option
Include
Define
Procedure
End
Declarative
Executable

The **option** statement is described in Section 10. The **include**, **define**, and **end** statements have been described above; they may not be followed by another statement on a line. Each procedure begins with a **procedure** statements and finishes with an **end** statement; these are discussed in Section 8. Declarations describe types and values of variables and procedures. Executable statements cause specific actions to be taken. A block is an example of an executable statement; it is made up of declarative and executable statements.

3.5. Labels

An executable statement may have a *label* which may be used in a branch statement. A label is an identifier followed by a colon, as in

```

                                read(, x)
                                if(x < 3) goto error
                                ...
error:                          fatal("bad input")

```

4. DATA TYPES AND VARIABLES

EFL supports a small number of basic (scalar) types. The programmer may define objects made up of variables of basic type; other aggregates may then be defined in terms of previously defined aggregates.

4.1. Basic Types

The basic types are

logical
integer
field(*m:n*)
real
complex
long real
long complex
character(*n*)

A logical quantity may take on the two values true and false. An integer may take on any whole number value in some machine-dependent range. A field quantity is an integer restricted to a particular closed interval (*[m:n]*). A "real" quantity is a floating point approximation to a real or rational number. A long real is a more precise approximation to a rational. (Real quantities are represented as single precision floating point numbers; long reals are double precision floating point numbers.) A complex quantity is an approximation to a complex number, and is represented as a pair of reals. A character quantity is a fixed-length string of *n* characters.

4.2. Constants

There is a notation for a constant of each basic type.

A logical may take on the two values

true
false

An integer or field constant is a fixed point constant, optionally preceded by a plus or minus sign, as in

17
-94
+6
0

A long real ("double precision") constant is a floating point constant containing an exponent field that begins with the letter **d**. A real ("single precision") constant is any other floating point constant. A real or long real constant may be preceded by a plus or minus sign. The following are valid **real** constants:

17.3
-.4
7.9e-6 ($= 7.9 \times 10^{-6}$)
14e9 ($= 1.4 \times 10^{10}$)

The following are valid **long real** constants

7.9d-6 ($= 7.9 \times 10^{-6}$)
5d3

A character constant is a quoted string.

4.3. Variables

A variable is a quantity with a name and a location. At any particular time the variable may also have a value. (A variable is said to be *undefined* before it is initialized or assigned its first value, and after certain indefinite operations are performed.) Each variable has certain attributes:

4.3.1. Storage Class

The association of a name and a location is either transitory or permanent. Transitory association is achieved when arguments are passed to procedures. Other associations are permanent (static). (A future extension of EFL may include dynamically allocated variables.)

4.3.2. Scope of Names

The names of common areas are global, as are procedure names: these names may be used anywhere in the program. All other names are local to the block in which they are declared.

4.3.3. Precision

Floating point variables are either of normal or **long** precision. This attribute may be stated independently of the basic type.

4.4. Arrays

It is possible to declare rectangular arrays (of any dimension) of values of the same type. The index set is always a cross-product of intervals of integers. The lower and upper bounds of the intervals must be constants for arrays that are local or **common**. A formal argument array may have intervals that are of length equal to one of the other formal arguments. An element of an array is denoted by the array name followed by a parenthesized comma-separated list of integer values, each of which must lie within the corresponding interval. (The intervals may include negative numbers.) Entire arrays may be passed as procedure arguments or in input/output lists, or they may be initialized; all other array references must be to individual elements.

4.5. Structures

It is possible to define new types which are made up of elements of other types. The compound object is known as a *structure*; its constituents are called *members* of the structure. The structure may be given a name, which acts as a type name in the remaining statements within the scope of its declaration. The elements of a structure may be of any type (including previously defined structures), or they may be arrays of such objects. Entire structures may be passed to procedures or be used in input/output lists; individual elements of structures may be referenced. The uses of structures will be detailed below. The following structure might represent a symbol table:

```

struct tableentry
{
    character(8) name
    integer hashvalue
    integer numberofelements
    field(0:1) initialized, used, set
    field(0:10) type
}

```

5. EXPRESSIONS

Expressions are syntactic forms that yield a value. An expression may have any of the following forms, recursively applied:

```

primary
( expression )
unary-operator expression
expression binary-operator expression

```

In the following table of operators, all operators on a line have equal precedence and have higher precedence than operators on later lines. The meanings of these operators are described in sections 5.3 and 5.4.

```

-> .
**
* / unary + - ++ --
+ -
< <= > >= == ~=
& &&
| ||
$
= += -= *= /= **= &= |= &&= ||=

```

Examples of expressions are

$$\begin{aligned} & \mathbf{a < b \ \&\& \ b < c} \\ & \mathbf{-(a + \sin(x)) / (5 + \cos(x))^{**2}} \end{aligned}$$

5.1. Primaries

Primaries are the basic elements of expressions, as follows:

5.1.1. Constants

Constants are described in Section 4.2.

5.1.2. Variables

Scalar variable names are primaries. They may appear on the left or the right side of an assignment. Unqualified names of aggregates (structures or arrays) may only appear as procedure arguments and in input/output lists.

5.1.3. Array Elements

An element of an array is denoted by the array name followed by a parenthesized list of subscripts, one integer value for each declared dimension:

$$\begin{aligned} & \mathbf{a(5)} \\ & \mathbf{b(6, -3, 4)} \end{aligned}$$

5.1.4. Structure Members

A structure name followed by a dot followed by the name of a member of that structure constitutes a reference to that element. If that element is itself a structure, the reference may be further qualified.

$$\begin{aligned} & \mathbf{a.b} \\ & \mathbf{x(3).y(4).z(5)} \end{aligned}$$

5.1.5. Procedure Invocations

A procedure is invoked by an expression of one of the forms

$$\begin{aligned} & \mathbf{procedurename ()} \\ & \mathbf{procedurename (expression)} \\ & \mathbf{procedurename (expression-1, \dots, expression-n)} \end{aligned}$$

The *procedurename* is either the name of a variable declared **external** or it is the name of a function known to the EFL compiler (see Section 8.5), or it is the actual name of a procedure, as it appears in a **procedure** statement. If a *procedurename* is declared **external** and is an argument of the current procedure, it is associated with the procedure name passed as actual argument; otherwise it is the actual name of a procedure. Each *expression* in the above is called an *actual argument*. Examples of procedure invocations are

$$\begin{aligned} & \mathbf{f(x)} \\ & \mathbf{work()} \\ & \mathbf{g(x, y+3, 'xx')} \end{aligned}$$

When one of these procedure invocations is to be performed, each of the actual argument expressions is first evaluated. The types, precisions, and bounds of actual and formal arguments should agree. If an actual argument is a variable name, array element, or structure member, the called procedure is permitted to use the corresponding formal argument as the left side of an assignment or in an input list; otherwise it may only use the value. After the formal and actual arguments are associated, control is passed to the first executable statement of the

procedure. When a **return** statement is executed in that procedure, or when control reaches the **end** statement of that procedure, the function value is made available as the value of the procedure invocation. The type of the value is determined by the attributes of the *procedurename* that are declared or implied in the calling procedure, which must agree with the attributes declared for the function in its procedure. In the special case of a generic function, the type of the result is also affected by the type of the argument. See Chapter 8 for details.

5.1.6. Input/Output Expressions

The EFL input/output syntactic forms may be used as integer primaries that have a non-zero value if an error occurs during the input or output. See Section 7.7.

5.1.7. Coercions

An expression of one precision or type may be converted to another by an expression of the form

$$\text{attributes (expression)}$$

At present, the only *attributes* permitted are precision and basic types. Attributes are separated by white space. An arithmetic value of one type may be coerced to any other arithmetic type; a character expression of one length may be coerced to a character expression of another length; logical expressions may not be coerced to a nonlogical type. As a special case, a quantity of **complex** or **long complex** type may be constructed from two integer or real quantities by passing two expressions (separated by a comma) in the coercion. Examples and equivalent values are

$$\begin{aligned} \text{integer}(5.3) &= 5 \\ \text{long real}(5) &= 5.0d0 \\ \text{complex}(5,3) &= 5+3i \end{aligned}$$

Most conversions are done implicitly, since most binary operators permit operands of different arithmetic types. Explicit coercions are of most use when it is necessary to convert the type of an actual argument to match that of the corresponding formal parameter in a procedure call.

5.1.8. Sizes

There is a notation which yields the amount of memory required to store a datum or an item of specified type:

$$\begin{aligned} \text{sizeof (leftside)} \\ \text{sizeof (attributes)} \end{aligned}$$

In the first case, *leftside* can denote a variable, array, array element, or structure member. The value of **sizeof** is an integer, which gives the size in arbitrary units. If the size is needed in terms of the size of some specific unit, this can be computed by division:

$$\text{sizeof}(x) / \text{sizeof}(\text{integer})$$

yields the size of the variable *x* in integer words.

The distance between consecutive elements of an array may not equal **sizeof** because certain data types require final padding on some machines. The **lengthof** operator gives this larger value, again in arbitrary units. The syntax is

$$\begin{aligned} \text{lengthof (leftside)} \\ \text{lengthof (attributes)} \end{aligned}$$

5.2. Parentheses

An expression surrounded by parentheses is itself an expression. A parenthesized expression must be evaluated before an expression of which it is a part is evaluated.

5.3. Unary Operators

All of the unary operators in EFL are prefix operators. The result of a unary operator has the same type as its operand.

5.3.1. Arithmetic

Unary $+$ has no effect. A unary $-$ yields the negative of its operand.

The prefix operator $++$ adds one to its operand. The prefix operator $--$ subtracts one from its operand. The value of either expression is the result of the addition or subtraction. For these two operators, the operand must be a scalar, array element, or structure member of arithmetic type. (As a side effect, the operand value is changed.)

5.3.2. Logical

The only logical unary operator is complement (\sim). This operator is defined by the equations

```

~ true = false
~ false = true

```

5.4. Binary Operators

Most EFL operators have two operands, separated by the operator. Because the character set must be limited, some of the operators are denoted by strings of two or three special characters. All binary operators except exponentiation are left associative.

5.4.1. Arithmetic

The binary arithmetic operators are

```

+  addition
-  subtraction
*  multiplication
/  division
** exponentiation

```

Exponentiation is right associative: $a**b**c = a**(b**c) = a^{(b^c)}$. The operations have the conventional meanings: $8+2 = 10$, $8-2 = 6$, $8*2 = 16$, $8/2 = 4$, $8**2 = 8^2 = 64$.

The type of the result of a binary operation $A \text{ op } B$ is determined by the types of its operands:

Type of A	Type of B				
	integer	real	long real	complex	long complex
integer	integer	real	long real	complex	long complex
real	real	real	long real	complex	long complex
long real	long real	long real	long real	long complex	long complex
complex	complex	complex	long complex	complex	long complex
long complex	long complex	long complex	long complex	long complex	long complex

If the type of an operand differs from the type of the result, the calculation is done as if the operand were first coerced to the type of the result. If both operands are integers, the result is of type integer, and is computed exactly. (Quotients are truncated toward zero, so $8/3=2$.)

5.4.2. Logical

The two binary logical operations in EFL, **and** and **or**, are defined by the truth tables:

A	B	A and B	A or B
false	false	false	false
false	true	false	true
true	false	false	true
true	true	true	true

Each of these operators comes in two forms. In one form, the order of evaluation is specified. The expression

a && b

is evaluated by first evaluating **a**; if it is false then the expression is false and **b** is not evaluated; otherwise the expression has the value of **b**. The expression

a || b

is evaluated by first evaluating **a**; if it is true then the expression is true and **b** is not evaluated; otherwise the expression has the value of **b**. The other forms of the operators (**&** for **and** and **|** for **or**) do not imply an order of evaluation. With the latter operators, the compiler may speed up the code by evaluating the operands in any order.

5.4.3. Relational Operators

There are six relations between arithmetic quantities. These operators are not associative.

EFL Operator	Meaning
<	< less than
<=	≤ less than or equal to
==	= equal to
~=	≠ not equal to
>	> greater than
>=	≥ greater than or equal

Since the complex numbers are not ordered, the only relational operators that may take complex operands are **==** and **~=**. The character collating sequence is not defined.

5.4.4. Assignment Operators

All of the assignment operators are right associative. The simple form of assignment is

basic-left-side = expression

A *basic-left-side* is a scalar variable name, array element, or structure member of basic type. This statement computes the expression on the right side, and stores that value (possibly after coercing the value to the type of the left side) in the location named by the left side. The value of the assignment expression is the value assigned to the left side after coercion.

There is also an assignment operator corresponding to each binary arithmetic and logical operator. In each case, $a \text{ op} = b$ is equivalent to $a = a \text{ op } b$. (The operator and equal sign must not be separated by blanks.) Thus, $n += 2$ adds 2 to n . The location of the left side is evaluated only once.

5.5. Dynamic Structures

EFL does not have an address (pointer, reference) type. However, there is a notation for dynamic structures,

leftside -> structurename

This expression is a structure with the shape implied by *structurename* but starting at the location of *leftside*. In effect, this overlays the structure template at the specified location. The *leftside* must be a variable, array, array element, or structure member. The type of the *leftside* must be one of the types in the structure declaration. An element of such a structure is denoted in the usual way using the dot operator. Thus,

place(i) -> st.elt

refers to the **elt** member of the **st** structure starting at the i^{th} element of the array **place**.

5.6. Repetition Operator

Inside of a list, an element of the form

integer-constant-expression \$ *constant-expression*

is equivalent to the appearance of the *expression* a number of times equal to the first expression. Thus,

(3, 3\$4, 5)

is equivalent to

(3, 4, 4, 4, 5)

5.7. Constant Expressions

If an expression is built up out of operators (other than functions) and constants, the value of the expression is a constant, and may be used anywhere a constant is required.

6. DECLARATIONS

Declarations statement describe the meaning, shape, and size of named objects in the EFL language.

6.1. Syntax

A declaration statement is made up of attributes and variables. Declaration statements are of two form:

attributes *variable-list*
attributes { *declarations* }

In the first case, each name in the *variable-list* has the specified attributes. In the second, each name in the declarations also has the specified attributes. A variable name may appear in more than one variable list, so long as the attributes are not contradictory. Each name of a nonargument variable may be accompanied by an initial value specification. The *declarations* inside the braces are one or more declaration statements. Examples of declarations are

```
integer k=2
long real b(7,3)
common(cname)
{
  integer i
  long real array(5,0:3) x, y
  character(7) ch
}
```

6.2. Attributes

6.2.1. Basic Types

The following are basic types in declarations

```

logical
integer
field(m:n)
character(k)
real
complex

```

In the above, the quantities k , m , and n denote integer constant expressions with the properties $k > 0$ and $n > m$.

6.2.2. Arrays

The dimensionality may be declared by an **array** attribute

```
array(b1, ..., bn)
```

Each of the b_i may either be a single integer expression or a pair of integer expressions separated by a colon. The pair of expressions form a lower and an upper bound; the single expression is an upper bound with an implied lower bound of 1. The number of dimensions is equal to n , the number of bounds. All of the integer expressions must be constants. An exception is permitted only if all of the variables associated with an array declarator are formal arguments of the procedure; in this case, each bound must have the property that $upper - lower + 1$ is equal to a formal argument of the procedure. (The compiler has limited ability to simplify expressions, but it will recognize important cases such as $(0:n-1)$. The upper bound for the last dimension (b_n) may be marked by an asterisk ($*$) if the size of the array is not known. The following are legal **array** attributes:

```

array(5)
array(5, 1:5, -3:0)
array(5, *)
array(0:m-1, m)

```

6.2.3. Structures

A structure declaration is of the form

```
struct structname { declaration statements }
```

The *structname* is optional; if it is present, it acts as if it were the name of a type in the rest of its scope. Each name that appears inside the *declarations* is a *member* of the structure, and has a special meaning when used to qualify any variable declared with the structure type. A name may appear as a member of any number of structures, and may also be the name of an ordinary variable, since a structure member name is used only in contexts where the parent type is known. The following are valid structure attributes

```

struct xx
{
integer a, b
real x(5)
}

struct { xx z(3); character(5) y }

```

The last line defines a structure containing an array of three *xx*'s and a character string.

6.2.4. Precision

Variables of floating point (**real** or **complex**) type may be declared to be **long** to ensure they have higher precision than ordinary floating point variables. The default precision is **short**.

6.2.5. Common

Certain objects called *common areas* have external scope, and may be referenced by any procedure that has a declaration for the name using a

common (*commonareaname*)

attribute. All of the variables declared with a particular **common** attribute are in the same block; the order in which they are declared is significant. Declarations for the same block in differing procedures must have the variables in the same order and with the same types, precision, and shapes, though not necessarily with the same names.

6.2.6. External

If a name is used as the procedure name in a procedure invocation, it is implicitly declared to have the **external** attribute. If a procedure name is to be passed as an argument, it is necessary to declare it in a statement of the form

external [*name*]

If a name has the external attribute and it is a formal argument of the procedure, then it is associated with a procedure identifier passed as an actual argument at each call. If the name is not a formal argument, then that name is the actual name of a procedure, as it appears in the corresponding **procedure** statement.

6.3. Variable List

The elements of a variable list in a declaration consist of a name, an optional dimension specification, and an optional initial value specification. The name follows the usual rules. The dimension specification is the same form and meaning as the parenthesized list in an **array** attribute. The initial value specification is an equal sign (=) followed by a constant expression. If the name is an array, the right side of the equal sign may be a parenthesized list of constant expressions, or repeated elements or lists; the total number of elements in the list must not exceed the number of elements of the array, which are filled in column-major order.

6.4. The Initial Statement

An initial value may also be specified for a simple variable, array, array element, or member of a structure using a statement of the form

initial [*var = val*]

The *var* may be a variable name, array element specification, or member of structure. The right side follows the same rules as for an initial value specification in other declaration statements.

7. EXECUTABLE STATEMENTS

Every useful EFL program contains executable statements, otherwise it would not do anything and would not need to be run. Statements are frequently made up of other statements. Blocks are the most obvious case, but many other forms contain statements as constituents.

To increase the legibility of EFL programs, some of the statement forms can be broken without an explicit continuation. A square (□) in the syntax represents a point where the end of a line will be ignored.

7.1. Expression Statements

7.1.1. Subroutine Call

A procedure invocation that returns no value is known as a subroutine call. Such an invocation is a statement. Examples are

```
work(in, out)
run()
```

Input/output statements (see Section 7.7) resemble procedure invocations but do not yield a value. If an error occurs the program stops.

7.1.2. Assignment Statements

An expression that is a simple assignment (=) or a compound assignment (+ = etc.) is a statement:

```
a = b
a = sin(x)/6
x *= y
```

7.2. Blocks

A block is a compound statement that acts as a statement. A block begins with a left brace, optionally followed by declarations, optionally followed by executable statements, followed by a right brace. A block may be used anywhere a statement is permitted. A block is not an expression and does not have a value. An example of a block is

```
{
integer i   # this variable is unknown outside the braces
big = 0
do i = 1,n
  if(big < a(i))
    big = a(i)
}
```

7.3. Test Statements

Test statements permit execution of certain statements conditional on the truth of a predicate.

7.3.1. If Statement

The simplest of the test statements is the if statement, of form

```
if ( logical-expression ) □ statement
```

The logical expression is evaluated; if it is true, then the *statement* is executed.

7.3.2. If-Else

A more general statement is of the form

```
if ( logical-expression ) □ statement-1 □ else □ statement-2
```

If the expression is true then *statement-1* is executed, otherwise *statement-2* is executed. Either of the consequent statements may itself be an if-else so a completely nested test sequence is possible:

```

if(x<y)
  if(a<b)
    k = 1
  else
    k = 2
else
  if(a<b)
    m = 1
  else
    m = 2

```

An **else** applies to the nearest preceding un-**else**d **if**. A more common use is as a sequential test:

```

if(x==1)
  k = 1
else if(x==3 | x==5)
  k = 2
else
  k = 3

```

7.3.3. Select Statement

A multiway test on the value of a quantity is succinctly stated as a **select** statement, which has the general form

```
select( expression ) □ block
```

Inside the block two special types of labels are recognized. A prefix of the form

```
case [ constant ] :
```

marks the statement to which control is passed if the expression in the select has a value equal to one of the case constants. If the expression equals none of these constants, but there is a label **default** inside the select, a branch is taken to that point; otherwise the statement following the right brace is executed. Once execution begins at a **case** or **default** label, it continues until the next **case** or **default** is encountered. The **else-if** example above is better written as

```

select(x)
{
  case 1:
    k = 1
  case 3,5:
    k = 2
  default:
    k = 3
}

```

Note that control does not “fall through” to the next case.

7.4. Loops

The loop forms provide the best way of repeating a statement or sequence of operations. The simplest (**while**) form is theoretically sufficient, but it is very convenient to have the more general loops available, since each expresses a mode of control that arises frequently in practice.

7.4.1. While Statement

This construct has the form

while (*logical-expression*) □ *statement*

The expression is evaluated; if it is true, the statement is executed, and then the test is performed again. If the expression is false, execution proceeds to the next statement.

7.5. For Statement

The **for** statement is a more elaborate looping construct. It has the form

for (*initial-statement* , □ *logical-expression* , □ *iteration-statement*) □ *body-statement*

Except for the behavior of the **next** statement (see Section 7.6.3), this construct is equivalent to

```

initial-statement
while ( logical-expression )
{
  body-statement
  iteration-statement
}
```

This form is useful for general arithmetic iterations, and for various pointer-type operations. The sum of the integers from 1 to 100 can be computed by the fragment

```

n = 0
for(i = 1, i <= 100, i += 1)
  n += i
```

Alternatively, the computation could be done by the single statement

```

for( { n = 0 ; i = 1 } , i <= 100 , { n += i ; ++i } )
;
```

Note that the body of the **for** loop is a null statement in this case. An example of following a linked list will be given later.

7.5.1. Repeat Statement

The statement

repeat □ *statement*

executes the *statement*, then does it again, without any termination test. Obviously, a test inside the *statement* is needed to stop the loop.

7.5.2. Repeat ... Until Statement

The **while** loop performs a test before each iteration. The statement

repeat □ *statement* □ **until** (*logical-expression*)

executes the *statement*, then evaluates the logical; if the logical is true the loop is complete; otherwise control returns to the *statement*. Thus, the body is always executed at least once. The **until** refers to the nearest preceding **repeat** that has not been paired with an **until**. In practice, this appears to be the least frequently used looping construct.

7.5.3. Do Loops

The simple arithmetic progression is a very common one in numerical applications. EFL has a special loop form for ranging over an ascending arithmetic sequence

```
do variable = expression-1, expression-2, expression-3
    statement
```

The variable is first given the value *expression-1*. The statement is executed, then *expression-3* is added to the variable. The loop is repeated until the variable exceeds *expression-2*. If *expression-3* and the preceding comma are omitted, the increment is taken to be 1. The loop above is equivalent to

```
t2 = expression-2
t3 = expression-3
for(variable = expression-1 , variable <= t2 , variable += t3)
    statement
```

(The compiler translates EFL **do** statements into Fortran DO statements, which are in turn usually compiled into excellent code.) The **do** *variable* may not be changed inside of the loop, and *expression-1* must not exceed *expression-2*. The sum of the first hundred positive integers could be computed by

```
n = 0
do i = 1, 100
    n += i
```

7.6. Branch Statements

Most of the need for branch statements in programs can be averted by using the loop and test constructs, but there are programs where they are very useful.

7.6.1. Goto Statement

The most general, and most dangerous, branching statement is the simple unconditional

```
goto label
```

After executing this statement, the next statement performed is the one following the given label. Inside of a **select** the case labels of that block may be used as labels, as in the following example:

```

select(k)
{
  case 1:
      error(7)

  case 2:
      k = 2
      goto case 4

  case 3:
      k = 5
      goto case 4

  case 4:
      fixup(k)
      goto default

  default:
      prmsg("ouch")
}

```

(If two **select** statements are nested, the case labels of the outer **select** are not accessible from the inner one.)

7.6.2. Break Statement

A safer statement is one which transfers control to the statement following the current **select** or loop form. A statement of this sort is almost always needed in a **repeat** loop:

```

repeat
{
  do a computation
  if( finished )
    break
}

```

More general forms permit controlling a branch out of more than one construct.

break 3

transfers control to the statement following the third loop and/or **select** surrounding the statement. It is possible to specify which type of construct (**for**, **while**, **repeat**, **do**, or **select**) is to be counted. The statement

break while

breaks out of the first surrounding **while** statement. Either of the statements

break 3 for
break for 3

will transfer to the statement after the third enclosing **for** loop.

7.6.3. Next Statement

The **next** statement causes the first surrounding loop statement to go on to the next iteration: the next operation performed is the test of a **while**, the *iteration-statement* of a **for**, the body of a **repeat**, the test of a **repeat...until**, or the increment of a **do**. Elaborations similar to those for **break** are available:

```

next
next 3
next 3 for
next for 3

```

A **next** statement ignores **select** statements.

7.6.4. Return

The last statement of a procedure is followed by a return of control to the caller. If it is desired to effect such a return from any other point in the procedure, a

```
return
```

statement may be executed. Inside a function procedure, the function value is specified as an argument of the statement:

```
return ( expression )
```

7.7. Input/Output Statements

EFL has two input statements (**read** and **readbin**), two output statements (**write** and **writabin**), and three control statements (**endfile**, **rewind**, and **backspace**). These forms may be used either as a primary with a **integer** value or as a statement. If an exception occurs when one of these forms is used as a statement, the result is undefined but will probably be treated as a fatal error. If they are used in a context where they return a value, they return zero if no exception occurs. For the input forms, a negative value indicates end-of-file and a positive value an error. The input/output part of EFL very strongly reflects the facilities of Fortran.

7.7.1. Input/Output Units

Each I/O statement refers to a "unit", identified by a small positive integer. Two special units are defined by EFL, the *standard input unit* and the *standard output unit*. These particular units are assumed if no unit is specified in an I/O transmission statement.

The data on the unit are organized into *records*. These records may be read or written in a fixed sequence, and each transmission moves an integral number of records. Transmission proceeds from the first record until the *end of file*.

7.7.2. Binary Input/Output

The **readbin** and **writabin** statements transmit data in a machine-dependent but swift manner. The statements are of the form

```

writabin( unit , binary-output-list )
readbin( unit , binary-input-list )

```

Each statement moves one unformatted record between storage and the device. The *unit* is an integer expression. A *binary-output-list* is an *iolist* (see below) without any format specifiers. A *binary-input-list* is an *iolist* without format specifiers in which each of the expressions is a variable name, array element, or structure member.

7.7.3. Formatted Input/Output

The **read** and **write** statements transmit data in the form of lines of characters. Each statement moves one or more records (lines). Numbers are translated into decimal notation. The exact form of the lines is determined by format specifications, whether provided explicitly in the statement or implicitly. The syntax of the statements is

```

write( unit , formatted-output-list )
read( unit , formatted-input-list )

```

The lists are of the same form as for binary I/O, except that the lists may include format specifications. If the *unit* is omitted, the standard input or output unit is used.

7.7.4. Iolists

An *iolist* specifies a set of values to be written or a set of variables into which values are to be read. An *iolist* is a list of one or more *ioexpressions* of the form

```

expression
{ iolist }
do-specification { iolist }

```

For formatted I/O, an *ioexpression* may also have the forms

```

ioexpression : format-specifier
: format-specifier

```

A *do-specification* looks just like a **do** statement, and has a similar effect: the values in the braces are transmitted repeatedly until the **do** execution is complete.

7.7.5. Formats

The following are permissible *format-specifiers*. The quantities *w*, *d*, and *k* must be integer constant expressions.

i (<i>w</i>)	integer with <i>w</i> digits
f (<i>w,d</i>)	floating point number of <i>w</i> characters, <i>d</i> of them to the right of the decimal point.
e (<i>w,d</i>)	floating point number of <i>w</i> characters, <i>d</i> of them to the right of the decimal point, with the exponent field marked with the letter e
l (<i>w</i>)	logical field of width <i>w</i> characters, the first of which is t or f (the rest are blank on output, ignored on input) standing for true and false respectively
c	character string of width equal to the length of the datum
c (<i>w</i>)	character string of width <i>w</i>
s (<i>k</i>)	skip <i>k</i> lines
x (<i>k</i>)	skip <i>k</i> spaces
" ... "	use the characters inside the string as a Fortran format

If no format is specified for an item in a formatted input/output statement, a default form is chosen.

If an item in a list is an array name, then the entire array is transmitted as a sequence of elements, each with its own format. The elements are transmitted in column-major order, the same order used for array initializations.

7.7.6. Manipulation Statements

The three input/output statements

```

backspace(unit)
rewind(unit)
endfile(unit)

```

look like ordinary procedure calls, but may be used either as statements or as integer expressions which yield non-zero if an error is detected. **backspace** causes the specified unit to back

up, so that the next read will re-read the previous record, and the next write will over-write it. **rewind** moves the device to its beginning, so that the next input statement will read the first record. **endfile** causes the file to be marked so that the record most recently written will be the last record on the file, and any attempt to read past is an error.

8. PROCEDURES

Procedures are the basic unit of an EFL program, and provide the means of segmenting a program into separately compilable and named parts.

8.1. Procedure Statement

Each procedure begins with a statement of one of the forms

```

procedure
  attributes procedure procedurename
  attributes procedure procedurename ( )
  attributes procedure procedurename ( [ name ] )

```

The first case specifies the main procedure, where execution begins. In the two other cases, the *attributes* may specify precision and type, or they may be omitted entirely. The precision and type of the procedure may be declared in an ordinary declaration statement. If no type is declared, then the procedure is called a *subroutine* and no value may be returned for it. Otherwise, the procedure is a function and a value of the declared type is returned for each call. Each *name* inside the parentheses in the last form above is called a *formal argument* of the procedure.

8.2. End Statement

Each procedure terminates with a statement

```

end

```

8.3. Argument Association

When a procedure is invoked, the actual arguments are evaluated. If an actual argument is the name of a variable, an array element, or a structure member, that entity becomes associated with the formal argument, and the procedure may reference the values in the object, and assign to it. Otherwise, the value of the actual is associated with the formal argument, but the procedure may not attempt to change the value of that formal argument.

If the value of one of the arguments is changed in the procedure, it is not permitted that the corresponding actual argument be associated with another formal argument or with a **common** element that is referenced in the procedure.

8.4. Execution and Return Values

After actual and formal arguments have been associated, control passes to the first executable statement of the procedure. Control returns to the invoker either when the **end** statement of the procedure is reached or when a **return** statement is executed. If the procedure is a function (has a declared type), and a **return**(*value*) is executed, the value is coerced to the correct type and precision and returned.

8.5. Known Functions

A number of functions are known to EFL, and need not be declared. The compiler knows the types of these functions. Some of them are *generic*; i.e., they name a family of functions that differ in the types of their arguments and return values. The compiler chooses which element of the set to invoke based upon the attributes of the actual arguments.

8.5.1. Minimum and Maximum Functions

The generic functions are **min** and **max**. The **min** calls return the value of their smallest argument; the **max** calls return the value of their largest argument. These are the only functions that may take different numbers of arguments in different calls. If any of the arguments are **long real** then the result is **long real**. Otherwise, if any of the arguments are **real** then the result is **real**; otherwise all the arguments and the result must be **integer**. Examples are

```
min(5, x, -3.20)
max(i, z)
```

8.5.2. Absolute Value

The **abs** function is a generic function that returns the magnitude of its argument. For integer and real arguments the type of the result is identical to the type of the argument; for complex arguments the type of the result is the real of the same precision.

8.5.3. Elementary Functions

The following generic functions take arguments of **real**, **long real**, or **complex** type and return a result of the same type:

sin	sine function
cos	cosine function
exp	exponential function (e^x).
log	natural (base e) logarithm
log10	common (base 10) logarithm
sqrt	square root function (\sqrt{x}).

In addition, the following functions accept only **real** or **long real** arguments:

atan	$\text{atan}(x) = \tan^{-1}x$
atan2	$\text{atan2}(x, y) = \tan^{-1}\frac{x}{y}$

8.5.4. Other Generic Functions

The **sign** functions takes two arguments of identical type; $\text{sign}(x, y) = \text{sgn}(y)|x|$. The **mod** function yields the remainder of its first argument when divided by its second. These functions accept integer and real arguments.

9. ATAVISMS

Certain facilities are included in the EFL language to ease the conversion of old Fortran or Ratfor programs to EFL.

9.1. Escape Lines

In order to make use of nonstandard features of the local Fortran compiler, it is occasionally necessary to pass a particular line through to the EFL compiler output. A line that begins with a percent sign ("%") is copied through to the output, with the percent sign removed but no other change. Inside of a procedure, each escape line is treated as an executable statement. If a sequence of lines constitute a continued Fortran statement, they should be enclosed in braces.

9.2. Call Statement

A subroutine call may be preceded by the keyword **call**.

```
call joe
call work(17)
```

9.3. Obsolete Keywords

The following keywords are recognized as synonyms of EFL keywords:

Fortran	EFL
double precision	long real
function	procedure
subroutine	procedure (<i>untyped</i>)

9.4. Numeric Labels

Standard statement labels are identifiers. A numeric (positive integer constant) label is also permitted; the colon is optional following a numeric label.

9.5. Implicit Declarations

If a name is used but does not appear in a declaration, the EFL compiler gives a warning and assumes a declaration for it. If it is used in the context of a procedure invocation, it is assumed to be a procedure name; otherwise it is assumed to be a local variable defined at nesting level 1 in the current procedure. The assumed type is determined by the first letter of the name. The association of letters and types may be given in an **implicit** statement, with syntax

```
implicit ( letter-list ) type
```

where a *letter-list* is a list of individual letters or ranges (pair of letters separated by a minus sign). If no **implicit** statement appears, the following rules are assumed:

```
implicit (a-h, o-z) real
implicit (i-n) integer
```

9.6. Computed Goto

Fortran contains an indexed multi-way branch; this facility may be used in EFL by the computed GOTO:

```
goto ( [ label ] ), expression
```

The expression must be of type integer and be positive but be no larger than the number of labels in the list. Control is passed to the statement marked by the label whose position in the list is equal to the expression.

9.7. Goto Statement

In unconditional and computed **goto** statements, it is permissible to separate the **go** and **to** words, as in

```
go to xyz
```

9.8. Dot Names

Fortran uses a restricted character set, and represents certain operators by multi-character sequences. There is an option (**dots=on**; see Section 10.2) which forces the compiler to recognize the forms in the second column below:

<	.lt.
<=	.le.
>	.gt.
>=	.ge.
==	.eq.
≠	.ne.
&	.and.
	.or.
&&	.andand.
	.oror.
~	.not.
true	.true.
false	.false.

In this mode, no structure element may be named `lt`, `le`, etc. The readable forms in the left column are always recognized.

9.9. Complex Constants

A complex constant may be written as a parenthesized list of real quantities, such as

`(1.5, 3.0)`

The preferred notation is by a type coercion,

`complex(1.5, 3.0)`

9.10. Function Values

The preferred way to return a value from a function in EFL is the `return(value)` construct. However, the name of the function acts as a variable to which values may be assigned; an ordinary `return` statement returns the last value assigned to that name as the function value.

9.11. Equivalence

A statement of the form

equivalence v_1, v_2, \dots, v_n

declares that each of the v_i starts at the same memory location. Each of the v_i may be a variable name, array element name, or structure member.

9.12. Minimum and Maximum Functions

There are a number of non-generic functions in this category, which differ in the required types of the arguments and the type of the return value. They may also have variable numbers of arguments, but all the arguments must have the same type.

Function	Argument Type	Result Type
amin0	integer	real
amin1	real	real
min0	integer	integer
min1	real	integer
dmin1	long real	long real
amax0	integer	real
amax1	real	real
max0	integer	integer
max1	real	integer
dmax1	long real	long real

10. COMPILER OPTIONS

A number of options can be used to control the output and to tailor it for various compilers and systems. The defaults chosen are conservative, but it is sometimes necessary to change the output to match peculiarities of the target environment.

Options are set with statements of the form

option [*opt*]

where each *opt* is of one of the forms

optionname
optionname = *optionvalue*

The *optionvalue* is either a constant (numeric or string) or a name associated with that option. The two names **yes** and **no** apply to a number of options.

10.1. Default Options

Each option has a default setting. It is possible to change the whole set of defaults to those appropriate for a particular environment by using the **system** option. At present, the only valid values are **system=unix** and **system=gcos**.

10.2. Input Language Options

The **dots** option determines whether the compiler recognizes **.lt.** and similar forms. The default setting is **no**.

10.3. Input/Output Error Handling

The **ioerror** option can be given three values: **none** means that none of the I/O statements may be used in expressions, since there is no way to detect errors. The implementation of the **ibm** form uses **ERR=** and **END=** clauses. The implementation of the **fortran77** form uses **IOSTAT=** clauses.

10.4. Continuation Conventions

By default, continued Fortran statements are indicated by a character in column 6 (Standard Fortran). The option **continue=column1** puts an ampersand (&) in the first column of the continued lines instead.

10.5. Default Formats

If no format is specified for a datum in an iolist for a **read** or **write** statement, a default is provided. The default formats can be changed by setting certain options

<u>Option</u>	<u>Type</u>
iformat	integer
rformat	real
dformat	long real
zformat	complex
zdformat	long complex
lformat	logical

The associated value must be a Fortran format, such as

option rformat=f22.6

10.6. Alignments and Sizes

In order to implement **character** variables, structures, and the **sizeof** and **lengthof** operators, it is necessary to know how much space various Fortran data types require, and what boundary alignment properties they demand. The relevant options are

<u>Fortran Type</u>	<u>Size Option</u>	<u>Alignment Option</u>
integer	isize	ialign
real	rsize	ralign
long real	dsize	dalign
complex	zsize	zalign
logical	lsize	lalign

The sizes are given in terms of an arbitrary unit; the alignment is given in the same units. The option **charperint** gives the number of characters per **integer** variable.

10.7. Default Input/Output Units

The options **ftnin** and **ftnout** are the numbers of the standard input and output units. The default values are **ftnin=5** and **ftnout=6**.

10.8. Miscellaneous Output Control Options

Each Fortran procedure generated by the compiler will be preceded by the value of the **prochheader** option.

No Hollerith strings will be passed as subroutine arguments if **hollincall=no** is specified.

The Fortran statement numbers normally start at 1 and increase by 1. It is possible to change the increment value by using the **deltastno** option.

11. EXAMPLES

In order to show the flavor or programming in EFL, we present a few examples. They are short, but show some of the convenience of the language.

11.1. File Copying

The following short program copies the standard input to the standard output, provided that the input is a formatted file containing lines no longer than a hundred characters.

```

procedure # main program
character(100) line

while( read( , line) == 0 )
    write( , line)
end

```

Since **read** returns zero until the end of file (or a read error), this program keeps reading and writing until the input is exhausted.

11.2. Matrix Multiplication

The following procedure multiplies the $m \times n$ matrix **a** by the $n \times p$ matrix **b** to give the $m \times p$ matrix **c**. The calculation obeys the formula $c_{ij} = \sum a_{ik} b_{kj}$.

```

procedure matmul(a,b,c, m,n,p)
integer i, j, k, m, n, p
long real a(m,n), b(n,p), c(m,p)
do i = 1,m
do j = 1,p
    {
        c(i,j) = 0
        do k = 1,n
            c(i,j) += a(i,k) * b(k,j)
        }
end

```

11.3. Searching a Linked List

Assume we have a list of pairs of numbers (x,y) . The list is stored as a linked list sorted in ascending order of x values. The following procedure searches this list for a particular value of x and returns the corresponding y value.

```

define LAST      0
define NOTFOUND -1

integer procedure val(list, first, x)

# list is an array of structures.
# Each structure contains a thread index value, an x, and a y value.
struct
    {
        integer nextindex
        integer x, y
    } list(*)

integer first, p, arg

for(p = first, p~=LAST && list(p).x<=x, p = list(p).nextindex)
    if(list(p).x == x)
        return( list(p).y )

return(NOTFOUND)
end

```

The search is a single **for** loop that begins with the head of the list and examines items until either the list is exhausted ($p = \text{LAST}$) or until it is known that the specified value is not on

the list ($\text{list}(p).x > x$). The two tests in the conjunction must be performed in the specified order to avoid using an invalid subscript in the $\text{list}(p)$ reference. Therefore, the $\&\&$ operator is used. The next element in the chain is found by the iteration statement $p = \text{list}(p).\text{nextindex}$.

11.4. Walking a Tree

As an example of a more complicated problem, let us imagine we have an expression tree stored in a common area, and that we want to print out an infix form of the tree. Each node is either a leaf (containing a numeric value) or it is a binary operator, pointing to a left and a right descendant. In a recursive language, such a tree walk would be implemented by the following simple pseudocode:

```

if this node is a leaf
    print its value
otherwise
    print a left parenthesis
    print the left node
    print the operator
    print the right node
    print a right parenthesis

```

In a nonrecursive language like EFL, it is necessary to maintain an explicit stack to keep track of the current state of the computation. The following procedure calls a procedure `outch` to print a single character and a procedure `outval` to print a value.

```

procedure walk(first) # print out an expression tree
integer first # index of root node
integer currentnode
integer stackdepth
common(nodes) struct
    {
        character(1) op
        integer leftp, rightp
        real val
    } tree(100) # array of structures

struct
    {
        integer nextstate
        integer nodep
    } stackframe(100)

define NODE tree(currentnode)
define STACK stackframe(stackdepth)

# nextstate values
define DOWN 1
define LEFT 2
define RIGHT 3

# initialize stack with root node
stackdepth = 1
STACK.nextstate = DOWN
STACK.nodep = first

```

```

while( stackdepth > 0 )
  {
  currentnode = STACK.nodep
  select(STACK.nextstate)
  {
  case DOWN:
    if(NODE.op == " ") # a leaf
      {
      outval( NODE.val )
      stackdepth -= 1
      }
    else { # a binary operator node
      outch( "(" )
      STACK.nextstate = LEFT
      stackdepth += 1
      STACK.nextstate = DOWN
      STACK.nodep = NODE.leftp
      }
  case LEFT:
    outch( NODE.op )
    STACK.nextstate = RIGHT
    stackdepth += 1
    STACK.nextstate = DOWN
    STACK.nodep = NODE.rightp
  case RIGHT:
    outch( ")" )
    stackdepth -= 1
  }
  }
end

```

12. PORTABILITY

One of the major goals of the EFL language is to make it easy to write portable programs. The output of the EFL compiler is intended to be acceptable to any Standard Fortran compiler (unless the **fortran77** option is specified).

12.1. Primitives

Certain EFL operations cannot be implemented in portable Fortran, so a few machine-dependent procedures must be provided in each environment.

12.1.1. Character String Copying

The subroutine **eflasc** is called to copy one character string to another. If the target string is shorter than the source, the final characters are not copied. If the target string is longer, its end is padded with blanks. The calling sequence is

```

subroutine eflasc(a, la, b, lb)
integer a(*), la, b(*), lb

```

and it must copy the first **lb** characters from **b** to the first **la** characters of **a**.

12.1.2. Character String Comparisons

The function `eflcmc` is invoked to determine the order of two character strings. The declaration is

```
integer function eflcmc(a, la, b, lb)
integer a(*), la, b(*), lb
```

The function returns a negative value if the string `a` of length `la` precedes the string `b` of length `lb`. It returns zero if the strings are equal, and a positive value otherwise. If the strings are of differing length, the comparison is carried out as if the end of the shorter string were padded with blanks.

13. ACKNOWLEDGEMENTS

A. D. Hall originated the EFL language and wrote the first compiler for it; he also gave inestimable aid when I took up the project. B. W. Kernighan and W. S. Brown made a number of useful suggestions about the language and about this report. N. L. Schryer has acted as willing, cheerful, and severe first user and helpful critic of each new version and facility. J. L. Blue, L. C. Kaufman, and D. D. Warner made very useful contributions by making serious use of the compiler, and noting and tolerating its misbehaviors.

14. REFERENCE

- [1] B. W. Kernighan. *RATFOR—A Preprocessor for a Rational FORTRAN*, Bell Laboratories.

APPENDIX A. RELATION BETWEEN EFL AND RATFOR

There are a number of differences between Ratfor and EFL, since EFL is a defined language while Ratfor is the union of the special control structures and the language accepted by the underlying Fortran compiler. Ratfor running over Standard Fortran is almost a subset of EFL. Most of the features described in the Atavisms section are present to ease the conversion of Ratfor programs to EFL.

There are a few incompatibilities: The syntax of the **for** statement is slightly different in the two languages: the three clauses are separated by semicolons in Ratfor, but by commas in EFL. (The initial and iteration statements may be compound statements in EFL because of this change). The input/output syntax is quite different in the two languages, and there is no **FOR-MAT** statement in EFL. There are no **ASSIGN** or assigned **GOTO** statements in EFL.

The major linguistic additions are character data, factored declaration syntax, block structure, assignment and sequential test operators, generic functions, and data structures. EFL permits more general forms for expressions, and provides a more uniform syntax. (One need not worry about the Fortran/Ratfor restrictions on subscript or **DO** expression forms, for example.)

APPENDIX B. COMPILER

B.1. Current Version

The current version of the EFL compiler is a two-pass translator written in portable C. It implements all of the features of the language described above except for **long complex** numbers. Versions of this compiler run under the GCOS and UNIX† operating systems.

B.2. Diagnostics

The EFL compiler diagnoses all syntax errors. It gives the line and file name (if known) on which the error was detected. Warnings are given for variables that are used but not explicitly declared.

B.3. Quality of Fortran Produced

The Fortran produced by EFL is quite clean and readable. To the extent possible, the variable names that appear in the EFL program are used in the Fortran code. The bodies of loops and test constructs are indented. Statement numbers are consecutive. Few unneeded GOTO and CONTINUE statements are used. It is considered a compiler bug if incorrect Fortran is produced (except for escaped lines). The following is the Fortran procedure produced by the EFL compiler for the matrix multiplication example (Section 11.2):

```

subroutine matmul(a, b, c, m, n, p)
integer m, n, p
double precision a(m, n), b(n, p), c(m, p)
integer i, j, k
do 3 i = 1, m
  do 2 j = 1, p
    c(i, j) = 0
    do 1 k = 1, n
      c(i, j) = c(i, j) + a(i, k) * b(k, j)
1      continue
2      continue
3      continue
end

```

† UNIX is a trademark of Bell Laboratories.

The following is the procedure for the tree walk (Section 11.4):

```

subroutine walk(first)
integer first
common /nodes/ tree
integer tree(4, 100)
real tree1(4, 100)
integer staame(2, 100), staph, curode
integer const1(1)
equivalence (tree(1,1), tree1(1,1))
data const1(1)/4h /
c print out an expression tree
c index of root node
c array of structures
c nextstate values
c initialize stack with root node
  staph = 1
  staame(1, staph) = 1
  staame(2, staph) = first
1  if (staph .le. 0) goto 9
    curode = staame(2, staph)
    goto 7
2  if (tree(1, curode) .ne. const1(1)) goto 3
    call outval(tree1(4, curode))
c a leaf
  staph = staph-1
  goto 4
3  call outch(1h())
c a binary operator node
  staame(1, staph) = 2
  staph = staph+1
  staame(1, staph) = 1
  staame(2, staph) = tree(2, curode)
4  goto 8
5  call outch(tree(1, curode))
  staame(1, staph) = 3
  staph = staph+1
  staame(1, staph) = 1
  staame(2, staph) = tree(3, curode)
  goto 8
6  call outch(1h))
  staph = staph-1
  goto 8
7  if (staame(1, staph) .eq. 3) goto 6
  if (staame(1, staph) .eq. 2) goto 5
  if (staame(1, staph) .eq. 1) goto 2
8  continue
  goto 1
9  continue
end-
```

APPENDIX C. CONSTRAINTS ON THE DESIGN OF THE EFL LANGUAGE

Although Fortran can be used to simulate any finite computation, there are realistic limits on the generality of a language that can be translated into Fortran. The design of EFL was constrained by the implementation strategy. Certain of the restrictions are petty (six character external names), but others are sweeping (lack of pointer variables). The following paragraphs describe the major limitations imposed by Fortran.

C.1. External Names

External names (procedure and COMMON block names) must be no longer than six characters in Fortran. Further, an external name is global to the entire program. Therefore, EFL can support block structure within a procedure, but it can have only one level of external name if the EFL procedures are to be compilable separately, as are Fortran procedures.

C.2. Procedure Interface

The Fortran standards, in effect, permit arguments to be passed between Fortran procedures either by reference or by copy-in/copy-out. This indeterminacy of specification shows through into EFL. A program that depends on the method of argument transmission is illegal in either language.

There are no procedure-valued variables in Fortran: a procedure name may only be passed as an argument or be invoked; it cannot be stored. Fortran (and EFL) would be noticeably simpler if a procedure variable mechanism were available.

C.3. Pointers

The most grievous problem with Fortran is its lack of a pointer-like data type. The implementation of the compiler would have been far easier if certain hard cases could have been handled by pointers. Further, the language could have been simplified considerably if pointers were accessible in Fortran. (There are several ways of simulating pointers by using subscripts, but they founder on the problems of external variables and initialization.)

C.4. Recursion

Fortran procedures are not recursive, so it was not practical to permit EFL procedures to be recursive. (Recursive procedures with arguments can be simulated only with great pain.)

C.5. Storage Allocation

The definition of Fortran does not specify the lifetime of variables. It would be possible but cumbersome to implement stack or heap storage disciplines by using COMMON blocks.

January 1981

UNIX Programming (Second Edition)

*Brian W. Kernighan**Dennis M. Ritchie*Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This paper is an introduction to programming on the UNIX[†] system. The emphasis is on how to write programs that interface to the operating system, either directly or through the standard I/O library. The topics discussed include

- handling command arguments
- rudimentary I/O; the standard input and output
- the standard I/O library; file system access
- low-level I/O: open, read, write, close, seek
- processes: exec, fork, pipes
- signals—interrupts, etc.

There is also an appendix that describes the standard I/O library in detail.

1. INTRODUCTION

This paper describes how to write programs that interface with the UNIX operating system in a non-trivial way. This includes programs that use files by name, that use pipes, that invoke other commands as they run, or that attempt to catch interrupts and other signals during execution.

This document collects material which is scattered throughout several sections of the *UNIX User's Manual* [1]. There is no attempt to be complete; only generally useful material is dealt with. It is assumed that you will be programming in C, so you must be able to read the language roughly up to the level of *The C Programming Language* [2]. Some of the material in sections 2 through 4 is based on topics covered more carefully there. You should also be familiar with UNIX itself at least to the level of *UNIX for Beginners* [3].

2. BASICS

2.1 Program Arguments

When a C program is run as a command, the arguments on the command line are made available to the function `main` as an argument count `argc` and an array `argv` of pointers to character strings that contain the arguments. By convention, `argv[0]` is the command name itself, so `argc` is always greater than 0.

The following program illustrates the mechanism: it simply echoes its arguments back to the terminal. (This is essentially the `echo` command.)

[†] UNIX is a trademark of Bell Laboratories.

```

main(argc, argv)    /* echo arguments */
int argc;
char *argv[];
{
    int i;
    for (i = 1; i < argc; i++)
        printf("%s%c", argv[i], (i<argc-1) ? ' ' : '\n');
}

```

`argv` is a pointer to an array whose individual elements are pointers to arrays of characters; each is terminated by `\0`, so they can be treated as strings. The program starts by printing `argv[1]` and loops until it has printed them all.

The argument count and the arguments are parameters to `main`. If you want to keep them around so other routines can get at them, you must copy them to external variables.

2.2 The "Standard Input" and "Standard Output"

The simplest input mechanism is to read the "standard input," which is generally the user's terminal. The function `getchar` returns the next input character each time it is called. A file may be substituted for the terminal by using the `<` convention: if `prog` uses `getchar`, then the command line

```
prog <file
```

causes `prog` to read `file` instead of the terminal. `prog` itself need know nothing about where its input is coming from. This is also true if the input comes from another program via the pipe mechanism:

```
otherprog | prog
```

provides the standard input for `prog` from the standard output of `otherprog`.

`getchar` returns the value `EOF` when it encounters the end of file (or an error) on whatever you are reading. The value of `EOF` is normally defined to be `-1`, but it is unwise to take any advantage of that knowledge. As will become clear shortly, this value is automatically defined for you when you compile a program, and need not be of any concern.

Similarly, `putchar(c)` puts the character `c` on the "standard output," which is also by default the terminal. The output can be captured on a file by using `>`: if `prog` uses `putchar`,

```
prog >outfile
```

writes the standard output on `outfile` instead of the terminal. `outfile` is created if it doesn't exist; if it already exists, its previous contents are overwritten. And a pipe can be used:

```
prog | otherprog
```

puts the standard output of `prog` into the standard input of `otherprog`.

The function `printf`, which formats output in various ways, uses the same mechanism as `putchar` does, so calls to `printf` and `putchar` may be intermixed in any order; the output will appear in the order of the calls.

Similarly, the function `scanf` provides for formatted input conversion; it will read the standard input and break it up into strings, numbers, etc., as desired. `scanf` uses the same mechanism as `getchar`, so calls to them may also be intermixed.

Many programs read only one input and write one output; for such programs I/O with `getchar`, `putchar`, `scanf`, and `printf` may be entirely adequate, and it is almost always enough to get started. This is particularly true if the UNIX pipe facility is used to connect the output of one program to the input of the next. For example, the following program strips out all ASCII control characters from its input (except for new-line and tab).

```

#include <stdio.h>

main()    /* ccstrip: strip non-graphic characters */
{
    int c;
    while ((c = getchar()) != EOF)
        if ((c >= ' ' && c < 0177) || c == '\t' || c == '\n')
            putchar(c);
    exit(0);
}

```

The line

```
#include <stdio.h>
```

should appear at the beginning of each source file. It causes the C compiler to read a file (*/usr/include/stdio.h*) of standard routines and symbols that includes the definition of `EOF`.

If it is necessary to treat multiple files, you can use `cat` to collect the files for you:

```
cat file1 file2 ... | ccstrip >output
```

and thus avoid learning how to access files from a program. By the way, the call to `exit` at the end is not necessary to make the program work properly, but it assures that any caller of the program will see a normal termination status (conventionally 0) from the program when it completes. Section 6 discusses status returns in more detail.

3. THE STANDARD I/O LIBRARY

The “Standard I/O Library” is a collection of routines intended to provide efficient and portable I/O services for most C programs. The standard I/O library is available on each system that supports C, so programs that confine their system interactions to its facilities can be transported from one system to another essentially without change.

In this section, we will discuss the basics of the standard I/O library. The appendix contains a more complete description of its capabilities.

3.1 File Access

The programs written so far have all read the standard input and written the standard output, which we have assumed are magically pre-defined. The next step is to write a program that accesses a file that is *not* already connected to the program. One simple example is `wc`, which counts the lines, words and characters in a set of files. For instance, the command

```
wc x.c y.c
```

prints the number of lines, words and characters in `x.c` and `y.c` and the totals.

The question is how to arrange for the named files to be read—that is, how to connect the file system names to the I/O statements which actually read the data.

The rules are simple. Before it can be read or written a file has to be *opened* by the standard library function `fopen`. `fopen` takes an external name (like `x.c` or `y.c`), does some housekeeping and negotiation with the operating system, and returns an internal name which must be used in subsequent reads or writes of the file.

This internal name is actually a pointer, called a *file pointer*, to a structure which contains information about the file, such as the location of a buffer, the current character position in the buffer, whether the file is being read or written, and the like. Users don’t need to know the details, because part of the standard I/O definitions obtained by including `stdio.h` is a structure definition called `FILE`. The only declaration needed for a file pointer is exemplified by

```
FILE *fp, *fopen();
```

This says that `fp` is a pointer to a `FILE`, and `fopen` returns a pointer to a `FILE`. (`FILE` is a type name, like `int`, not a structure tag.)

The actual call to `fopen` in a program is

```
fp = fopen(name, mode);
```

The first argument of `fopen` is the name of the file, as a character string. The second argument is the mode, also as a character string, which indicates how you intend to use the file. The only allowable modes are read ("`r`"), write ("`w`"), or append ("`a`").

If a file that you open for writing or appending does not exist, it is created (if possible). Opening an existing file for writing causes the old contents to be discarded. Trying to read a file that does not exist is an error, and there may be other causes of error as well (like trying to read a file when you don't have permission). If there is any error, `fopen` will return the null pointer value `NULL` (which is defined as zero in `stdio.h`).

The next thing needed is a way to read or write the file once it is open. There are several possibilities, of which `getc` and `putc` are the simplest. `getc` returns the next character from a file; it needs the file pointer to tell it what file. Thus

```
c = getc(fp)
```

places in `c` the next character from the file referred to by `fp`; it returns `EOF` when it reaches end of file. `putc` is the inverse of `getc`:

```
putc(c, fp)
```

puts the character `c` on the file `fp` and returns `c`. `getc` and `putc` return `EOF` on error.

When a program is started, three files are opened automatically, and file pointers are provided for them. These files are the standard input, the standard output, and the standard error output; the corresponding file pointers are called `stdin`, `stdout`, and `stderr`. Normally these are all connected to the terminal, but may be redirected to files or pipes as described in Section 2.2. `stdin`, `stdout` and `stderr` are pre-defined in the I/O library as the standard input, output and error files; they may be used anywhere an object of type `FILE *` can be. They are constants, however, *not* variables, so don't try to assign to them.

With some of the preliminaries out of the way, we can now write `wc`. The basic design is one that has been found convenient for many programs: if there are command-line arguments, they are processed in order. If there are no arguments, the standard input is processed. This way the program can be used stand-alone or as part of a larger process.

```

#include <stdio.h>

main(argc, argv)    /* wc: count lines, words, chars */
int argc;
char *argv[];
{
    int c, i, inword;
    FILE *fp, *fopen();
    long linect, wordct, charct;
    long tlinect = 0, twordct = 0, tcharct = 0;

    i = 1;
    fp = stdin;
    do {
        if (argc > 1 && (fp=fopen(argv[i], "r")) == NULL) {
            fprintf(stderr, "wc: can't open %s\n", argv[i]);
            continue;
        }
        linect = wordct = charct = inword = 0;
        while ((c = getc(fp)) != EOF) {
            charct++;
            if (c == '\n')
                linect++;
            if (c == ' ' || c == '\t' || c == '\n')
                inword = 0;
            else if (inword == 0) {
                inword = 1;
                wordct++;
            }
        }
        printf("%7ld %7ld %7ld", linect, wordct, charct);
        printf(argc > 1 ? " %s\n" : "\n", argv[i]);
        fclose(fp);
        tlinect += linect;
        twordct += wordct;
        tcharct += charct;
    } while (++i < argc);
    if (argc > 2)
        printf("%7ld %7ld %7ld total\n", tlinect, twordct,
            tcharct);
    exit(0);
}

```

The function `fprintf` is identical to `printf`, save that the first argument is a file pointer that specifies the file to be written.

The function `fclose` is the inverse of `fopen`; it breaks the connection between the file pointer and the external name that was established by `fopen`, freeing the file pointer for another file. Since there is a limit on the number of files that a program may have open simultaneously, it's a good idea to free things when they are no longer needed. There is also another reason to call `fclose` on an output file—it flushes the buffer in which `putc` is collecting output. (`fclose` is called automatically for each open file when a program terminates normally.)

3.2 Error Handling—Stderr and Exit

`stderr` is assigned to a program in the same way that `stdin` and `stdout` are. Output written on `stderr` appears on the user's terminal even if the standard output is redirected. `wc` writes its diagnostics on `stderr` instead of `stdout` so that if one of the files can't be accessed for some reason, the message finds its way to the user's terminal instead of disappearing down a pipeline or into an output file.

The program actually signals errors in another way, using the function `exit` to terminate program execution. The argument of `exit` is available to whatever process called it (see Section 6), so the success or failure of the program can be tested by another program that uses this one as a sub-process. By convention, a return value of 0 signals that all is well; non-zero values signal abnormal situations.

`exit` itself calls `fclose` for each open output file, to flush out any buffered output, then calls a routine named `_exit`. The function `_exit` causes immediate termination without any buffer flushing; it may be called directly if desired.

3.3 Miscellaneous I/O Functions

The standard I/O library provides several other I/O functions besides those we have illustrated above.

Normally output with `putc`, etc., is buffered (except to `stderr`); to force it out immediately, use `fflush(fp)`.

`fscanf` is identical to `scanf`, except that its first argument is a file pointer (as with `fprintf`) that specifies the file from which the input comes; it returns EOF at end of file.

The functions `sscanf` and `sprintf` are identical to `fscanf` and `fprintf`, except that the first argument names a character string instead of a file pointer. The conversion is done from the string for `sscanf` and into it for `sprintf`.

`fgets(buf, size, fp)` copies the next line from `fp`, up to and including a new-line, into `buf`; at most `size-1` characters are copied; it returns NULL at end of file. `fputs(buf, fp)` writes the string in `buf` onto file `fp`.

The function `ungetc(c, fp)` "pushes back" the character `c` onto the input stream `fp`; a subsequent call to `getc`, `fscanf`, etc., will encounter `c`. Only one character of push-back per file is permitted.

4. LOW-LEVEL I/O

This section describes the bottom level of I/O on the UNIX system. The lowest level of I/O in UNIX provides no buffering or any other services; it is in fact a direct entry into the operating system. You are entirely on your own, but on the other hand, you have the most control over what happens. And since the calls and usage are quite simple, this isn't as bad as it sounds.

4.1 File Descriptors

In the UNIX operating system, all input and output is done by reading or writing files, because all peripheral devices, even the user's terminal, are files in the file system. This means that a single, homogeneous interface handles all communication between a program and peripheral devices.

In the most general case, before reading or writing a file, it is necessary to inform the system of your intent to do so, a process called "opening" the file. If you are going to write on a file, it may also be necessary to create it. The system checks your right to do so (does the file exist? do you have permission to access it?), and if all is well, returns a small positive integer called a *file descriptor*. Whenever I/O is to be done on the file, the file descriptor is used instead of the name to identify the file. (This is roughly analogous to the use of `READ(5, ...)` and `WRITE(6, ...)` in Fortran.) All information about an open file is maintained by the system; the user program refers to the file only by the file descriptor.

The file pointers discussed in section 3 are similar in spirit to file descriptors, but file descriptors are more fundamental. A file pointer is a pointer to a structure that contains, among other things, the file descriptor for the file in question.

Since input and output involving the user's terminal are so common, special arrangements exist to make this convenient. When the command interpreter (the "shell") runs a program, it opens three files, with file descriptors 0, 1, and 2, called the standard input, the standard output, and the standard error output. All of these are normally connected to the terminal, so if a program reads file descriptor 0 and writes file descriptors 1 and 2, it can do terminal I/O without worrying about opening the files.

If I/O is redirected to and from files with `<` and `>`, as in

```
prog <infile >outfile
```

the shell changes the default assignments for file descriptors 0 and 1 from the terminal to the named files. Similar observations hold if the input or output is associated with a pipe. Normally file descriptor 2 remains attached to the terminal, so error messages can go there. In all cases, the file assignments are changed by the shell, not by the program. The program does not need to know where its input comes from nor where its output goes, so long as it uses file 0 for input and 1 and 2 for output.

4.2 Read and Write

All input and output is done by two functions called `read` and `write`. For both, the first argument is a file descriptor. The second argument is a buffer in your program where the data is to come from or go to. The third argument is the number of bytes to be transferred. The calls are

```
n_read = read(fd, buf, n);
n_written = write(fd, buf, n);
```

Each call returns a byte count which is the number of bytes actually transferred. On reading, the number of bytes returned may be less than the number asked for, because fewer than `n` bytes remained to be read. (When the file is a terminal, `read` normally reads only up to the next new-line, which is generally less than what was requested.) A return value of zero bytes implies end of file, and `-1` indicates an error of some sort. For writing, the returned value is the number of bytes actually written; it is generally an error if this isn't equal to the number supposed to be written.

The number of bytes to be read or written is quite arbitrary. The two most common values are 1, which means one character at a time ("unbuffered"), and 512, which corresponds to a physical block size on many peripheral devices. This latter size will be most efficient, but even character at a time I/O is not inordinately expensive.

Putting these facts together, we can write a simple program to copy its input to its output. This program will copy anything to anything, since the input and output can be redirected to any file or device.

```
#define BUFSIZE 512 /* best size for PDP-11 UNIX */
main() /* copy input to output */
{
    char buf[BUFSIZE];
    int n;
    while ((n = read(0, buf, BUFSIZE)) > 0)
        write(1, buf, n);
    exit(0);
}
```

If the file size is not a multiple of `BUFSIZE`, some `read` will return a smaller number of bytes to be written by `write`; the next call to `read` after that will return zero.

It is instructive to see how `read` and `write` can be used to construct higher level routines like `getchar`, `putchar`, etc. For example, here is a version of `getchar` which does unbuffered input.

```
#define CMASK    0377 /* for making char's > 0 */
getchar() /* unbuffered single character input */
{
    char c;
    return((read(0, &c, 1) > 0) ? c & CMASK : EOF);
}
```

`c` *must* be declared `char`, because `read` accepts a character pointer. The character being returned must be masked with `0377` to ensure that it is positive; otherwise sign extension may make it negative. (The constant `0377` is appropriate for the PDP-11 but not necessarily for other machines.)

The second version of `getchar` does input in big chunks, and hands out the characters one at a time.

```
#define CMASK    0377 /* for making char's > 0 */
#define BUFSIZE  512
getchar() /* buffered version */
{
    static char    buf[BUFSIZE];
    static char    *bufp = buf;
    static int     n = 0;
    if (n == 0) { /* buffer is empty */
        n = read(0, buf, BUFSIZE);
        bufp = buf;
    }
    return((--n >= 0) ? *bufp++ & CMASK : EOF);
}
```

4.3 Open, Creat, Close, Unlink

Other than the default standard input, output and error files, you must explicitly open files in order to read or write them. There are two system entry points for this, `open` and `creat` [sic].

`open` is rather like the `fopen` discussed in the previous section, except that instead of returning a file pointer, it returns a file descriptor, which is just an `int`.

```
int fd;
fd = open(name, rwmode);
```

As with `fopen`, the `name` argument is a character string corresponding to the external file name. The access mode argument is different, however: `rwmode` is 0 for read, 1 for write, and 2 for read and write access. `open` returns `-1` if any error occurs; otherwise it returns a valid file descriptor.

It is an error to try to open a file that does not exist. The entry point `creat` is provided to create new files, or to re-write old ones.

```
fd = creat(name, pmode);
```

returns a file descriptor if it was able to create the file called `name`, and `-1` if not. If the file already exists, `creat` will truncate it to zero length; it is not an error to `creat` a file that already exists.

If the file is brand new, `creat` creates it with the *protection mode* specified by the `pmode` argument. In the UNIX file system, there are nine bits of protection information associated with a file, controlling read, write and execute permission for the owner of the file, for the owner's group, and for all others. Thus a three-digit octal number is most convenient for specifying the permissions. For example, 0755 specifies read, write and execute permission for the owner, and read and execute permission for the group and everyone else.

To illustrate, here is a simplified version of the UNIX utility `cp`, a program which copies one file to another. (The main simplification is that our version copies only one file, and does not permit the second argument to be a directory.)

```
#define NULL 0
#define BUFSIZE 512
#define PMODE 0644 /* RW for owner, R for group, others */

main(argc, argv) /* cp: copy f1 to f2 */
int argc;
char *argv[];
{
    int f1, f2, n;
    char buf[BUFSIZE];

    if (argc != 3)
        error("Usage: cp from to", NULL);
    if ((f1 = open(argv[1], 0)) == -1)
        error("cp: can't open %s", argv[1]);
    if ((f2 = creat(argv[2], PMODE)) == -1)
        error("cp: can't create %s", argv[2]);

    while ((n = read(f1, buf, BUFSIZE)) > 0)
        if (write(f2, buf, n) != n)
            error("cp: write error", NULL);
    exit(0);
}

error(s1, s2) /* print error message and die */
char *s1, *s2;
{
    printf(s1, s2);
    printf("\n");
    exit(1);
}
```

As we said earlier, there is a limit (typically 15-25) on the number of files which a program may have open simultaneously. Accordingly, any program which intends to process many files must be prepared to re-use file descriptors. The routine `close` breaks the connection between a file descriptor and an open file, and frees the file descriptor for use with some other file. Termination of a program via `exit` or return from the main program closes all open files.

The function `unlink(filename)` removes the file `filename` from the file system.

4.4 Random Access—Seek and Lseek

File I/O is normally sequential: each `read` or `write` takes place at a position in the file right after the previous one. When necessary, however, a file can be read or written in any arbitrary order. The system call `lseek` provides a way to move around in a file without actually reading or writing:

```
lseek(fd, offset, origin);
```

forces the current position in the file whose descriptor is `fd` to move to position `offset`, which is taken relative to the location specified by `origin`. Subsequent reading or writing will

begin at that position. `offset` is a `long`; `fd` and `origin` are `int`'s. `origin` can be 0, 1, or 2 to specify that `offset` is to be measured from the beginning, from the current position, or from the end of the file respectively. For example, to append to a file, seek to the end before writing:

```
lseek(fd, 0L, 2);
```

To get back to the beginning ("rewind"),

```
lseek(fd, 0L, 0);
```

Notice the `0L` argument; it could also be written as `(long) 0`.

With `lseek`, it is possible to treat files more or less like large arrays, at the price of slower access. For example, the following simple function reads any number of bytes from any arbitrary place in a file.

```
get(fd, pos, buf, n) /* read n bytes from position pos */
int fd, n;
long pos;
char *buf;
{
    lseek(fd, pos, 0); /* get to pos */
    return(read(fd, buf, n));
}
```

Before Version 7, the basic entry point to the UNIX I/O system was called `seek`. `seek` is identical to `lseek`, except that its `offset` argument is an `int` rather than a `long`. Accordingly, since PDP-11 integers have only 16 bits, the `offset` specified for `seek` is limited to 65,535; for this reason, `origin` values of 3, 4, 5 cause `seek` to multiply the given offset by 512 (the number of bytes in one physical block) and then interpret `origin` as if it were 0, 1, or 2 respectively. Thus to get to an arbitrary place in a large file requires two seeks, first one which selects the block, then one which has `origin` equal to 1 and moves to the desired byte within the block.

4.5 Error Processing

The routines discussed in this section, and in fact all the routines which are direct entries into the system can incur errors. Usually they indicate an error by returning a value of `-1`. Sometimes it is nice to know what sort of error occurred; for this purpose all these routines, when appropriate, leave an error number in the external cell `errno`. The meanings of the various error numbers are listed in the introduction to Section 2 of the *UNIX User's Manual* [1], so your program can, for example, determine if an attempt to open a file failed because it did not exist or because the user lacked permission to read it. Perhaps more commonly, you may want to print out the reason for failure. The routine `perror` will print a message associated with the value of `errno`; more generally, `sys_errno` is an array of character strings which can be indexed by `errno` and printed by your program.

5. PROCESSES

It is often easier to use a program written by someone else than to invent one's own. This section describes how to execute a program from within another.

5.1 The "System" Function

The easiest way to execute a program from another is to use the standard library routine `system`. `system` takes one argument, a command string exactly as typed at the terminal (except for the new-line at the end) and executes it. For instance, to time-stamp the output of a program,

```

main()
{
    system("date");
    /* rest of processing */
}

```

If the command string has to be built from pieces, the in-memory formatting capabilities of `sprintf` may be useful.

Remember that `getc` and `putc` normally buffer their input; terminal I/O will not be properly synchronized unless this buffering is defeated. For output, use `fflush`; for input, see `setbuf` in the appendix.

5.2 Low-Level Process Creation—`execl` and `execv`

If you're not using the standard library, or if you need finer control over what happens, you will have to construct calls to other programs using the more primitive routines that the standard library's `system` routine is based on.

The most basic operation is to execute another program *without returning*, by using the routine `execl`. To print the date as the last action of a running program, use

```
execl("/bin/date", "date", NULL);
```

The first argument to `execl` is the *file name* of the command; you have to know where it is found in the file system. The second argument is conventionally the program name (that is, the last component of the file name), but this is seldom used except as a place-holder. If the command takes arguments, they are strung out after this; the end of the list is marked by a `NULL` argument.

The `execl` call overlays the existing program with the new one, runs that, then exits. There is *no* return to the original program.

More realistically, a program might fall into two or more phases that communicate only through temporary files. Here it is natural to make the second pass simply an `execl` call from the first.

The one exception to the rule that the original program never gets control back occurs when there is an error, for example if the file can't be found or is not executable. If you don't know where `date` is located, say

```

execl("/bin/date", "date", NULL);
execl("/usr/bin/date", "date", NULL);
fprintf(stderr, "Someone stole 'date'\n");

```

A variant of `execl` called `execv` is useful when you don't know in advance how many arguments there are going to be. The call is

```
execv(filename, argp);
```

where `argp` is an array of pointers to the arguments; the last pointer in the array must be `NULL` so `execv` can tell where the list ends. As with `execl`, `filename` is the file in which the program is found, and `argp[0]` is the name of the program. (This arrangement is identical to the `argv` array for program arguments.)

Neither of these routines provides the niceties of normal command execution. There is no automatic search of multiple directories—you have to know precisely where the command is located. Nor do you get the expansion of metacharacters like `<`, `>`, `*`, `?`, and `[]` in the argument list. If you want these, use `execl` to invoke the shell `sh`, which then does all the work. Construct a string `commandline` that contains the complete command as it would have been typed at the terminal, then say

```
execl("/bin/sh", "sh", "-c", commandline, NULL);
```

The shell is assumed to be at a fixed place, `/bin/sh`. Its argument `-c` says to treat the next argument as a whole command line, so it does just what you want. The only problem is in constructing the right information in `commandline`.

5.3 Control of Processes—Fork and Wait

So far what we've talked about isn't really all that useful by itself. Now we will show how to regain control after running a program with `execl` or `execv`. Since these routines simply overlay the new program on the old one, to save the old one requires that it first be split into two copies; one of these can be overlaid, while the other waits for the new, overlaying program to finish. The splitting is done by a routine called `fork`:

```
proc_id = fork();
```

splits the program into two copies, both of which continue to run. The only difference between the two is the value of `proc_id`, the "process id." In one of these processes (the "child"), `proc_id` is zero. In the other (the "parent"), `proc_id` is non-zero; it is the process number of the child. Thus the basic way to call, and return from, another program is

```
if (fork() == 0)
    execl("/bin/sh", "sh", "-c", cmd, NULL);    /* in child */
```

And in fact, except for handling errors, this is sufficient. The `fork` makes two copies of the program. In the child, the value returned by `fork` is zero, so it calls `execl` which does the command and then dies. In the parent, `fork` returns non-zero so it skips the `execl`. (If there is any error, `fork` returns `-1`).

More often, the parent wants to wait for the child to terminate before continuing itself. This can be done with the function `wait`:

```
int status;
if (fork() == 0)
    execl( ... );
wait(&status);
```

This still doesn't handle any abnormal conditions, such as a failure of the `execl` or `fork`, or the possibility that there might be more than one child running simultaneously. (The `wait` returns the process id of the terminated child, if you want to check it against the value returned by `fork`.) Finally, this fragment doesn't deal with any funny behavior on the part of the child (which is reported in `status`). Still, these three lines are the heart of the standard library's system routine, which we'll show in a moment.

The `status` returned by `wait` encodes in its low-order eight bits the system's idea of the child's termination status; it is 0 for normal termination and non-zero to indicate various kinds of problems. The next higher eight bits are taken from the argument of the call to `exit` which caused a normal termination of the child process. It is good coding practice for all programs to return meaningful status.

When a program is called by the shell, the three file descriptors 0, 1, and 2 are set up pointing at the right files, and all other possible file descriptors are available for use. When this program calls another one, correct etiquette suggests making sure the same conditions hold. Neither `fork` nor the `exec` calls affects open files in any way. If the parent is buffering output that must come out before output from the child, the parent must flush its buffers before the `execl`. Conversely, if a caller buffers an input stream, the called program will lose any information that has been read by the caller.

5.4 Pipes

A *pipe* is an I/O channel intended for use between two cooperating processes: one process writes into the pipe, while the other reads. The system looks after buffering the data and synchronizing the two processes. Most pipes are created by the shell, as in

```
ls | pr
```

which connects the standard output of `ls` to the standard input of `pr`. Sometimes, however, it is most convenient for a process to set up its own plumbing; in this section, we will illustrate how the pipe connection is established and used.

The system call `pipe` creates a pipe. Since a pipe is used for both reading and writing, two file descriptors are returned; the actual usage is like this:

```
int  fd[2];
stat = pipe(fd);
if (stat == -1)
    /* there was an error ... */
```

`fd` is an array of two file descriptors, where `fd[0]` is the read side of the pipe and `fd[1]` is for writing. These may be used in `read`, `write` and `close` calls just like any other file descriptors.

If a process reads a pipe which is empty, it will wait until data arrives; if a process writes into a pipe which is too full, it will wait until the pipe empties somewhat. If the write side of the pipe is closed, a subsequent `read` will encounter end of file.

To illustrate the use of pipes in a realistic setting, let us write a function called `popen(cmd, mode)`, which creates a process `cmd` (just as `system` does), and returns a file descriptor that will either read or write that process, according to `mode`. That is, the call

```
fout = popen("pr", WRITE);
```

creates a process that executes the `pr` command; subsequent `write` calls using the file descriptor `fout` will send their data to that process through the pipe.

`popen` first creates the the pipe with a `pipe` system call; it then `forks` to create two copies of itself. The child decides whether it is supposed to read or write, closes the other side of the pipe, then calls the shell (via `exec1`) to run the desired process. The parent likewise closes the end of the pipe it does not use. These closes are necessary to make end-of-file tests work properly. For example, if a child that intends to read fails to close the write end of the pipe, it will never see the end of the pipe file, just because there is one writer potentially active.

```

#include <stdio.h>

#define READ 0
#define WRITE 1
#define tst(a, b) (mode == READ ? (b) : (a))
static int popen_pid;

popen(cmd, mode)
char *cmd;
int mode;
{
    int p[2];
    if (pipe(p) < 0)
        return(NULL);
    if ((popen_pid = fork()) == 0) {
        close(tst(p[WRITE], p[READ]));
        close(tst(0, 1));
        dup(tst(p[READ], p[WRITE]));
        close(tst(p[READ], p[WRITE]));
        execl("/bin/sh", "sh", "-c", cmd, 0);
        _exit(1); /* disaster has occurred if we get here */
    }
    if (popen_pid == -1)
        return(NULL);
    close(tst(p[READ], p[WRITE]));
    return(tst(p[WRITE], p[READ]));
}

```

The sequence of `closes` in the child is a bit tricky. Suppose that the task is to create a child process that will read data from the parent. Then the first `close` closes the write side of the pipe, leaving the read side open. The lines

```

close(tst(0, 1));
dup(tst(p[READ], p[WRITE]));

```

are the conventional way to associate the pipe descriptor with the standard input of the child. The `close` closes file descriptor 0, that is, the standard input. `dup` is a system call that returns a duplicate of an already open file descriptor. File descriptors are assigned in increasing order and the first available one is returned, so the effect of the `dup` is to copy the file descriptor for the pipe (read side) to file descriptor 0; thus the read side of the pipe becomes the standard input. (Yes, this is a bit tricky, but it's a standard idiom.) Finally, the old read side of the pipe is closed.

A similar sequence of operations takes place when the child process is supposed to write from the parent instead of reading. You may find it a useful exercise to step through that case.

The job is not quite done, for we still need a function `pclose` to close the pipe created by `popen`. The main reason for using a separate function rather than `close` is that it is desirable to wait for the termination of the child process. First, the return value from `pclose` indicates whether the process succeeded. Equally important when a process creates several children is that only a bounded number of unwaited-for children can exist, even if some of them have terminated; performing the `wait` lays the child to rest. Thus:


```

#include <signal.h>
pclose(fd)      /* close pipe fd */
int fd;
{
    register r, (*hstat>(), (*istat>(), (*qstat>();
    int status;
    extern int popen_pid;

    close(fd);
    istat = signal(SIGINT, SIG_IGN);
    qstat = signal(SIGQUIT, SIG_IGN);
    hstat = signal(SIGHUP, SIG_IGN);
    while ((r = wait(&status)) != popen_pid && r != -1);
    if (r == -1)
        status = -1;
    signal(SIGINT, istat);
    signal(SIGQUIT, qstat);
    signal(SIGHUP, hstat);
    return(status);
}

```

The calls to `signal` make sure that no interrupts, etc., interfere with the waiting process; this is the topic of the next section.

The routine as written has the limitation that only one pipe may be open at once, because of the single shared variable `popen_pid`; it really should be an array indexed by file descriptor. A `popen` function, with slightly different arguments and return value is available as part of the standard I/O library discussed below. As currently written, it shares the same limitation.

6. SIGNALS—INTERRUPTS AND ALL THAT

This section is concerned with how to deal gracefully with signals from the outside world (like interrupts), and with program faults. Since there's nothing very useful that can be done from within C about program faults, which arise mainly from illegal memory references or from execution of peculiar instructions, we'll discuss only the outside-world signals: *interrupt*, which is sent when the DEL character is typed; *quit*, generated by the FS character; *hangup*, caused by hanging up the phone; and *terminate*, generated by the *kill* command. When one of these events occurs, the signal is sent to *all* processes which were started from the corresponding terminal; unless other arrangements have been made, the signal terminates the process. In the *quit* case, a core image file is written for debugging purposes.

The routine which alters the default action is called `signal`. It has two arguments: the first specifies the signal, and the second specifies how to treat it. The first argument is just a number code, but the second is the address is either a function, or a somewhat strange code that requests that the signal either be ignored, or that it be given the default action. The include file `signal.h` gives names for the various arguments, and should always be included when signals are used. Thus

```

#include <signal.h>
...
signal(SIGINT, SIG_IGN);

```

causes interrupts to be ignored, while

```

signal(SIGINT, SIG_DFL);

```

restores the default action of process termination. In all cases, `signal` returns the previous value of the signal. The second argument to `signal` may instead be the name of a function (which has to be declared explicitly if the compiler hasn't seen it already). In this case, the named routine will be called when the signal occurs. Most commonly this facility is used to

allow the program to clean up unfinished business before terminating, for example to delete a temporary file:

```
#include <signal.h>

main()
{
    int onintr();

    if (signal(SIGINT, SIG_IGN) != SIG_IGN)
        signal(SIGINT, onintr);

    /* Process ... */
    exit(0);
}

onintr()
{
    unlink(tempfile);
    exit(1);
}
```

Why the test and the double call to `signal`? Recall that signals like `interrupt` are sent to *all* processes started from a particular terminal. Accordingly, when a program is to be run non-interactively (started by `&`), the shell turns off interrupts for it so it won't be stopped by interrupts intended for foreground processes. If this program began by announcing that all interrupts were to be sent to the `onintr` routine regardless, that would undo the shell's effort to protect it when run in the background.

The solution, shown above, is to test the state of interrupt handling, and to continue to ignore interrupts if they are already being ignored. The code as written depends on the fact that `signal` returns the previous state of a particular signal. If signals were already being ignored, the process should continue to ignore them; otherwise, they should be caught.

A more sophisticated program may wish to intercept an interrupt and interpret it as a request to stop what it is doing and return to its own command-processing loop. Think of a text editor: interrupting a long printout should not cause it to terminate and lose the work already done. The outline of the code for this case is probably best written like this:

```
#include <signal.h>
#include <setjmp.h>
jmp_buf  sjbuf;

main()
{
    int (*istat)(), onintr();

    istat = signal(SIGINT, SIG_IGN); /* save original status */
    setjmp(sjbuf); /* save current stack position */
    if (istat != SIG_IGN)
        signal(SIGINT, onintr);

    /* main processing loop */
}

onintr()
{
    printf("\nInterrupt\n");
    longjmp(sjbuf); /* return to saved state */
}
```

The include file `setjmp.h` declares the type `jmp_buf` an object in which the state can be saved. `sjbuf` is such an object; it is an array of some sort. The `setjmp` routine then saves the state of things. When an interrupt occurs, a call is forced to the `onintr` routine, which

can print a message, set flags, or whatever. `longjmp` takes as argument an object stored into by `setjmp`, and restores control to the location after the call to `setjmp`, so control (and the stack level) will pop back to the place in the main routine where the signal is set up and the main loop entered. Notice, by the way, that the signal gets set again after an interrupt occurs. This is necessary; most signals are automatically reset to their default action when they occur.

Some programs that want to detect signals simply can't be stopped at an arbitrary point, for example in the middle of updating a linked list. If the routine called on occurrence of a signal sets a flag and then returns instead of calling `exit` or `longjmp`, execution will continue at the exact point it was interrupted. The interrupt flag can then be tested later.

There is one difficulty associated with this approach. Suppose the program is reading the terminal when the interrupt is sent. The specified routine is duly called; it sets its flag and returns. If it were really true, as we said above, that "execution resumes at the exact point it was interrupted," the program would continue reading the terminal until the user typed another line. This behavior might well be confusing, since the user might not know that the program is reading; he presumably would prefer to have the signal take effect instantly. The method chosen to resolve this difficulty is to terminate the terminal read when execution resumes after the signal, returning an error code which indicates what happened.

Thus programs which catch and resume execution after signals should be prepared for "errors" which are caused by interrupted system calls. (The ones to watch out for are reads from a terminal, `wait`, and `pause`.) A program whose `onintr` program just sets `intflag`, resets the interrupt signal, and returns, should usually include code like the following when it reads the standard input:

```

    if (getchar() == EOF)
        if (intflag)
            /* EOF caused by interrupt */
        else
            /* true end-of-file */

```

A final subtlety to keep in mind becomes important when signal-catching is combined with execution of other programs. Suppose a program catches interrupts, and also includes a method (like "!" in the editor) whereby other programs can be executed. Then the code should look something like this:

```

    if (fork() == 0)
        execl( ... );
    signal(SIGINT, SIG_IGN); /* ignore interrupts */
    wait(&status); /* until the child is done */
    signal(SIGINT, onintr); /* restore interrupts */

```

Why is this? Again, it's not obvious but not really difficult. Suppose the program you call catches its own interrupts. If you interrupt the subprogram, it will get the signal and return to its main loop, and probably read your terminal. But the calling program will also pop out of its wait for the subprogram and read your terminal. Having two processes reading your terminal is very unfortunate, since the system figuratively flips a coin to decide who should get each line of input. A simple way out is to have the parent program ignore interrupts until the child is done. This reasoning is reflected in the standard I/O library function `system`:

```

#include <signal.h>
system(s) /* run command string s */
char *s;
{
    int status, pid, w;
    register int (*istat)(), (*qstat)();
    if ((pid = fork()) == 0) {
        execl("/bin/sh", "sh", "-c", s, 0);
        _exit(127);
    }
    istat = signal(SIGINT, SIG_IGN);
    qstat = signal(SIGQUIT, SIG_IGN);
    while ((w = wait(&status)) != pid && w != -1)
        ;
    if (w == -1)
        status = -1;
    signal(SIGINT, istat);
    signal(SIGQUIT, qstat);
    return(status);
}

```

As an aside on declarations, the function `signal` obviously has a rather strange second argument. It is in fact a pointer to a function delivering an integer, and this is also the type of the signal routine itself. The two values `SIG_IGN` and `SIG_DFL` have the right type, but are chosen so they coincide with no possible actual functions. For the enthusiast, here is how they are defined for the PDP-11; the definitions should be sufficiently ugly and nonportable to encourage use of the include file.

```

#define SIG_DFL (int (*)())0
#define SIG_IGN (int (*)())1

```

REFERENCES

- [1] Dolotta, T. A., Olsson, S. B., and Petruccelli, A. G. (eds.). *UNIX User's Manual—Release 3.0*, Bell Laboratories (June 1980).
- [2] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*, Prentice-Hall, Inc., 1978.
- [3] B. W. Kernighan. *UNIX for Beginners—Second Edition*, Bell Laboratories, 1978.

Appendix — The Standard I/O Library

D. M. Ritchie

Bell Laboratories
Murray Hill, New Jersey 07974

The standard I/O library was designed with the following goals in mind.

1. It must be as efficient as possible, both in time and in space, so that there will be no hesitation in using it no matter how critical the application.
2. It must be simple to use, and also free of the magic numbers and mysterious calls whose use mars the understandability and portability of many programs using older packages.
3. The interface provided should be applicable on all machines, whether or not the programs which implement it are directly portable to other systems, or to machines other than the PDP-11 running a version of UNIX.

1. GENERAL USAGE

Each program using the library must have the line

```
#include <stdio.h>
```

which defines certain macros and variables. The routines are in the normal C library, so no special library argument is needed for loading. All names in the include file intended only for internal use begin with an underscore `_` to reduce the possibility of collision with a user name. The names intended to be visible outside the package are

`stdin` The name of the standard input file
`stdout` The name of the standard output file
`stderr` The name of the standard error file
`EOF` is actually `-1`, and is the value returned by the read routines on end-of-file or error.
`NULL` is a notation for the null pointer, returned by pointer-valued functions to indicate an error
`FILE` expands to `struct _iob` and is a useful shorthand when declaring pointers to streams.
`BUFSIZ` is a number (viz. 512) of the size suitable for an I/O buffer supplied by the user. See `setbuf`, below.
`getc`, `getchar`, `putc`, `putchar`, `feof`, `ferror`, `fileno`
are defined as macros. Their actions are described below; they are mentioned here to point out that it is not possible to redeclare them and that they are not actually functions; thus, for example, they may not have breakpoints set on them.

The routines in this package offer the convenience of automatic buffer allocation and output flushing where appropriate. The names `stdin`, `stdout`, and `stderr` are in effect constants and may not be assigned to.

2. CALLS

`FILE *fopen(filename, type) char *filename, *type;`
opens the file and, if needed, allocates a buffer for it. `filename` is a character string specifying the name. `type` is a character string (not a single character). It may be `"r"`, `"w"`, or `"a"` to indicate intent to read, write, or append. The value returned is a file pointer. If it is `NULL` the attempt to open failed.

FILE *freopen(filename, type, ioptr) char *filename, *type; FILE *ioptr;
 The stream named by *ioptr* is closed, if necessary, and then reopened as if by *fopen*. If the attempt to open fails, *NULL* is returned, otherwise *ioptr*, which will now refer to the new file. Often the reopened stream is *stdin* or *stdout*.

int getc(ioptr) FILE *ioptr;
 returns the next character from the stream named by *ioptr*, which is a pointer to a file such as returned by *fopen*, or the name *stdin*. The integer *EOF* is returned on end-of-file or when an error occurs. The null character *\0* is a legal character.

int fgetc(ioptr) FILE *ioptr;
 acts like *getc* but is a genuine function, not a macro, so it can be pointed to, passed as an argument, etc.

putc(c, ioptr) FILE *ioptr;
putc writes the character *c* on the output stream named by *ioptr*, which is a value returned from *fopen* or perhaps *stdout* or *stderr*. The character is returned as value, but *EOF* is returned on error.

fputc(c, ioptr) FILE *ioptr;
 acts like *putc* but is a genuine function, not a macro.

fclose(ioptr) FILE *ioptr;
 The file corresponding to *ioptr* is closed after any buffers are emptied. A buffer allocated by the I/O system is freed. *fclose* is automatic on normal termination of the program.

fflush(ioptr) FILE *ioptr;
 Any buffered information on the (output) stream named by *ioptr* is written out. Output files are normally buffered if and only if they are not directed to the terminal; however, *stderr* always starts off unbuffered and remains so unless *setbuf* is used, or unless it is reopened.

exit(errcode);
 terminates the process and returns its argument as status to the parent. This is a special version of the routine which calls *fflush* for each output file. To terminate without flushing, use *_exit*.

feof(ioptr) FILE *ioptr;
 returns non-zero when end-of-file has occurred on the specified input stream.

ferror(ioptr) FILE *ioptr;
 returns non-zero when an error has occurred while reading or writing the named stream. The error indication lasts until the file has been closed.

getchar();
 is identical to *getc(stdin)*.

putchar(c);
 is identical to *putc(c, stdout)*.

char *fgets(s, n, ioptr) char *s; FILE *ioptr;
 reads up to *n-1* characters from the stream *ioptr* into the character pointer *s*. The read terminates with a new-line character. The new-line character is placed in the buffer followed by a null character. *fgets* returns the first argument, or *NULL* if error or end-of-file occurred.

fputs(s, ioptr) char *s; FILE *ioptr;
 writes the null-terminated string (character array) *s* on the stream *ioptr*. No new-line is appended. No value is returned.

`ungetc(c, ioptr) FILE *ioptr;`
 The argument character `c` is pushed back on the input stream named by `ioptr`. Only one character may be pushed back.

`printf(format, a1, ...) char *format;`
`fprintf(ioptr, format, a1, ...) FILE *ioptr; char *format;`
`sprintf(s, format, a1, ...) char *s, *format;`
`printf` writes on the standard output. `fprintf` writes on the named output stream. `sprintf` puts characters in the character array (string) named by `s`. The specifications are as described in entry `printf(3) UNIX User's Manual [1]`.

`scanf(format, a1, ...) char *format;`
`fscanf(ioptr, format, a1, ...) FILE *ioptr; char *format;`
`sscanf(s, format, a1, ...) char *s, *format;`
`scanf` reads from the standard input. `fscanf` reads from the named input stream. `sscanf` reads from the character string supplied as `s`. `scanf` reads characters, interprets them according to a format, and stores the results in its arguments. Each routine expects as arguments a control string `format`, and a set of arguments, *each of which must be a pointer*, indicating where the converted input should be stored.

`scanf` returns as its value the number of successfully matched and assigned input items. This can be used to decide how many input items were found. On end of file, EOF is returned; note that this is different from 0, which means that the next input character does not match what was called for in the control string.

`fread(ptr, sizeof(*ptr), nitems, ioptr) FILE *ioptr;`
 reads `nitems` of data beginning at `ptr` from file `ioptr`. No advance notification that binary I/O is being done is required; when, for portability reasons, it becomes required, it will be done by adding an additional character to the mode-string on the `fopen` call.

`fwrite(ptr, sizeof(*ptr), nitems, ioptr) FILE *ioptr;`
 Like `fread`, but in the other direction.

`rewind(ioptr) FILE *ioptr;`
 rewinds the stream named by `ioptr`. It is not very useful except on input, since a rewound output file is still open only for output.

`system(string) char *string;`
 The `string` is executed by the shell as if typed at the terminal.

`getw(ioptr) FILE *ioptr;`
 returns the next word from the input stream named by `ioptr`. EOF is returned on end-of-file or error, but since this a perfectly good integer `feof` and `ferror` should be used. A "word" is 16 bits on the PDP-11.

`putw(w, ioptr) FILE *ioptr;`
 writes the integer `w` on the named output stream.

`setbuf(ioptr, buf) FILE *ioptr; char *buf;`
`setbuf` may be used after a stream has been opened but before I/O has started. If `buf` is NULL, the stream will be unbuffered. Otherwise the buffer supplied will be used. It must be a character array of sufficient size:

```
char buf[BUFSIZ];
```

`fileno(ioptr) FILE *ioptr;`
 returns the integer file descriptor associated with the file.

`fseek(ioptr, offset, ptrname) FILE *ioptr; long offset;`

The location of the next byte in the stream named by `ioptr` is adjusted. `offset` is a long integer. If `ptrname` is 0, the offset is measured from the beginning of the file; if `ptrname` is 1, the offset is measured from the current read or write pointer; if `ptrname` is 2, the offset is measured from the end of the file. The routine accounts properly for any buffering. (When this routine is used on non-UNIX systems, the offset must be a value returned from `ftell` and the `ptrname` must be 0).

`long ftell(ioptr) FILE *ioptr;`

The byte offset, measured from the beginning of the file, associated with the named stream is returned. Any buffering is properly accounted for. (On non-UNIX systems the value of this call is useful only for handing to `fseek`, so as to position the file to the same place it was when `ftell` was called.)

`getpw(uid, buf) char *buf;`

The password file is searched for the given integer user ID. If an appropriate line is found, it is copied into the character array `buf`, and 0 is returned. If no line is found corresponding to the user ID then 1 is returned.

`char *malloc(num);`

allocates `num` bytes. The pointer returned is sufficiently well aligned to be usable for any purpose. `NULL` is returned if no space is available.

`char *calloc(num, size);`

allocates space for `num` items each of size `size`. The space is guaranteed to be set to 0 and the pointer is sufficiently well aligned to be usable for any purpose. `NULL` is returned if no space is available.

`cfree(ptr) char *ptr;`

Space is returned to the pool used by `calloc`. Disorder can be expected if the pointer was not obtained from `calloc`.

The following are macros whose definitions may be obtained by including `<ctype.h>`.

`isalpha(c)` returns non-zero if the argument is alphabetic.

`isupper(c)` returns non-zero if the argument is upper-case alphabetic.

`islower(c)` returns non-zero if the argument is lower-case alphabetic.

`isdigit(c)` returns non-zero if the argument is a digit.

`isspace(c)` returns non-zero if the argument is a spacing character: tab, new-line, carriage return, vertical tab, form feed, space.

`ispunct(c)` returns non-zero if the argument is any punctuation character, i.e., not a space, letter, digit or control character.

`isalnum(c)` returns non-zero if the argument is a letter or a digit.

`isprint(c)` returns non-zero if the argument is printable—a letter, digit, or punctuation character.

`isctr1(c)` returns non-zero if the argument is a control character.

`isascii(c)` returns non-zero if the argument is an ASCII character, i.e., less than octal 0200.

`toupper(c)` returns the upper-case character corresponding to the lower-case letter `c`.

`tolower(c)` returns the lower-case character corresponding to the upper-case letter `c`.

Make—A Program for Maintaining Computer Programs

S. I. Feldman

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

In a programming project, it is easy to lose track of which files need to be reprocessed or recompiled after a change is made in some part of the source. *Make* provides a simple mechanism for maintaining up-to-date versions of programs that result from many operations on a number of files. It is possible to tell *make* the sequence of commands that create certain files and the list of files that require other files to be current before the operations can be done. Whenever a change is made in any part of the program, the *make* command will create the proper files simply, correctly, and with minimum effort.

The basic operation of *make* is to find the name of a needed target in the description, ensure that all of the files on which it depends exist and are up to date, and then create the target if it has not been modified since its generators were. The description file really defines the graph of dependencies; *make* does a depth-first search of this graph to determine what work is really necessary.

Make also provides a simple macro substitution facility and the ability to encapsulate commands in a single file for convenient administration.

Make was originally designed to run on the UNIX† time-sharing system.

Introduction

It is common practice to divide large programs into smaller, more manageable pieces. The pieces may require quite different treatments: some may need to be run through a macro processor, some may need to be processed by a sophisticated program generator (e.g., Yacc [1] or Lex [2]). The outputs of these generators may then have to be compiled with special options and with certain definitions and declarations. The code resulting from these transformations may then need to be loaded together with certain libraries under the control of special options. Related maintenance activities involve running complicated test scripts and installing validated modules. Unfortunately, it is very easy for a programmer to forget which files depend on which others, which files have been modified recently, and the exact sequence of operations needed to make or exercise a new version of the program. After a long editing session, one may easily lose track of which files have been changed and which object modules are still valid, since a change to a declaration can obsolete a dozen other files. Forgetting to compile a routine that has been changed or that uses changed declarations will result in a program that will not work, and a bug that can be very hard to track down. On the other hand, recompiling everything in sight just to be safe is very wasteful.

The program described in this report mechanizes many of the activities of program development and maintenance. If the information on inter-file dependences and command sequences is stored in a file, the simple command

† UNIX is a trademark of Bell Laboratories.

make

is frequently sufficient to update the interesting files, regardless of the number that have been edited since the last “make”. In most cases, the description file is easy to write and changes infrequently. It is usually easier to type the *make* command than to issue even one of the needed operations, so the typical cycle of program development operations becomes

think — edit — *make* — test . . .

Make is most useful for medium-sized programming projects; it does not solve the problems of maintaining multiple source versions or of describing huge programs. *Make* was designed for use on UNIX, but a version runs on GCOS.

Basic Features

The basic operation of *make* is to update a target file by ensuring that all of the files on which it depends exist and are up to date, then creating the target if it has not been modified since its dependents were. *Make* does a depth-first search of the graph of dependences. The operation of the command depends on the ability to find the date and time that a file was last modified.

To illustrate, let us consider a simple example: A program named *prog* is made by compiling and loading three C-language files *x.c*, *y.c*, and *z.c* with the *IS* library. By convention, the output of the C compilations will be found in files named *x.o*, *y.o*, and *z.o*. Assume that the files *x.c* and *y.c* share some declarations in a file named *defs*, but that *z.c* does not. That is, *x.c* and *y.c* have the line

```
#include "defs"
```

The following text describes the relationships and operations:

```
prog: x.o y.o z.o
      cc x.o y.o z.o -lS -o prog
x.o y.o: defs
```

If this information were stored in a file named *makefile*, the command

```
make
```

would perform the operations needed to recreate *prog* after any changes had been made to any of the four source files *x.c*, *y.c*, *z.c*, or *defs*.

Make operates using three sources of information: a user-supplied description file (as above), file names and “last-modified” times from the file system, and built-in rules to bridge some of the gaps. In our example, the first line says that *prog* depends on three “.o” files. Once these object files are current, the second line describes how to load them to create *prog*. The third line says that *x.o* and *y.o* depend on the file *defs*. From the file system, *make* discovers that there are three “.c” files corresponding to the needed “.o” files, and uses built-in information on how to generate an object from a source file (*i.e.*, issue a “cc -c” command).

The following long-winded description file is equivalent to the one above, but takes no advantage of *make*’s innate knowledge:

```

prog : x.o y.o z.o
      cc x.o y.o z.o -lS -o prog
x.o : x.c defs
      cc -c x.c
y.o : y.c defs
      cc -c y.c
z.o : z.c
      cc -c z.c

```

If none of the source or object files had changed since the last time *prog* was made, all of the files would be current, and the command

```
make
```

would just announce this fact and stop. If, however, the *defs* file had been edited, *x.c* and *y.c* (but not *z.c*) would be recompiled, and then *prog* would be created from the new “.o” files. If only the file *y.c* had changed, only it would be recompiled, but it would still be necessary to reload *prog*.

If no target name is given on the *make* command line, the first target mentioned in the description is created; otherwise the specified targets are made. The command

```
make x.o
```

would recompile *x.o* if *x.c* or *defs* had changed.

If the file exists after the commands are executed, its time of last modification is used in further decisions; otherwise the current time is used. It is often quite useful to include rules with mnemonic names and commands that do not actually produce a file with that name. These entries can take advantage of *make*'s ability to generate files and substitute macros. Thus, an entry “save” might be included to copy a certain set of files, or an entry “cleanup” might be used to throw away unneeded intermediate files. In other cases one may maintain a zero-length file purely to keep track of the time at which certain actions were performed. This technique is useful for maintaining remote archives and listings.

Make has a simple macro mechanism for substituting in dependency lines and command strings. Macros are defined by command arguments or description file lines with embedded equal signs. A macro is invoked by preceding the name by a dollar sign; macro names longer than one character must be parenthesized. The name of the macro is either the single character after the dollar sign or a name inside parentheses. The following are valid macro invocations:

```

$(CFLAGS)
$2
$(xy)
$Z
$(Z)

```

The last two invocations are identical. \$\$ is a dollar sign. All of these macros are assigned values during input, as shown below. Four special macros change values during the execution of the command: \$*, \$@, \$?, and \$<. They will be discussed later. The following fragment shows the use:

```

OBJECTS = x.o y.o z.o
LIBES = -lS
prog: $(OBJECTS)
      cc $(OBJECTS) $(LIBES) -o prog
...

```

The command

make

loads the three object files with the *IS* library. The command

```
make "LIBES= -ll -lS"
```

loads them with both the Lex (“-ll”) and the Standard (“-lS”) libraries, since macro definitions on the command line override definitions in the description. (It is necessary to quote arguments with embedded blanks in UNIX commands.)

The following sections detail the form of description files and the command line, and discuss options and built-in rules in more detail.

Description Files and Substitutions

A description file contains three types of information: macro definitions, dependency information, and executable commands. There is also a comment convention: all characters after a sharp (#) are ignored, as is the sharp itself. Blank lines and lines beginning with a sharp (#) are totally ignored. If a non-comment line is too long, it can be continued using a backslash. If the last character of a line is a backslash, then the backslash, the new-line, and all following blanks and tabs are replaced by a single blank.

A macro definition is a line containing an equal sign not preceded by a colon or a tab. The name (string of letters and digits) to the left of the equal sign (trailing blanks and tabs are stripped) is assigned the string of characters following the equal sign (leading blanks and tabs are stripped.) The following are valid macro definitions:

```
2 = xyz
abc = -ll -ly -lS
LIBES =
```

The last definition assigns LIBES the null string. A macro that is never explicitly defined has the null string as value. Macro definitions may also appear on the *make* command line (see below).

Other lines give information about target files. The general form of an entry is:

```
target1 [target2 . . .] [:] [dependent1 . . .] [; commands] [# . . .]
[(tab) commands] [# . . .]
```

Items inside brackets may be omitted. Targets and dependents are strings of letters, digits, periods, and slashes. (Shell metacharacters “*” and “?” are expanded.) A command is any string of characters not including a sharp (#—except in quotes) or new-line. Commands may appear either after a semicolon on a dependency line or on lines beginning with a tab immediately following a dependency line.

A dependency line may have either a single or a double colon. A target name may appear on more than one dependency line, but all of those lines must be of the same (single or double colon) type.

1. For the usual single-colon case, at most one of these dependency lines may have a command sequence associated with it. If the target is out of date with any of the dependents on any of the lines, and a command sequence is specified (even a null one following a semicolon or tab), it is executed; otherwise a default creation rule may be invoked.
2. In the double-colon case, a command sequence may be associated with each dependency line; if the target is out of date with any of the files on a particular line, the associated commands are executed. A built-in rule may also be executed. This detailed form is of particular value in updating archive-type files.

If a target must be created, the sequence of commands is executed. Normally, each command line is printed and then passed to a separate invocation of the Shell after substituting for

macros. (The printing is suppressed in silent mode or if the command line begins with an @ sign). *Make* normally stops if any command signals an error by returning a non-zero error code. (Errors are ignored if the “-i” flag has been specified on the *make* command line, if the fake target name “.IGNORE” appears in the description file, or if the command string in the description file begins with a hyphen. Some UNIX commands return meaningless status). Because each command line is passed to a separate invocation of the Shell, care must be taken with certain commands (e.g., *cd* and Shell control commands) that have meaning only within a single Shell process; the results are forgotten before the next line is executed.

Before issuing any command, certain macros are set. $\$@$ is set to the name of the file to be “made”. $\$?$ is set to the string of names that were found to be younger than the target. If the command was generated by an implicit rule (see below), $\$<$ is the name of the related file that caused the action, and $\$*$ is the prefix shared by the current and the dependent file names.

If a file must be made but there are no explicit commands or relevant built-in rules, the commands associated with the name “.DEFAULT” are used. If there is no such name, *make* prints a message and stops.

Command Usage

The *make* command takes four kinds of arguments: macro definitions, flags, description file names, and target file names.

```
make [ flags ] [ macro definitions ] [ targets ]
```

The following summary of the operation of the command explains how these arguments are interpreted.

First, all macro definition arguments (arguments with embedded equal signs) are analyzed and the assignments made. Command-line macros override corresponding definitions found in the description files.

Next, the flag arguments are examined. The permissible flags are

- i Ignore error codes returned by invoked commands. This mode is entered if the fake target name “.IGNORE” appears in the description file.
- s Silent mode. Do not print command lines before executing. This mode is also entered if the fake target name “.SILENT” appears in the description file.
- r Do not use the built-in rules.
- n No execute mode. Print commands, but do not execute them. Even lines beginning with an “@” sign are printed.
- t Touch the target files (causing them to be up to date) rather than issue the usual commands.
- q Question. The *make* command returns a zero or non-zero status code depending on whether the target file is or is not up to date.
- p Print out the complete set of macro definitions and target descriptions
- d Debug mode. Print out detailed information on files and times examined.
- f Description file name. The next argument is assumed to be the name of a description file. A file name of “-” denotes the standard input. If there are no “-f” arguments, the file named *makefile* or *Makefile* in the current directory is read. The contents of the description files override the built-in rules if they are present).

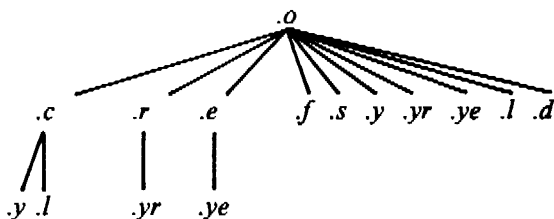
Finally, the remaining arguments are assumed to be the names of targets to be made; they are done in left to right order. If there are no such arguments, the first name in the description files that does not begin with a period is “made”.

Implicit Rules

The *make* program uses a table of interesting suffixes and a set of transformation rules to supply default dependency information and implied commands. (The Appendix describes these tables and means of overriding them.) The default suffix list is:

<i>.o</i>	Object file
<i>.c</i>	C source file
<i>.e</i>	Efl source file
<i>.r</i>	Ratfor source file
<i>.f</i>	Fortran source file
<i>.s</i>	Assembler source file
<i>.y</i>	Yacc-C source grammar
<i>.yr</i>	Yacc-Ratfor source grammar
<i>.ye</i>	Yacc-Efl source grammar
<i>.l</i>	Lex source grammar

The following diagram summarizes the default transformation paths. If there are two paths connecting a pair of suffixes, the longer one is used only if the intermediate file exists or is named in the description.



If the file *x.o* were needed and there were an *x.c* in the description or directory, it would be compiled. If there were also an *x.l*, that grammar would be run through Lex before compiling the result. However, if there were no *x.c* but there were an *x.l*, *make* would discard the intermediate C-language file and use the direct link in the graph above.

It is possible to change the names of some of the compilers used in the default, or the flag arguments with which they are invoked by knowing the macro names used. The compiler names are the macros AS, CC, RC, EC, YACC, YACCR, YACCE, and LEX. The command

```
make CC=newcc
```

will cause the "newcc" command to be used instead of the usual C compiler. The macros CFLAGS, RFLAGS, EFLAGS, YFLAGS, and LFLAGS may be set to cause these commands to be issued with optional flags. Thus,

```
make "CFLAGS=-O"
```

causes the optimizing C compiler to be used.

Example

As an example of the use of *make*, we will present the description file used to maintain the *make* command itself. The code for *make* is spread over a number of C source files and a Yacc grammar. The description file contains:

```

# Description file for the Make command
P = und -3 | opr -r2    # send to GCOS to be printed
FILES = Makefile version.c defs main.c doname.c misc.c files.c dosys.cgram.y lex.c gcos.c
OBJECTS = version.o main.o doname.o misc.o files.o dosys.o gram.o
LIBES = -lS
LINT = lint -p
CFLAGS = -O
make: $(OBJECTS)
      cc $(CFLAGS) $(OBJECTS) $(LIBES) -o make
      size make
$(OBJECTS): defs
gram.o: lex.c
cleanup:
      -rm *.o gram.c
      -du
install:
      @size make /usr/bin/make
      cp make /usr/bin/make ; rm make
print: $(FILES)    # print recently changed files
      pr $? | $P
      touch print
test:
      make -dp | grep -v TIME >1zap
      /usr/bin/make -dp | grep -v TIME >2zap
      diff 1zap 2zap
      rm 1zap 2zap
lint : dosys.c doname.c files.c main.c misc.c version.c gram.c
      $(LINT) dosys.c doname.c files.c main.c misc.c version.c gram.c
      rm gram.c
arch:
      ar uv /sys/source/s2/make.a $(FILES)

```

Make usually prints out each command before issuing it. The following output results from typing the simple command

```
make
```

in a directory containing only the source and description file:

```

cc -c version.c
cc -c main.c
cc -c doname.c
cc -c misc.c
cc -c files.c
cc -c dosys.c
yacc gram.y
mv y.tab.c gram.c
cc -c gram.c
cc version.o main.o doname.o misc.o files.o dosys.o gram.o -lS -o make
13188+3348+3044 = 19580b = 046174b

```

Although none of the source files or grammars were mentioned by name in the description file, *make* found them using its suffix rules and issued the needed commands. The string of digits results from the "size make" command; the printing of the command line itself was suppressed

by an @ sign. The @ sign on the *size* command in the description file suppressed the printing of the command, so only the sizes are written.

The last few entries in the description file are useful maintenance sequences. The "print" entry prints only the files that have been changed since the last "make print" command. A zero-length file *print* is maintained to keep track of the time of the printing; the \$? macro in the command line then picks up only the names of the files changed since *print* was touched. The printed output can be sent to a different printer or to a file by changing the definition of the *P* macro:

```
make print "P = opr -sp"           or
make print "P= cat >zap"
```

Suggestions and Warnings

The most common difficulties arise from *make*'s specific meaning of dependency. If file *x.c* has a "#include "defs"" line, then the object file *x.o* depends on *defs*; the source file *x.c* does not. (If *defs* is changed, it is not necessary to do anything to the file *x.c*, while it is necessary to recreate *x.o*.)

To discover what *make* would do, the "-n" option is very useful. The command

```
make -n
```

orders *make* to print out the commands it would issue without actually taking the time to execute them. If a change to a file is absolutely certain to be benign (e.g., adding a new definition to an include file), the "-t" (touch) option can save a lot of time: instead of issuing a large number of superfluous recompilations, *make* updates the modification times on the affected file. Thus, the command

```
make -ts
```

("touch silently") causes the relevant files to appear up to date. Obvious care is necessary, since this mode of operation subverts the intention of *make* and destroys all memory of the previous relationships.

The debugging flag ("-d") causes *make* to print out a very detailed description of what it is doing, including the file times. The output is verbose, and recommended only as a last resort.

Acknowledgements

I would like to thank S. C. Johnson for suggesting this approach to program maintenance control. I would like to thank S. C. Johnson and H. Gajewska for being the prime guinea pigs during development of *make*.

References

1. S. C. Johnson. *YACC—Yet Another Compiler-Compiler*, Bell Laboratories.
2. M. E. Lesk. *LEX—A Lexical Analyzer Generator*, Bell Laboratories.

Appendix. Suffixes and Transformation Rules

The *make* program itself does not know what file name suffixes are interesting or how to transform a file with one suffix into a file with another suffix. This information is stored in an internal table that has the form of a description file. If the “-r” flag is used, this table is not used.

The list of suffixes is actually the dependency list for the name “.SUFFIXES”; *make* looks for a file with any of the suffixes on the list. If such a file exists, and if there is a transformation rule for that combination, *make* acts as described earlier. The transformation rule names are the concatenation of the two suffixes. The name of the rule to transform a “.r” file to a “.o” file is thus “.r.o”. If the rule is present and no explicit command sequence has been given in the user’s description files, the command sequence for the rule “.r.o” is used. If a command is generated by using one of these suffixing rules, the macro \$* is given the value of the stem (everything but the suffix) of the name of the file to be made, and the macro \$< is the name of the dependent that caused the action.

The order of the suffix list is significant, since it is scanned from left to right, and the first name that is formed that has both a file and a rule associated with it is used. If new names are to be appended, the user can just add an entry for “.SUFFIXES” in his own description file; the dependents will be added to the usual list. A “.SUFFIXES” line without any dependents deletes the current list. (It is necessary to clear the current list if the order of names is to be changed). The following is an excerpt from the default rules file:

```
.SUFFIXES : .o .c .e .r .f .y .yr .ye .l .s
YACC=yacc
YACCR=yacc -r
YACCE=yacc -e
YFLAGS=
LEX=lex
LFLAGS=
CC=cc
AS=as -
CFLAGS=
RC=ec
RFLAGS=
EC=ec
EFLAGS=
FFLAGS=
.c.o :
    $(CC) $(CFLAGS) -c $<
.e.o .r.o .f.o :
    $(EC) $(RFLAGS) $(EFLAGS) $(FFLAGS) -c $<
.s.o :
    $(AS) -o $@ $<
.y.o :
    $(YACC) $(YFLAGS) $<
    $(CC) $(CFLAGS) -c y.tab.c
    rm y.tab.c
    mv y.tab.o $@
.y.c :
    $(YACC) $(YFLAGS) $<
    mv y.tab.c $@
```

An Augmented Version of MAKE

E. G. Bradford

Bell Laboratories
Whippany, New Jersey 07981

ABSTRACT

This paper describes an augmented version of the UNIX† *make* command. With one debatable exception, this version is completely upward compatible with the old version. This paper describes and gives examples only of additional features. The reader is assumed to have read the original *make* paper by S. I. Feldman.¹ Further possible developments for *make* are also discussed.

1. INTRODUCTION

This paper describes in some detail an augmented version of the *make* program. Some justification will be given for the chosen implementation and examples will demonstrate the additional features.

2. MOTIVATION FOR THE CURRENT IMPLEMENTATION

The *make* program was originally written for personal use by S. I. Feldman. However, it became popular on the BTL Research UNIX machine and a more formal version was built and installed for general use. Further developments of *make* have not been necessary in the BTL Research environment, and thus have not been done.

Elsewhere, *make* was perceived as an excellent program administrative tool and has been used extensively in at least one project for over two years. However, *make* had many shortcomings: handling of libraries was tedious; handling of the SCCS² file-name format was difficult or impossible; environment variables were completely ignored by *make*; and the general lack of ability to maintain files in a remote directory. These shortcomings hindered large scale use of *make* as a program support tool.

Make has been modified to handle the problems above. The additional features are within the original syntactic framework of *make* and few if any new syntactical entities have been introduced. A notable exception is the *include* file capability. Further, most of the additions result in a "Don't know how to make ..." message from the old version of *make*.

3. THE ADDITIONAL FEATURES

The following paragraphs describe with examples the additional features of the *make* program. In general, the examples are taken from existing *makefiles*. Also, the appendices are working *makefiles*.

3.1 The Environment Variables

Environment variables are read and added to the macro definitions each time *make* executes. Precedence is a prime consideration in doing this properly. For example, if the environment variable CC is set to *occ*, does it override the command line? Does it override the definition in the makefile? The following describes *make*'s interaction with the environment.

† UNIX is a trademark of Bell Laboratories.

1. *MAKE—A Program for Maintaining Computer Programs* by S. I. Feldman.
2. *Source Code Control System User's Guide* by L. E. Bonanni and C. A. Salemi.

A new macro, MAKEFLAGS is maintained by *make*. It is defined as the collection of all input flag arguments into a string (without minus signs). It is exported, and thus accessible to further invocations of *make*. Command line flags and assignments in the *makefile* update MAKEFLAGS. Thus, to describe how the environment interacts with *make*, we also need to consider the MAKEFLAGS macro (environment variable).

When executed, *make* assigns macro definitions in the following order:

1. Read the MAKEFLAGS environment variable. If it is not present or null, the internal *make* variable MAKEFLAGS is set to the null string. Otherwise, each letter in MAKEFLAGS is assumed to be an input flag argument and is processed as such. (The only exceptions are the *-f*, *-p*, and *-r* flags.)
2. Read and set the input flags from the command line. The command line adds to the previous settings from the MAKEFLAGS environment variable.
3. Read macro definitions from the command line. These are made *not resettable*. Thus any further assignments to these names are ignored.
4. Read the internal list of macro definitions. These are found in the file *rules.c* of the source for *make*. (See Appendix A for the complete makefile which represents the internally defined macros and rules.) They give default definitions for the C compiler (CC=cc), the assembler (AS=as), etc.
5. Read the environment. The environment variables are treated as macro definitions and marked as *exported* (in the shell sense). Note, MAKEFLAGS will be read and set again. However, because it is not an internally defined variable (in *rules.c*), this has the effect of doing the same assignment twice. The exception to this is when MAKEFLAGS is assigned on the command line. (The reason it was read previously, was to be able to turn the debug flag on before anything else was done.)
6. Read the *makefile(s)*. The assignments in the *makefile(s)* will override the environment. This order was chosen so when one reads a makefile and executes *make* one knows what to expect. That is, one gets what one sees unless the *-e* flag is used. The *-e* is an additional command-line flag that tells *make* to have the environment override the *makefile* assignments. Thus if *make -e ...* is typed, the variables in the environment override the definitions in the *makefile*. (Note, there is no way to override the command line assignments.) Also note that if MAKEFLAGS is assigned it will override the environment. (This would be useful for further invocations of *make* from the current *makefile*.)

It may be clearer to list the precedence of assignments. Thus, in order from least binding to most binding, we have:

1. Internal definitions (from *rules.c*).
2. Environment.
3. *Makefile(s)*.
4. Command line.

The *-e* flag has the effect of changing the order to:

1. Internal definitions (from *rules.c*).
2. *Makefile(s)*.
3. Environment.
4. Command line.

This ordering is general enough to allow a programmer to define a *makefile* or set of *makefiles* whose parameters are dynamically definable.

3.2 Recursive Makefiles

Another feature was added to *make* concerning the environment and recursive invocations. If the sequence \$(MAKE) appears anywhere in a shell command line, the line will be executed even if the *-n* flag is set. Because the *-n* flag is exported across invocations of *make*,

(through the MAKEFLAGS variable) the only thing which will actually get executed is the *make* command itself. This feature is useful when a hierarchy of *makefile(s)* describes a set of software subsystems. For testing purposes, **make -n ...** can be executed and everything that would have been done will get printed out; including output from lower level invocations of *make*.

3.3 Format of Shell Commands within Make

Make remembers embedded new-lines and tabs in shell command sequences. Thus, if the programmer puts a *for* loop in the makefile with indentation, when *make* prints it out, it retains the indentation and backslashes. The output can still be piped to the shell and is readable. This is obviously a cosmetic change; no new functionality is gained.

3.4 Archive Libraries

Make has an improved interface to archive libraries. Due to a lack of documentation, most people are probably not aware of the current syntax for addressing members of archive libraries. The previous version of *make* allows a user to name a member of a library in the following manner:

```
lib(object.o)
```

or:

```
lib((_localtime))
```

where the second method actually refers to an entry point of an object file within the library. (*Make* looks through the library, locates the entry point and translates it to the correct object file name.)

To use this procedure to maintain an archive library, the following type of *makefile* is required:

```
lib:: lib(ctime.o)
      $(CC) -c -O ctime.c
      ar rv lib ctime.o
      rm ctime.o
lib:: lib(fopen.o)
      $(CC) -c -O fopen.c
      ar rv lib fopen.o
      rm fopen.o
... and so on for each object ...
```

This is tedious and error prone. Obviously, the command sequences for adding a C file to a library are the same for each invocation, the file name being the only difference each time. (This is true in most cases.) Similarly for assembler and YACC and LEX files.

The current version gives the user access to a rule for building libraries. The *handle* for the rule is the **.a** suffix. Thus a **.c.a** rule is the rule for compiling a C source file, adding it to the library, and removing the **.o** cadaver. Similarly, the **.y.a**, the **.s.a** and the **.l.a** rules rebuild YACC, assembler, and LEX files, respectively. The current archive rules defined internally are **.c.a**, **.c~.a**, and **.s~.a**. (The tilde (~) syntax will be described shortly.) The user may define in his makefile any other rules he may need.

The above two-member library is then maintained with the following shorter makefile:

```
lib: lib(ctime.o) lib(fopen.o)
     @echo lib up-to-date.
```

The internal rules are already defined to complete the preceding library maintenance. The actual **.c.a** rules is as follows:

```
.c.a:
    $(CC) -c $(CFLAGS) $<
    ar rv $@ $*.o
    rm -f $*.o
```

Thus, the `$@` macro is the `.a` target (`lib`) and the `$<` and `$*` macros are set to the out-of-date C file and the file name without suffix, respectively (`ctime.c` and `ctime`). The `$<` macro (in the preceding rule) could have been changed to `$*.c`.

It might be useful to go into some detail about exactly what *make* does when it sees the construction:

```
lib:    lib(ctime.o)
        @echo lib up-to-date
```

Assume the object in the library is out-of-date with respect to `ctime.c`. Also, there is no `ctime.o` file:

1. Do `lib`.
2. To do `lib`, do each dependent of `lib`.
3. Do `lib(ctime.o)`.
4. To do `lib(ctime.o)`, do each dependent of `lib(ctime.o)`. (There are none.)
5. Use internal rules to try to build `lib(ctime.o)`. (There is no explicit rule.) Note that `lib(ctime.o)` has a parenthesis in the name so identify the target suffix as `.a`. (This is the key. There is no explicit `.a` at the end of the `lib` library name. The parenthesis forces the `.a` suffix.) In this sense, the `.a` is hard-wired into *make*.
6. Break the name `lib(ctime.o)` up into `lib` and `ctime.o`. Define two macros, `$@` (`=lib`) and `$*` (`=ctime`).
7. Look for a rule `.X.a` and a file `$*.X`. The first `.X` (in the `.SUFFIXES` list) which fulfills these conditions is `.c` so the rule is `.c.a` and the file is `ctime.c`. Set `$<` to be `ctime.c` and execute the rule. (In fact, *make* must then do `ctime.c`. However, the search of the current directory yields no other candidates, whence, the search ends.)
8. The library has been updated. Do the rule associated with the `lib`: dependency; namely:

```
echo lib up-to-date
```

It should be noted that to let `ctime.o` have dependencies, the following syntax is required:

```
lib(ctime.o): $(INCDIR)/stdio.h
```

Thus, explicit references to `.o` files are unnecessary. There is also a new macro for referencing the archive member name when this form is used. `$%` is evaluated each time `$@` is evaluated. If there is no current archive member, `$%` is null. If an archive member exists, then `$%` evaluates to the expression between the parentheses.

An example *makefile* for a larger library is given in Appendix B. The reader will note also that there are no lingering `*.o` files left around. The result is a library maintained directly from the source files (or more generally, from the SCCS files).

3.5 SCCS File Names: The Tilde

The syntax of *make* does not directly permit referencing of prefixes. For most types of files on UNIX machines this is acceptable, because nearly everyone uses a suffix to distinguish different types of files. SCCS files are the exception. Here, `s.` precedes the file-name part of the complete path name.

To allow *make* easy access to the prefix `s.` requires either a redefinition of the rule naming syntax of *make* or a trick. The trick is to use the tilde (`~`) as an identifier of SCCS files. Hence, `.c~.o` refers to the rule which transforms an SCCS C source file into an object. Specifically, the internal rule is:

```
.c~.o:
    $(GET) $(GFLAGS) -p $< > $*.c
    $(CC) $(CFLAGS) -c $*.c
    -rm -f $*.c
```

Thus the tilde appended to any suffix transforms the file search into an SCCS file-name search with the actual suffix named by the dot and all characters up to (but not including) the tilde. The following SCCS suffixes are internally defined:

```
.c~
.y~
.s~
.sh~
.h~
```

The following rules involving SCCS transformations are internally defined:

```
.c~:
.sh~:
.c~.o:
.s~.o:
.y~.o:
.l~.o:
.y~.c:
.c~.a:
.s~.a:
.h~.h:
```

Obviously, the user can define other rules and suffixes which may prove useful. The tilde gives him a handle on the SCCS file-name format so that this is possible.

3.6 The Null Suffix

In the UNIX source code, there are many commands that consist of a single source file. It seemed wasteful to maintain an object of such files for *make*'s pleasure. The current implementation supports single-suffix rules, or, if one prefers, a null suffix. Thus, to maintain the program *cat* one needs a rule in the *makefile* of the following form:

```
.c:
    $(CC) -n -O $< -o $@
```

In fact, this *.c:* rule is internally defined so no *makefile* is necessary at all! One need only type:

```
make cat dd echo date
```

(these are notable single-file programs) and all four C source files are passed through the above shell command line associated with the *.c:* rule. The internally defined single-suffix rules are:

```
.c:
.c~:
.sh:
.sh~:
```

Others may be added in the *makefile* by the user.

3.7 Include Files

Make has an include file capability. If the string **include** appears as the first seven letters of a line in a *makefile* and is followed by a blank or a tab, the following string is assumed to be a file name that the current invocation of *make* will read. The file descriptors are stacked for reading *include* files so no more than about sixteen levels of nested includes is supported.

3.8 Invisible SCCS Makefiles

SCCS *makefiles* are invisible to *make*. That is, if *make* is typed and only a file named

s.makefile exists, *make* will do a *get(1)* on it, then read it and remove it; likewise for *-f* arguments and *include* files.

3.9 Dynamic Dependency Parameters

A new dependency parameter has been defined. It has meaning only on the dependency line in a makefile. **\$\$@** refers to the current "thing" to the left of the colon (which is **\$@**). Also the form **\$\$(@F)** exists which allows access to the file part of **\$@**. Thus, in the following:

```
cat:  $$@.c
```

the dependency is translated at execution time to the string *cat.c*. This is useful for building large numbers of executable files, each of which has only one source file. For instance the UNIX command directory could have a *makefile* like:

```
CMDS = cat dd echo date cc cmp comm ar ld chown
$(CMDS):  $$@.c
$(CC) -O $? -o $@
```

Obviously, this is a subset of all the single-file programs. For multiple-file programs, a directory is usually allocated and a separate *makefile* is made. For any particular file which has a peculiar compilation procedure, a specific entry must be made in the *makefile*.

The second useful form of the dependency parameter is **\$\$(@F)**. It represents the file-name part of **\$\$@**. Again, it is evaluated at execution time. Its usefulness becomes evident when trying to maintain the */usr/include* directory from a makefile in the */usr/src/head* directory. Thus the */usr/src/head/makefile* would look like:

```
INCDIR = /usr/include
INCLUDES = \
    $(INCDIR)/stdio.h \
    $(INCDIR)/pwd.h \
    $(INCDIR)/dir.h \
    $(INCDIR)/a.out.h
$(INCLUDES): $$(@F)
cp $? $@
chmod 0444 $@
```

This would completely maintain the */usr/include* directory whenever one of the above files in */usr/src/head* was updated.

3.10 Extensions of \$*, \$@, and \$<

The internally generated macros **\$***, **\$@**, and **\$<** are useful generic terms for current targets and out-of-date relatives. To this list has been added the following related macros: **\$(@D)**, **\$(@F)**, **\$(*D)**, **\$(*F)**, **\$(<D)**, and **\$(<F)**. The **D** refers to the directory part of the single-letter macro. The **F** refers to the file-name part of the single-letter macro. These additions are useful when building hierarchical makefiles. They allow access to directory names for purposes of using the *cd* command of the shell. Thus, a shell command can be:

```
cd $(<D); $(MAKE) $(<F)
```

An interesting example of the use of these features can be found in the set of *makefiles* in Appendix C.

3.11 Output Translations

Macros in shell commands can now be translated when evaluated. The form is as follows:

```
$(macro:string1=string2)
```

The meaning is that **\$(macro)** is evaluated. For each appearance of *string1* in the evaluated macro, *string2* is substituted. The meaning of finding *string1* in **\$(macro)** is that the evaluated **\$(macro)** is considered as a bunch of strings each delimited by white space (blanks or tabs). Thus the occurrence of *string1* in **\$(macro)** means that a regular expression of the following form has been found:

```
.*<string1>[TAB|BLANK]
```

This particular form was chosen because *make* usually concerns itself with suffixes. A more general regular expression match could be implemented if the need arises. The usefulness of this type of translation occurs when maintaining archive libraries. Now, all that is necessary is to accumulate the out-of-date members and write a shell script which can handle all the C programs (i.e., those file ending in *.c*). Thus the following fragment will optimize the executions of *make* for maintaining an archive library:

```
$(LIB): $(LIB)(a.o) $(LIB)(b.o) $(LIB)c.o
$(CC) -c $(CFLAGS) $(?:.o=.c)
ar rv $(LIB) $?
rm $?
```

A dependency of the preceding form would be necessary for each of the different types of source files (suffixes) which define the archive library. These translations are added in an effort to make more general use of the wealth of information that *make* generates.

4. CONCLUSIONS

The augmentations described above have increased the size of *make* significantly, but it is our belief that this increase in size is a reasonable price to pay for the resulting additional features.

Appendix A: Internal Definitions

The following *makefile* will exactly reproduce the internal rules of the current version of *make*. Thus if `make -r ...` is typed and a *makefile* includes this *makefile* the results would be identical to excluding the `-r` option and the *include* line in the *makefile*. Note that this output can be reproduced by:

```
make -fp - < /dev/null 2>/dev/null
```

The output will appear on the standard output.)

```
# LIST OF SUFFIXES
.SUFFIXES: .o .c .c~ .y .y~ .l .l~ .s .s~ .sh .sh~ .h .h~

# PRESET VARIABLES
MAKE=make
YACC=yacc
YFLAGS=
LEX=lex
LFLAGS=
LD=ld
LDFLAGS=
CC=cc
CFLAGS=-O
AS=as
ASFLAGS=
GET=get
GFLAGS=

# SINGLE SUFFIX RULES
.c:
    $(CC) -n -O $< -o $@

.c~:
    $(GET) $(GFLAGS) -p $< > $*.c
    $(CC) -n -O $*.c -o $*
    -rm -f $*.c

.sh:
    cp $< $@

.sh~:
    $(GET) $(GFLAGS) -p $< > $*.sh
    cp $*.sh $*
    -rm -f $*.sh

# DOUBLE SUFFIX RULES
.c.o:
    $(CC) $(CFLAGS) -c $<

.c~.o:
    $(GET) $(GFLAGS) -p $< > $*.c
    $(CC) $(CFLAGS) -c $*.c
    -rm -f $*.c

.c~.c:
    $(GET) $(GFLAGS) -p $< > $*.c

.s.o:
    $(AS) $(ASFLAGS) -o $@ $<
```

```

.s~.o:
$(GET) $(GFLAGS) -p $< > $*.s
$(AS) $(ASFLAGS) -o $*.o $*.s
-rm -f $*.s

.y.o:
$(YACC) $(YFLAGS) $<
$(CC) $(CFLAGS) -c y.tab.c
rm y.tab.c
mv y.tab.o $@

.y~.o:
$(GET) $(GFLAGS) -p $< > $*.y
$(YACC) $(YFLAGS) $*.y
$(CC) $(CFLAGS) -c y.tab.c
rm -f y.tab.c $*.y
mv y.tab.o $*.o

.l.o:
$(LEX) $(LFLAGS) $<
$(CC) $(CFLAGS) -c lex.yy.c
rm lex.yy.c
mv lex.yy.o $@

.l~.o:
$(GET) $(GFLAGS) -p $< > $*.l
$(LEX) $(LFLAGS) $*.l
$(CC) $(CFLAGS) -c lex.yy.c
rm -f lex.yy.c $*.l
mv lex.yy.o $*.o

.y.c :
$(YACC) $(YFLAGS) $<
mv y.tab.c $@

.y~.c:
$(GET) $(GFLAGS) -p $< > $*.y
$(YACC) $(YFLAGS) $*.y
mv y.tab.c $*.c
-rm -f $*.y

.l.c :
$(LEX) $<
mv lex.yy.c $@

.c.a:
$(CC) -c $(CFLAGS) $<
ar rv $@ $*.o
rm -f $*.o

.c~.a:
$(GET) $(GFLAGS) -p $< > $*.c
$(CC) -c $(CFLAGS) $*.c
ar rv $@ $*.o
rm -f $*.[co]

.s~.a:
$(GET) $(GFLAGS) -p $< > $*.s
$(AS) $(ASFLAGS) -o $*.o $*.s
ar rv $@ $*.o
-rm -f $*.[so]

.h~.h:
$(GET) $(GFLAGS) -p $< > $*.h

```

Appendix B: A Library Makefile

The following library maintaining makefile is from current work on LSX. It completely maintains the LSX operating system library.

```

#      @(#)/usr/src/cmd/make/make.tm      3.2
LIB = lsplib

PR = vpr -b LSX

INSDIR = /r1/flop0/
INS = eval

lsx::  $(LIB) low.o mch.o
        ld -x low.o mch.o $(LIB)
        mv a.out lsx
        @size lsx

#      Here, $(INS) is either : or eval.
lsx::  $(INS) `cp lsx $(INSDIR)lsx && \
        strip $(INSDIR)lsx && \
        ls -l $(INSDIR)lsx`

print:
        $(PR) header.s low.s mch.s *.h *.c Makefile

$(LIB): \
        $(LIB)(clock.o) \
        $(LIB)(main.o) \
        $(LIB)(tty.o) \
        $(LIB)(trap.o) \
        $(LIB)(sysent.o) \
        $(LIB)(sys2.o) \
        $(LIB)(sys3.o) \
        $(LIB)(sys4.o) \
        $(LIB)(sys1.o) \
        $(LIB)(sig.o) \
        $(LIB)(fio.o) \
        $(LIB)(kl.o) \
        $(LIB)(alloc.o) \
        $(LIB)(nami.o) \
        $(LIB)(iget.o) \
        $(LIB)(rdwri.o) \
        $(LIB)(subr.o) \
        $(LIB)(bio.o) \
        $(LIB)(decfd.o) \
        $(LIB)(slp.o) \
        $(LIB)(space.o) \
        $(LIB)(puts.o)
        @echo $(LIB) now up-to-date.

.s.o:
        as -o $*.o header.s $*.s

.o.a:
        ar rv $@ $<
        rm -f $<

```

```
.s.a:
    as -o $*.o header.s $*.s
    ar rv $@ $*.o
    rm -f $*.o

.PRECIOUS: $(LIB)
```

Appendix C: Example of Recursive Use of Makefiles

The following set of *makefiles* maintain the operating system for Columbus UNIX. They are included here to provide realistic examples of the use of *make*. Each one is named **70.mk**. The following command forces a complete rebuild of the operating system:

```
FRC=FRC make -f 70.mk
```

where the current directory is **ucb**. Here, we have used some of the conventions described in A. Chellis's paper entitled *Proposed Structure for UNIX/TS and UNIX/RT Makefiles*. FRC is a convention for *FoR*Cing *make* to completely rebuild a target starting from scratch.

```
./ucb makefile

#      @(#)/usr/src/cmd/make/make.tm    3.2
#      ucb/70.mk makefile

VERSION = 70

DEPS = \
    os/low.$(VERSION).o \
    os/mch.$(VERSION).o \
    os/conf.$(VERSION).o \
    os/lib1.$(VERSION).a \
    io/lib2.$(VERSION).a

#      This makefile will re-load unix.$(VERSION) if any
#      of the $(DEPS) is out-of-date wrt unix.$(VERSION).
#      Note, it will not go out and check each member
#      of the libraries. To do this, the FRC macro must
#      be defined.

unix.$(VERSION):    $(DEPS) $(FRC)
                   load -s $(VERSION)

$(DEPS):            $(FRC)
                   cd $(@D); $(MAKE) -f $(VERSION).mk $(@F)

all:                unix.$(VERSION)
                   @echo unix.$(VERSION) up-to-date.

includes:
                   cd head/sys; $(MAKE) -f $(VERSION).mk

FRC:                includes;
```

```

#      @(#)/usr/src/cmd/make/make.tm      3.2
#      ucb/os/70.mk makefile

VERSION = 70

LIB = lib1.$(VERSION).a
COMPOOL=

LIBOBS = \
    $(LIB)(main.o) \
    $(LIB)(alloc.o) \
    $(LIB)(iget.o) \
    $(LIB)(prf.o) \
    $(LIB)(rdwri.o) \
    $(LIB)(slp.o) \
    $(LIB)(subr.o) \
    $(LIB)(text.o) \
    $(LIB)(trap.o) \
    $(LIB)(sig.o) \
    $(LIB)(sysent.o) \
    $(LIB)(sys1.o) \
    $(LIB)(sys2.o) \
    $(LIB)(sys3.o) \
    $(LIB)(sys4.o) \
    $(LIB)(sys5.o) \
    $(LIB)(syscb.o) \
    $(LIB)(maus.o) \
    $(LIB)(messag.o) \
    $(LIB)(nami.o) \
    $(LIB)(fio.o) \
    $(LIB)(clock.o) \
    $(LIB)(acct.o) \
    $(LIB)(errlog.o)

ALL = \
    conf.$(VERSION).o \
    low.$(VERSION).o \
    mch.$(VERSION).o \
    $(LIB)

all:    $(ALL)
        @echo `$(ALL)` now up-to-date.

$(LIB)::$(LIBOBS)

$(LIBOBS):    $(FRC);

FRC:
    rm -f $(LIB)

clobber:cleanup
    -rm -f $(LIB)

clean cleanup:;

install: all;

.PRECIOUS:    $(LIB)

```

```
#    @(#)/usr/src/cmd/make/make.tm    3.2
#    ucb/io/70.mk makefile
```

```
VERSION = 70
```

```
LIB = lib2.$(VERSION).a
```

```
COMPOOL=
```

```
LIB2OBJS = \
$(LIB)(mx1.o) \
$(LIB)(mx2.o) \
$(LIB)(bio.o) \
$(LIB)(tty.o) \
$(LIB)(malloc.o) \
$(LIB)(pipe.o) \
$(LIB)(dhdm.o) \
$(LIB)(dh.o) \
$(LIB)(dhfdm.o) \
$(LIB)(dj.o) \
$(LIB)(dn.o) \
$(LIB)(ds40.o) \
$(LIB)(dz.o) \
$(LIB)(alarm.o) \
$(LIB)(hf.o) \
$(LIB)(hps.o) \
$(LIB)(hpmap.o) \
$(LIB)(hp45.o) \
$(LIB)(hs.o) \
$(LIB)(ht.o) \
$(LIB)(jy.o) \
$(LIB)(kl.o) \
$(LIB)(lfh.o) \
$(LIB)(lp.o) \
$(LIB)(mem.o) \
$(LIB)(nmpipe.o) \
$(LIB)(rf.o) \
$(LIB)(rk.o) \
$(LIB)(rp.o) \
$(LIB)(rx.o) \
$(LIB)(sys.o) \
$(LIB)(trans.o) \
$(LIB)(ttdma.o) \
$(LIB)(tec.o) \
$(LIB)(tex.o) \
$(LIB)(tm.o) \
$(LIB)(vp.o) \
$(LIB)(vs.o) \
$(LIB)(vtlp.o) \
$(LIB)(vt11.o) \
$(LIB)(fakevtlp.o) \
$(LIB)(vt61.o) \
$(LIB)(vt100.o) \
$(LIB)(vtmon.o) \
$(LIB)(vtdbg.o) \
$(LIB)(vtutil.o) \
```

```

$(LIB)(vtast.o) \
$(LIB)(partab.o) \
$(LIB)(rh.o) \
$(LIB)(devstart.o) \
$(LIB)(dmc11.o) \
$(LIB)(rop.o) \
$(LIB)(ioctl.o) \
$(LIB)(fakemx.o)

all: $(LIB)
    @echo $(LIB) is now up-to-date.

$(LIB)::$(LIB2OBS)
$(LIB2OBS): $(FRC)
FRC:
    rm -f $(LIB)

clobber: cleanup
    -rm -f $(LIB) *.o

clean cleanup;;

install: all;

.PRECIOUS: $(LIB)

.s.a:
    $(AS) $(ASFLAGS) -o $*.o $<
    ar rcv $@ $*.o
    rm $*.o

#    @(#)/usr/src/cmd/make/make.tm    3.2
#    ucb/head/sys/70.mk makefile

COMPOOL = /usr/include/sys

HEADERS = \
    $(COMPOOL)/buf.h \
    $(COMPOOL)/bufx.h \
    $(COMPOOL)/conf.h \
    $(COMPOOL)/confx.h \
    $(COMPOOL)/crtctl.h \
    $(COMPOOL)/dir.h \
    $(COMPOOL)/dm11.h \
    $(COMPOOL)/elog.h \
    $(COMPOOL)/file.h \
    $(COMPOOL)/filex.h \
    $(COMPOOL)/filsys.h \
    $(COMPOOL)/ino.h \
    $(COMPOOL)/inode.h \
    $(COMPOOL)/inodex.h \
    $(COMPOOL)/ioctl.h \

```



```

$(COMPOOL)/ipcomm.h \
$(COMPOOL)/ipcommx.h \
$(COMPOOL)/lfsh.h \
$(COMPOOL)/lock.h \
$(COMPOOL)/maus.h \
$(COMPOOL)/mx.h \
$(COMPOOL)/param.h \
$(COMPOOL)/proc.h \
$(COMPOOL)/procx.h \
$(COMPOOL)/reg.h \
$(COMPOOL)/seg.h \
$(COMPOOL)/sgtty.h \
$(COMPOOL)/sigdef.h \
$(COMPOOL)/sprof.h \
$(COMPOOL)/sprofx.h \
$(COMPOOL)/stat.h \
$(COMPOOL)/syserr.h \
$(COMPOOL)/sysmes.h \
$(COMPOOL)/sysmesx.h \
$(COMPOOL)/system.h \
$(COMPOOL)/text.h \
$(COMPOOL)/textx.h \
$(COMPOOL)/timeb.h \
$(COMPOOL)/trans.h \
$(COMPOOL)/tty.h \
$(COMPOOL)/ttyx.h \
$(COMPOOL)/types.h \
$(COMPOOL)/user.h \
$(COMPOOL)/userx.h \
$(COMPOOL)/version.h \
$(COMPOOL)/votrax.h \
$(COMPOOL)/vt11.h \
$(COMPOOL)/vtmn.h

```

```

all: $(FRC) $(HEADERS)
      @echo Headers are now up-to-date.

```

```

$(HEADERS): s.$$(@F)
            $(GET) -s -p $(GFLAGS) $? > xtemp
            move xtemp 444 src sys $@

```

```

FRC:
      rm -f $(HEADERS)

```

```

.PRECIOUS: $(HEADERS)

```

```

.h~.h:
      get -s $<

```

```

.DEFAULT:
      cpmv $? 444 src sys $@

```

SDB—A Symbolic Debugger

H. P. Katseff

Bell Laboratories
Holmdel, New Jersey 07733

ABSTRACT

Sdb is a symbolic debugging program currently implemented for the languages C and F77 on the UNIX† operating system. *Sdb* allows one to interact with a debugged program at the source language level. When debugging a “core image” from an aborted program, *sdb* reports which line in the source program caused the error and allows all variables, including array and structure elements, to be accessed symbolically and displayed in the correct format.

One may place breakpoints at selected statements or single step on a line by line basis. To facilitate specification of lines in the program without a source listing, *sdb* provides a mechanism for examining the source text.

Procedures may be called directly from the debugger. This feature is useful both for testing individual procedures and for calling user-provided routines which provide formatted printout of structured data.

1. INTRODUCTION

This document describes a symbolic debugger, *sdb*, as implemented for C and F77 programs on the UNIX operating system. *Sdb* is useful both for examining “core images” of aborted programs and for providing an environment in which execution of a program can be monitored and controlled.

2. EXAMINING CORE IMAGES

In order to use *sdb*, it is necessary to compile the source program with the “-g” flag. This causes the compiler to generate additional information about the variables and statements of the compiled program. When the debug flag is specified, *sdb* can be used to obtain a trace of the called procedures at the time of the abort and interactively display the values of variables.

2.1. Invoking Sdb

A typical sequence of shell commands for debugging a core image is:

```
$ cc -g foo.c -o foo
$ foo
Bus error - core dumped
$ sdb foo
main:25:      x[i] = 0;
*
```

† UNIX is a trademark of Bell Laboratories.

The program *foo* was compiled with the “-g” flag and then executed. An error occurred which caused a core dump. *Sdb* is then invoked to examine the core dump to determine the cause of the error. It reports that the Bus error occurred in procedure *main* at line 25 (line numbers are always relative to the beginning of the file) and outputs the source text of the offending line. *Sdb* then prompts the user with a “*” indicating that it awaits a command.

It is useful to know that *sdb* has a notion of current procedure and current line. In this example, they are initially set to “main” and “25” respectively.

In the above example *sdb* was called with one argument, “foo”. In general it takes three arguments on the command line. The first is the name of the executable file which is to be debugged; it defaults to *a.out* when not specified. The second is the name of the core file, defaulting to *core* and the third is the name of the directory containing the source of the program being debugged. *Sdb* currently requires all source to reside in a single directory. The default is the working directory. In the example the second and third arguments defaulted to the correct values, so only the first was specified.

It is possible that the error occurred in a procedure which was not compiled with the debug flag. In this case, *sdb* prints the procedure name and the address at which the error occurred. The current line and procedure are set to the first line in *main*. *Sdb* will complain if *main* was not compiled with “-g” but debugging can continue for those routines compiled with the debug flag.

2.2. Printing a Stack Trace

It is often useful to obtain a listing of the procedure calls which led to the error. This is obtained with the *t* command. For example:

```
*t
sub(x=2,y=3)      [foo.c:25]
inter(i=16012)   [foo.c:96]
main(argc=1,argv=0x7ffff54,envp=0x7ffff5c) [foo.c:15]
```

This indicates that the error occurred within the procedure *sub* at line 25 in file *foo.c*. *Sub* was called with the arguments *x=2* and *y=3* from *inter* at line 96. *Inter* was called from *main* at line 15. *Main* is always called by the shell with three arguments, often referred to as *argc*, *argv* and *envp*. Note that *argv* and *envp* are pointers, so their values are printed in hexadecimal.

2.3. Examining Variables

Sdb can be used to display variables in the stopped program. Variables are displayed by typing their name followed by a slash, so

```
*errflg/
```

causes *sdb* to display the value of variable *errflg*. Unless otherwise specified, variables are assumed to be either local to or accessible from the current procedure. To specify a different procedure, use the form

```
*sub:i/
```

to display variable *i* in procedure *sub*. F77 users can specify a common block name in the same manner. Section 3.2 will explain how to change the current procedure.

Sdb supports a limited form of pattern matching for variable and procedure names. The symbol “*” is used to match any sequence of characters of a variable name and “?” to match any single character. Consider the following commands:

```
***/
*sub:y?/
**/
```

The first prints the values of all variables beginning with "x", the second prints the values of all two letter variables in procedure *sub* beginning with "y", and the last prints all variables. In the first and last examples, only variables accessible from the current procedure are printed. The command

```
**:/
```

displays the variables for each procedure on the call stack.

Sdb normally displays the variable in a format determined by its type as declared in the source program. To request a different format, a specifier is placed after the slash. The specifier consists of an optional length specification followed by the format. The length specifiers are:

- b** one byte.
- h** two bytes (half-word).
- l** four bytes (long word).

The lengths are only effective with the formats **d**, **o**, **x** and **u**. If no length is specified, the word length of the host machine, (i.e., 4 for the DEC VAX-11/780) is used. A numeric length specifier may be used for the **s** or **a** commands. These commands normally print characters until either a null is reached or 128 characters are printed. The number specifies how many characters should be printed.

There are a number of format specifiers available:

- c** character.
- d** decimal.
- u** decimal unsigned.
- o** octal.
- x** hexadecimal.
- f** 32-bit single-precision floating point.
- g** 64-bit double-precision floating point.
- s** assume variable is a string pointer and print characters until a null is reached.
- a** print characters starting at the variable's address until a null is reached.
- p** pointer to procedure.
- i** interpret as a machine-language instruction.

As an example, the variable *i* can be displayed in hexadecimal with the following command

```
*i/x
```

Sdb also knows about structures, one dimensional arrays and pointers so that all of the following commands work.

```
*array[2]/
*sym.id/
*psym->usage/
*xsym[20].p->usage/
```

The only restriction is that array subscripts must be numbers. Note that, as a special case

```
*psym->/d
```

displays the location pointed to by *psym* in decimal.

Core locations can also be displayed by specifying their absolute addresses. The command

```
*1024/
```

displays location 1024 in decimal. As in C, numbers may also be specified in octal or hexadecimal so the above command is equivalent to both of

```
*02000/
*0x400/
```

It is possible to intermix numbers and variables, so that

```
*1000.x/
```

refers to an element of a structure starting at address 1000 and

```
*1000->x/
```

refers to an element of a structure whose address is at 1000.

The address of a variable is printed with the "=" command, so

```
*i=
```

displays the address of *i*. Another feature whose usefulness will become apparent later is the command

```
*./
```

which redisplay the last variable typed.

3. SOURCE FILE DISPLAY AND MANIPULATION

Sdb has been designed to make it easy to debug a program without constant reference to a current source listing. Facilities are provided which perform context searches within the source files of the program being debugged and to display selected portions of the source files. The commands are similar to those of the UNIX editors *ed* [1] and *ex* [2]. Like these editors, *sdb* has a notion of current file and line within the file. *Sdb* also knows how the lines of a file are partitioned into procedures, so that it also has a notion of current procedure. As noted in other parts of this document, the current procedure is used by a number of *sdb* commands.

3.1. Displaying the Source File

Four command exist for displaying lines in the source file. They are useful for perusing the source program and for determining the context of the current line. The commands are:

```
p      Print the current line.
w      Window. Print a window of 10 lines around the current line.
z      Print 10 lines starting at the current line. Advance the current line by 10.
control-d Scroll. Print the next 10 lines and advance the current line by 10. This command is used to cleanly display long segments of the program.
```

When a line from a file is printed, it is preceded by its line number. This not only gives an indication of its relative position in the file, but is also used as input by some *sdb* commands.

3.2. Changing the Current Source File or Procedure

The *e* command is used to change the current source file. Either of the forms

```
*e procedure
*e file.c
```

may be used. The first causes the file containing the named procedure to become the current file and the current line becomes the first line of the procedure. The other form causes the named file to become current. In this case the current line is set to the first line of the named file. Finally, an *e* command with no argument causes the current procedure and file named to be printed.

3.3. Changing the Current Line in the Source File

As mentioned in Section 3.1, the `z` and `control-d` commands have a side effect of changing the current line in the source file. This section describes other commands that change the current line.

There are two commands for searching for instances of regular expressions in source files. They are

```
*/regular expression/
*?regular expression?
```

The first command searches forward through the file for a line containing a string that matches the regular expression and the second searches backwards. The trailing “/” and “?” may be omitted from these commands. Regular expression matching is identical to that of `ed`.

The `+` and `-` commands may be used to move the current line forwards or backwards by a specified number of lines. Typing a new-line advances the current line by one and typing a number causes that line to become the current line in the file. These commands may be catenated with the display commands so that

```
*+15z
```

advances the current line by 15 and then prints 10 lines.

4. A CONTROLLED ENVIRONMENT FOR PROGRAM TESTING

One very useful feature of *sdb* is breakpoint debugging. After entering the debugger, certain lines in the source program may be specified to be *breakpoints*. The program is then started with a *sdb* command. Execution of the program proceeds as normal until it is about to execute one of the lines at which a breakpoint has been set. The program stops and *sdb* reports which breakpoint the program is stopped at. Now, *sdb* commands may be used to display the trace of procedure calls and the values of variables. If the user is satisfied that the program is working correctly to this point, some breakpoints can be deleted and others set, and then program execution may be continued from the point where it stopped.

A useful alternative to setting breakpoints is single stepping. *Sdb* can be requested to execute the next line of the program and then stop. This feature is especially useful for testing new programs, so they can be verified on a statement by statement basis. Note that if an attempt is made to single step through a procedure which has not been compiled with the “-g” flag, execution proceeds until a statement in a procedure compiled with the debug flag is reached.

4.1. Setting and Deleting Breakpoints

Breakpoints can be set at any line in a procedure which contains executable code. The command format is:

```
*12b
*proc:12b
*proc:b
*b
```

The first form sets a breakpoint at line 12 in the current procedure. The line numbers are relative to the beginning of the file, as printed by the source file display commands. The second form sets a breakpoint at line 12 of procedure `proc` and the third sets a breakpoint at the first line of `proc`. The last sets a breakpoint at the current line.

Breakpoints are deleted similarly with the commands:

```
*12d
*proc:12d
*proc:d
```

In addition, if the command **d** is given alone, the breakpoints are deleted interactively. Each breakpoint location is printed and a line is read from the user. If the line begins with a "y" or "d", the breakpoint is deleted.

A list of the current breakpoints is printed in response to a **B** command and the **D** command deletes all breakpoints. It is sometimes desirable to have *sdb* automatically perform a sequence of commands at a breakpoint and then have execution continue. This is achieved with another form of the **b** command:

```
*12 b t;x/
```

causes both a trace-back and the value of *x* to be printed each time execution gets to line 12. The **a** command is a special case of the above command. There are two forms:

```
*proc: a
*proc:12 a
```

The first prints the procedure name and its arguments each time it is called and the second prints the source line each time it is about to be executed.

4.2. Running the Program

The **r** command is used to begin program execution. It restarts the program as if it were invoked from the shell. The command

```
*r args
```

runs the program with the given arguments, as if they had been typed on the shell command line. If no arguments are specified, then the arguments from the last execution of the program are used. To run a program with no arguments, use the **R** command.

After the program is started, execution continues until a breakpoint is encountered, a signal such as INTERRUPT or QUIT occurs or the program terminates. In all cases, after an appropriate message is printed, control returns to *sdb*.

The **c** command may be used to continue execution of a stopped program. A line number may be specified, as in:

```
*proc:12 c
```

This places a temporary breakpoint at the named line. The breakpoint is deleted when the **c** command finishes. There is also a **C** command which continues, but passes the signal which stopped the program back to the program. This is useful for testing user-written signal handlers. Execution may be continued at a specified line with the **g** command. For example,

```
*17 g
```

continues at line 17 of the current procedure. A use for this command is to avoid executing a section of code which is known to be bad. The user should not attempt to continue execution in a different procedure than that of the breakpoint.

The **s** command is used to run the program for a single line. It is useful for slowly executing the program to examine its behavior in detail. An important alternative is the **S** command. This command is like the **s** command, but does not stop within called procedures. It is often used when one is confident that the called procedure works correctly, but is interested in testing the calling routine.

4.3. Calling Procedures

It is possible to call any of the procedures of the program from the debugger. This feature is useful both for testing individual procedures with different arguments and for calling a procedure which prints structured data in a nice way. There are two ways to call a procedure:

```
*proc(arg1, arg2, ...)
*proc(arg1, arg2, ...)/
```

The first simply executes the procedure. The second is intended for calling functions: It executes the procedure and prints the value that it returns. The value is printed in decimal unless some other format is specified. Arguments to procedures may be integer, character or string constants, or values of variables which are accessible from the current procedure.

An unfortunate bug in the current implementation is that if a procedure is called when the program is *not* stopped at a breakpoint (such as when a core image is being debugged), all variables are initialized before the procedure is started. This makes it impossible to use a procedure which formats data from a dump.

5. MACHINE LANGUAGE DEBUGGING

Sdb has facilities for examining programs at the machine language level. It is possible to print the machine language statements associated with a line in the source and to place breakpoints at arbitrary addresses. *Sdb* can also be used to display or modify the contents of the machine registers.

5.1. Displaying Machine Language Statements

To display the machine language statements associated with line 25 in procedure *main*, use the command

```
*main:25?
```

The `?` command is identical to the `/` command except that it displays from text space. The default format for printing text space is the `i` format, which interprets the machine language instruction. The `control-d` command may be used to print the next 10 instructions.

Absolute addresses may be specified instead of line numbers by appending a `:"` to them so that

```
*0x1024:?
```

displays the contents of address `0x1024` in text space. Note that the command

```
*0x1024?
```

displays the instruction corresponding to line `0x1024` in the current procedure. It is also possible to set or delete a breakpoint by specifying its absolute address:

```
*0x1024:b
```

sets a breakpoint at address `0x1024`.

5.2. Manipulating Registers

The `x` command prints the values of all the registers. Also, individual registers may be named instead of variables by appending a `%` to their name, so that

```
*r3%!22
```

changes the value of register `r3` to 22.

6. OTHER COMMANDS

To exit the debugger, use the `q` command.

The `!` command is identical to that in `ed` and is used to have the shell execute a command.

It is possible to change the values of variables when the program is stopped at a breakpoint. This is done with the command

```
*variable!value
```

which sets the variable to the given value. The value may be a number, character constant or the name of another variable. If the variable is of type float or double, the value can also be a floating-point constant.

ACKNOWLEDGEMENT

I thank Bill Joy and Chuck Haley for their comments and constructive criticisms.

REFERENCES

- [1] Dolotta, T. A., Olsson, S. B., and Petruccelli, A. G. (eds.). *UNIX User's Manual*—Release 3.0, Bell Laboratories (June 1980).
- [2] Joy, W. N. *Ex Reference Manual*, Computer Science Division, University of California, Berkeley, November 1977.

Appendix 1. EXAMPLE OF USAGE

```

$ cat testdiv2.c
main(argc, argv, envp)
char **argv, **envp; {
    int i;
    i = div2(-1);
    printf("-1/2 = %d\n", i);
}
div2(i) {
    int j;
    j = i>>1;
    return(j);
}
$ cc -g testdiv2.c
$ a.out
-1/2 = -1
$ sdb
No core image          # Warning message from sdb
*/^div2                # Search for procedure "div2"
7: div2(i) {           # It starts on line 7
*z                      # Print the next few lines
7: div2(i) {
8:     int j;
9:     j = i>>1;
10:    return(j);
11: }
*div2:b                # Place a breakpoint at the beginning of "div2"
div2:9 b               # Sdb echoes proc name and line number
*r                      # Run the procedure
a.out                  # Sdb echoes command line executed
Breakpoint at         # Executions stops just before line 9
div2:9: j = i>>1;
*t                    # Print trace of subroutine calls
div2(i=-1) [testdiv2.c:9]
main(argc=1,argv=0x7ffff50,envp=0x7ffff58) [testdiv2.c:4]
*i/                   # Print i
-1
*s                    # Single step
div2:10: return(j);   # Execution stops just before line 10
*j/                   # Print j
-1
*9d                  # Delete the breakpoint
*div2(1)/             # Try running "div2" with different arguments
0
*div2(-2)/
-1
*div2(-3)/
-2
*q
$

```

A Tutorial Introduction to ADB

J. F. Maranzano

S. R. Bourne

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Debugging tools generally provide a wealth of information about the inner workings of programs. These tools have been available on UNIX† to allow users to examine “core” files that result from aborted programs. A new debugging program, ADB, provides enhanced capabilities to examine “core” and other program files in a variety of formats, run programs with embedded breakpoints and patch files.

ADB is an indispensable but complex tool for debugging crashed systems and/or programs. This document provides an introduction to ADB with examples of its use. It explains the various formatting options, techniques for debugging C programs, examples of printing file system information and patching.

1. Introduction

ADB is a new debugging program that is available on UNIX. It provides capabilities to look at “core” files resulting from aborted programs, print output in a variety of formats, patch files, and run programs with embedded breakpoints. This document provides examples of the more useful features of ADB. The reader is expected to be familiar with the basic commands on UNIX with the C language, and with References 1, 2 and 3.

2. A Quick Survey

2.1. Invocation

ADB is invoked as:

adb objfile corefile

where *objfile* is an executable UNIX file and *corefile* is a core image file. Many times this will look like:

adb a.out core

or more simply:

adb

where the defaults are *a.out* and *core* respectively. The file name minus (-) means ignore this argument as in:

adb - core

† UNIX is a trademark of Bell Laboratories.

ADB has requests for examining locations in either file. The ? request examines the contents of *objfile*, the / request examines the *corefile*. The general form of these requests is:

address ? format

or

address / format

2.2. Current Address

ADB maintains a current address, called dot, similar in function to the current pointer in the UNIX editor. When an address is entered, the current address is set to that location, so that:

0126?i

sets dot to octal 126 and prints the instruction at that address. The request:

.,10/d

prints 10 decimal numbers starting at dot. Dot ends up referring to the address of the last item printed. When used with the ? or / requests, the current address can be advanced by typing new-line; it can be decremented by typing ~.

Addresses are represented by expressions. Expressions are made up from decimal, octal, and hexadecimal integers, and symbols from the program under test. These may be combined with the operators +, -, *, % (integer division), & (bitwise and), | (bitwise inclusive or), # (round up to the next multiple), and ~ (not). (All arithmetic within ADB is 32 bits.) When typing a symbolic address for a C program, the user can type *name* or *_name*; ADB will recognize both forms.

2.3. Formats

To print data, a user specifies a collection of letters and characters that describe the format of the printout. Formats are "remembered" in the sense that typing a request without one will cause the new printout to appear in the previous format. The following are the most commonly used format letters.

b	one byte in octal
c	one byte as a character
o	one word in octal
d	one word in decimal
f	two words in floating point
i	PDP 11 instruction
s	a null terminated character string
a	the value of dot
u	one word as unsigned integer
n	print a new-line
r	print a blank space
~	backup dot

Format letters are also available for "long" values, for example, 'D' for long decimal, and 'F' for double floating point. For other formats see the ADB manual.

2.4. General Request Meanings

The general form of a request is:

address,count command modifier

which sets 'dot' to *address* and executes the command *count* times.

The following table illustrates some general ADB command meanings:

Command Meaning	
?	Print contents from <i>a.out</i> file
/	Print contents from <i>core</i> file
=	Print value of "dot"
:	Breakpoint control
\$	Miscellaneous requests
;	Request separator
!	Escape to shell

ADB catches signals, so a user cannot use a quit signal to exit from ADB. The request \$q or \$Q (or cntl-D) must be used to exit from ADB.

3. Debugging C Programs

3.1. Debugging A Core Image

Consider the C program in Figure 1. The program is used to illustrate a common error made by C programmers. The object of the program is to change the lower case "t" to upper case in the string pointed to by *charp* and then write the character string to the file indicated by argument 1. The bug shown is that the character "T" is stored in the pointer *charp* instead of the string pointed to by *charp*. Executing the program produces a core file because of an out of bounds memory reference.

ADB is invoked by:

```
adb a.out core
```

The first debugging request:

```
$c
```

is used to give a C back-trace through the subroutines called. As shown in Figure 2 only one function (*main*) was called and the arguments *argc* and *argv* have octal values 02 and 0177762 respectively. Both of these values look reasonable; 02 = two arguments, 0177762 = address on stack of parameter vector.

The next request:

```
$C
```

is used to give a C back-trace plus an interpretation of all the local variables in each function and their values in octal. The value of the variable *cc* looks incorrect since *cc* was declared as a character.

The next request:

```
$r
```

prints out the registers including the program counter and an interpretation of the instruction at that location.

The request:

```
$e
```

prints out the values of all external variables.

A map exists for each file handled by ADB. The map for the *a.out* file is referenced by ? whereas the map for *core* file is referenced by /. Furthermore, a good rule of thumb is to use ? for instructions and / for data when looking at programs. To print out information about the maps type:

```
$m
```

This produces a report of the contents of the maps. More about these maps later.

In our example, it is useful to see the contents of the string pointed to by *charp*. This is done by:

```
*charp/s
```

which says use *charp* as a pointer in the *core* file and print the information as a character string. This printout clearly shows that the character buffer was incorrectly overwritten and helps identify the error. Printing the locations around *charp* shows that the buffer is unchanged but that the pointer is destroyed. Using ADB similarly, we could print information about the arguments to a function. The request:

```
main.argc/d
```

prints the decimal *core* image value of the argument *argc* in the function *main*.

The request:

```
*main.argv,3/o
```

prints the octal values of the three consecutive cells pointed to by *argv* in the function *main*. Note that these values are the addresses of the arguments to *main*. Therefore:

```
0177770/s
```

prints the ASCII value of the first argument. Another way to print this value would have been

```
*"/s
```

The *** means ditto which remembers the last address typed, in this case *main.argc*; the *** instructs ADB to use the address field of the *core* file as a pointer.

The request:

```
.=o
```

prints the current address (not its contents) in octal which has been set to the address of the first argument. The current address, dot, is used by ADB to "remember" its current location. It allows the user to reference locations relative to the current address, for example:

```
.-10/d
```

3.2. Multiple Functions

Consider the C program illustrated in Figure 3. This program calls functions *f*, *g*, and *h* until the stack is exhausted and a core image is produced.

Again you can enter the debugger via:

```
adb
```

which assumes the names *a.out* and *core* for the executable file and core image file respectively. The request:

```
$c
```

will fill a page of back-trace references to *f*, *g*, and *h*. Figure 4 shows an abbreviated list (typing *DEL* will terminate the output and bring you back to ADB request level).

The request:

```
,5$C
```

prints the five most recent activations.

Notice that each function (*f,g,h*) has a counter of the number of times it was called.

The request:

fcnt/d

prints the decimal value of the counter for the function *f*. Similarly *gcnt* and *hcnt* could be printed. To print the value of an automatic variable, for example the decimal value of *x* in the last call of the function *h*, type:

h.x/d

It is currently not possible in the exported version to print stack frames other than the most recent activation of a function. Therefore, a user can print everything with **SC** or the occurrence of a variable in the most recent call of a function. It is possible with the **SC** request, however, to print the stack frame starting at some address as **addressSC**.

3.3. Setting Breakpoints

Consider the C program in Figure 5. This program, which changes tabs into blanks, is adapted from *Software Tools* by Kernighan and Plauger, pp. 18-27.

We will run this program under the control of ADB (see Figure 6a) by:

adb a.out -

Breakpoints are set in the program as:

address:b [request]

The requests:

settab+4:b
fopen+4:b
getc+4:b
tabpos+4:b

set breakpoints at the start of these functions. C does not generate statement labels. Therefore it is currently not possible to plant breakpoints at locations other than function entry points without a knowledge of the code generated by the C compiler. The above addresses are entered as **symbol+4** so that they will appear in any C back-trace since the first instruction of each function is a call to the C save routine (*csv*). Note that some of the functions are from the C library.

To print the location of breakpoints one types:

\$b

The display indicates a *count* field. A breakpoint is bypassed *count - 1* times before causing a stop. The *command* field indicates the ADB requests to be executed each time the breakpoint is encountered. In our example no *command* fields are present.

By displaying the original instructions at the function *settab* we see that the breakpoint is set after the *jsr* to the C save routine. We can display the instructions using the ADB request:

settab,5?ia

This request displays five instructions starting at *settab* with the addresses of each location displayed. Another variation is:

settab,5?i

which displays the instructions with only the starting address.

Notice that we accessed the addresses from the *a.out* file with the **?** command. In general when asking for a printout of multiple items, ADB will advance the current address the number of bytes necessary to satisfy the request; in the above example five instructions were displayed and the current address was advanced 18 (decimal) bytes.

To run the program one simply types:

```
:r
```

To delete a breakpoint, for instance the entry to the function *settab*, one types:

```
settab+4:d
```

To continue execution of the program from the breakpoint type:

```
:c
```

Once the program has stopped (in this case at the breakpoint for *fopen*), ADB requests can be used to display the contents of memory. For example:

```
$C
```

to display a stack trace, or:

```
tabs,3/8o
```

to print three lines of 8 locations each from the array called *tabs*. By this time (at location *fopen*) in the C program, *settab* has been called and should have set a one in every eighth location of *tabs*.

3.4. Advanced Breakpoint Usage

We continue execution of the program with:

```
:c
```

See Figure 6b. *getc* is called three times and the contents of the variable *c* in the function *main* are displayed each time. The single character on the left hand edge is the output from the C program. On the third occurrence of *getc* the program stops. We can look at the full buffer of characters by typing:

```
ibuf+6/20c
```

When we continue the program with:

```
:c
```

we hit our first breakpoint at *tabpos* since there is a tab following the "This" word of the data.

Several breakpoints of *tabpos* will occur until the program has changed the tab into equivalent blanks. Since we feel that *tabpos* is working, we can remove the breakpoint at that location by:

```
tabpos+4:d
```

If the program is continued with:

```
:c
```

it resumes normal execution after ADB prints the message

```
a.out:running
```

The UNIX quit and interrupt signals act on ADB itself rather than on the program being debugged. If such a signal occurs then the program being debugged is stopped and control is returned to ADB. The signal is saved by ADB and is passed on to the test program if:

```
:c
```

is typed. This can be useful when testing interrupt handling routines. The signal is not passed on to the test program if one types:

```
:c 0
```


Now let us reset the breakpoint at *settab* and display the instructions located there when we reach the breakpoint. This is accomplished by:

```
settab+4:b settab,5?ia ‡
```

It is also possible to execute the ADB requests for each occurrence of the breakpoint but only stop after the third occurrence by typing:

```
getc+4,3:b main.c?C ‡
```

This request will print the local variable *c* in the function *main* at each occurrence of the breakpoint. The semicolon is used to separate multiple ADB requests on a single line.

Warning: setting a breakpoint causes the value of dot to be changed; executing the program under ADB does not change dot. Therefore:

```
settab+4:b .,5?ia
fopen+4:b
```

will print the last thing dot was set to (in the example *fopen+4*) *not* the current location (*settab+4*) at which the program is executing.

A breakpoint can be overwritten without first deleting the old breakpoint. For example:

```
settab+4:b settab,5?ia; ptab/o ‡
```

could be entered after typing the above requests.

Now the display of breakpoints:

```
$b
```

shows the above request for the *settab* breakpoint. When the breakpoint at *settab* is encountered the ADB requests are executed. Note that the location at *settab+4* has been changed to plant the breakpoint; all the other locations match their original value.

Using the functions, *f*, *g* and *h* shown in Figure 3, we can follow the execution of each function by planting non-stopping breakpoints. We call ADB with the executable program of Figure 3 as follows:

```
adb ex3 -
```

Suppose we enter the following breakpoints:

```
h+4:b hcnt/d; h.hi/; h.hr/
g+4:b gcnt/d; g.gi/; g.gr/
f+4:b fcnt/d; f.fi/; f.fr/
:r
```

Each request line indicates that the variables are printed in decimal (by the specification *d*). Since the format is not changed, the *d* can be left off all but the first request.

The output in Figure 7 illustrates two points. First, the ADB requests in the breakpoint line are not examined until the program under test is run. That means any errors in those ADB requests are not detected until run time. At the location of the error ADB stops running the program.

‡ Owing to a bug in early versions of ADB these statements must be written as:

```
settab+4:b settab,5?ia;0
getc+4,3:b main.c?C;0
settab+4:b settab,5?ia; ptab/o;0
```

Note that ;0 will set dot to zero and stop at the breakpoint.

The second point is the way ADB handles register variables. ADB uses the symbol table to address variables. Register variables, like *f.fr* above, have pointers to uninitialized places on the stack. Therefore the message "symbol not found".

Another way of getting at the data in this example is to print the variables used in the call as:

```
f+4:b    fcnt/d; f.a/; f.b/; f.fi/
g+4:b    gcnt/d; g.p/; g.q/; g.gi/
:c
```

The operator / was used instead of ? to read values from the *core* file. The output for each function, as shown in Figure 7, has the same format. For the function *f*, for example, it shows the name and value of the *external* variable *fcnt*. It also shows the address on the stack and value of the variables *a*, *b* and *fi*.

Notice that the addresses on the stack will continue to decrease until no address space is left for program execution at which time (after many pages of output) the program under test aborts. A display with names would be produced by requests like the following:

```
f+4:b    fcnt/d; f.a/"a="d; f.b/"b="d; f.fi/"fi="d
```

In this format the quoted string is printed literally and the *d* produces a decimal display of the variables. The results are shown in Figure 7.

3.5. Other Breakpoint Facilities

- Arguments and change of standard input and output are passed to a program as:

```
:r arg1 arg2 ... <infile >outfile
```

This request kills any existing program under test and starts the *a.out* afresh.

- The program being debugged can be single stepped by:

```
:s
```

If necessary, this request will start up the program being debugged and stop after executing the first instruction.

- ADB allows a program to be entered at a specific address by typing:

```
address:r
```

- The count field can be used to skip the first *n* breakpoints as:

```
,n:r
```

The request:

```
,n:c
```

may also be used for skipping the first *n* breakpoints when continuing a program.

- A program can be continued at an address different from the breakpoint by:

```
address:c
```

- The program being debugged runs as a separate process and can be killed by:

```
:k
```

4. Maps

UNIX supports several executable file formats. These are used to tell the loader how to load the program file. File type 407 is the most common and is generated by a C compiler invocation such as `cc pgm.c`. A 410 file is produced by a C compiler command of the form `cc -n pgm.c`, whereas a 411 file is produced by `cc -i pgm.c`. ADB interprets these different file formats and provides access to the different segments through a set of maps (see Figure 8). To print the maps type:

\$m

In 407 files, both text (instructions) and data are intermixed. This makes it impossible for ADB to differentiate data from instructions and some of the printed symbolic addresses look incorrect; for example, printing data addresses as offsets from routines.

In 410 files (shared text), the instructions are separated from data and `?*` accesses the data part of the *a.out* file. The `?*` request tells ADB to use the second part of the map in the *a.out* file. Accessing data in the *core* file shows the data after it was modified by the execution of the program. Notice also that the data segment may have grown during program execution.

In 411 files (separated I & D space), the instructions and data are also separated. However, in this case, since data is mapped through a separate set of segmentation registers, the base of the data segment is also relative to address zero. In this case since the addresses overlap it is necessary to use the `?*` operator to access the data space of the *a.out* file. In both 410 and 411 files the corresponding core file does not contain the program text.

Figure 9 shows the display of three maps for the same program linked as a 407, 410, 411 respectively. The *b*, *e*, and *f* fields are used by ADB to map addresses into file addresses. The “*f1*” field is the length of the header at the beginning of the file (020 bytes for an *a.out* file and 02000 bytes for a *core* file). The “*f2*” field is the displacement from the beginning of the file to the data. For a 407 file with mixed text and data this is the same as the length of the header; for 410 and 411 files this is the length of the header plus the size of the text portion.

The “*b*” and “*e*” fields are the starting and ending locations for a segment. Given an address, *A*, the location in the file (either *a.out* or *core*) is calculated as:

$$\begin{aligned} b1 \leq A \leq e1 &\Rightarrow \text{file address} = (A - b1) + f1 \\ b2 \leq A \leq e2 &\Rightarrow \text{file address} = (A - b2) + f2 \end{aligned}$$

A user can access locations by using the ADB defined variables. The `$v` request prints the variables initialized by ADB:

b	base address of data segment
d	length of the data segment
s	length of the stack
t	length of the text
m	execution type (407,410,411)

In Figure 9 variables not present are zero; they can be used by expressions such as:

`<b`

in the address field. Similarly, values of variables can be changed by an assignment request such as:

`02000>b`

that sets *b* to octal 2000. These variables are useful to know if the file under examination is an executable or *core* image file.

ADB reads the header of the *core* image file to find the values for these variables. If the second file specified does not seem to be a *core* file, or if it is missing then the header of the executable file is used instead.

5. Advanced Usage

It is possible with ADB to combine formatting requests to provide elaborate displays. Below are several examples.

5.1. Formatted dump

The line:

```
<b,-1/4o4^8Cn
```

prints 4 octal words followed by their ASCII interpretation from the data space of the core image file. Broken down, the various request pieces mean:

```
<b      The base address of the data segment.
<b,-1   Print from the base address to the end of file. A negative count is
        used here and elsewhere to loop indefinitely or until some error con-
        dition (like end of file) is detected.
```

The format `4o4^8Cn` is broken down as follows:

```
4o      Print 4 octal locations.
4^      Backup the current address 4 locations (to the original start of the
        field).
8C      Print 8 consecutive characters using an escape convention; each
        character in the range 0 to 037 is printed as @ followed by the
        corresponding character in the range 0140 to 0177. An @ is printed
        as @@.
n       Print a new-line.
```

The request:

```
<b,<d/4o4^8Cn
```

could have been used instead to allow the printing to stop at the end of the data segment (`<d` provides the data segment size in bytes).

The formatting requests can be combined with ADB's ability to read in a script to produce a core image dump script. ADB is invoked as:

```
adb a.out core < dump
```

to read in a script file, *dump*, of requests. An example of such a script is:

```
120$w
4095$s
$v
=3n
$m
=3n"C Stack Back-trace"
$C
=3n"C External Variables"
$e
=3n"Registers"
$R
0$s
```

```
=3n'Data Segment'
<b,-1/8ona
```

The request **120\$w** sets the width of the output to 120 characters (normally, the width is 80 characters). ADB attempts to print addresses as:

```
symbol + offset
```

The request **4095\$s** increases the maximum permissible offset to the nearest symbolic address from 255 (default) to 4095. The request **=** can be used to print literal strings. Thus, headings are provided in this *dump* program with requests of the form:

```
=3n'C Stack Back-trace'
```

that spaces three lines and prints the literal string. The request **\$v** prints all non-zero ADB variables (see Figure 8). The request **0\$s** sets the maximum offset for symbol matches to zero thus suppressing the printing of symbolic labels in favor of octal values. Note that this is only done for the printing of the data segment. The request:

```
<b,-1/8ona
```

prints a dump from the base of the data segment to the end of file with an octal address field and eight octal numbers per line.

Figure 11 shows the results of some formatting requests on the C program of Figure 10.

5.2. Directory Dump

As another illustration (Figure 12) consider a set of requests to dump the contents of a directory (which is made up of an integer *inumber* followed by a 14 character name):

```
adb dir -
=n8t'Inum'8t'Name'
0,-1? u8t14cn
```

In this example, the **u** prints the *inumber* as an unsigned decimal integer, the **8t** means that ADB will space to the next multiple of 8 on the output line, and the **14c** prints the 14 character file name.

5.3. Ilist Dump

Similarly the contents of the *ilist* of a file system, (e.g. /dev/src, on UNIX systems distributed by the UNIX Support Group; see UNIX Programmer's Manual Section 5) could be dumped with the following set of requests:

```
adb /dev/src -
02000>b
?m <b
<b,-1?*flags'8ton'links,uid,gid'8t3bn',size'8tbrdn'addr'8t8un'times'8t2Y2na
```

In this example the value of the base for the map was changed to 02000 (by saying **?m<b**) since that is the start of an *ilist* within a file system. An artifice (**brd** above) was used to print the 24 bit size field as a byte, a space, and a decimal integer. The last access time and last modify time are printed with the **2Y** operator. Figure 12 shows portions of these requests as applied to a directory and file system.

5.4. Converting values

ADB may be used to convert values from one representation to another. For example:

```
072 = odx
```

will print

```
072      58      #3a
```

which is the octal, decimal and hexadecimal representations of 072 (octal). The format is remembered so that typing subsequent numbers will print them in the given formats. Character values may be converted similarly, for example:

```
'a' = co
```

prints

```
a      0141
```

It may also be used to evaluate expressions but be warned that all binary operators have the same precedence which is lower than that for unary operators.

6. Patching

Patching files with ADB is accomplished with the *write*, **w** or **W**, request (which is not like the *ed* editor write command). This is often used in conjunction with the *locate*, **l** or **L** request. In general, the request syntax for **l** and **w** are similar as follows:

```
?l value
```

The request **l** is used to match on two bytes, **L** is used for four bytes. The request **w** is used to write two bytes, whereas **W** writes four bytes. The **value** field in either *locate* or *write* requests is an expression. Therefore, decimal and octal numbers, or character strings are supported.

In order to modify a file, ADB must be called as:

```
adb -w file1 file2
```

When called with this option, *file1* and *file2* are created if necessary and opened for both reading and writing.

For example, consider the C program shown in Figure 10. We can change the word "This" to "The " in the executable file for this program, *ex7*, by using the following requests:

```
adb -w ex7 -
?l 'Th'
?W 'The '
```

The request **?l** starts at dot and stops at the first match of "Th" having set dot to the address of the location found. Note the use of **?** to write to the *a.out* file. The form **?*** would have been used for a 411 file.

More frequently the request will be typed as:

```
?l 'Th'; ?s
```

and locates the first occurrence of "Th" and print the entire string. Execution of this ADB request will set dot to the address of the "Th" characters.

As another example of the utility of the patching facility, consider a C program that has an internal logic flag. The user can set the flag through ADB and run the program, e.g.:

```
adb a.out -
:s arg1 arg2
flag/w 1
:c
```

The `:s` request is normally used to single step through a process or start a process in single step mode. In this case it starts `a.out` as a subprocess with arguments `arg1` and `arg2`. If there is a subprocess running ADB writes to it rather than to the file so the `w` request causes `flag` to be changed in the memory of the subprocess.

7. Anomalies

Below is a list of some strange things that users should be aware of.

1. Function calls and arguments are put on the stack by the C save routine. Putting breakpoints at the entry point to routines means that the function appears not to have been called when the breakpoint occurs.
2. When printing addresses, ADB uses either text or data symbols from the `a.out` file. This sometimes causes unexpected symbol names to be printed with data (e.g. `savr5+022`). This does not happen if `?` is used for text (instructions) and `/` for data.
3. ADB cannot handle C register variables in the most recently activated function.

8. Acknowledgements

The authors are grateful for the thoughtful comments on how to organize this document from R. B. Brandt, E. N. Pinson and B. A. Tague. D. M. Ritchie made the system changes necessary to accommodate tracing within ADB. He also participated in discussions during the writing of ADB. His earlier work with DB and CDB led to many of the features found in ADB.

9. References

1. D. M. Ritchie and K. Thompson. "The UNIX Time-Sharing System," *CACM*, July 1974.
2. B. W. Kernighan and D. M. Ritchie. *The C Programming Language*, Prentice-Hall, 1978.
3. T. A. Dolotta, S. B. Olsson, and A. G. Petruccelli (eds). *UNIX User's Manual—Release 3.0*, Bell Laboratories (June 1980).
4. B. W. Kernighan and P. J. Plauger. *Software Tools*, Addison-Wesley, 1976.

Figure 1: C program with pointer bug

```
struct buf {
    int fildes;
    int nleft;
    char *nextp;
    char buff[512];
}bb;
struct buf *obuf;

char *charp "this is a sentence.";

main(argc,argv)
int argc;
char **argv;
{
    char    cc;

    if(argc < 2) {
        printf("Input file missing\n");
        exit(8);
    }

    if((fcreat(argv[1],obuf)) < 0){
        printf("%s : not found\n", argv[1]);
        exit(8);
    }
    charp = 'T';
    printf("debug 1 %s\n",charp);
    while(cc= *charp++)
        putc(cc,obuf);
    fflush(obuf);
}
```


Figure 2: ADB output for C program of Figure 1

```

adb a.out core
$c
~main(02,0177762)
$C
~main(02,0177762)
    argc:    02
    argv:    0177762
    cc:      02124

$R
ps    0170010
pc    0204   ~main+0152
sp    0177740
r5    0177752
r4    01
r3    0
r2    0
r1    0
r0    0124
~main+0152:  mov    _obuf,(sp)
$e
savr5:    0
_obuf:    0
_cheap:   0124
_errno:   0
_fout:    0
$m
text map  `ex1`
b1 = 0          e1 = 02360          f1 = 020
b2 = 0          e2 = 02360          f2 = 020
data map  `core1`
b1 = 0          e1 = 03500          f1 = 02000
b2 = 0175400   e2 = 0200000       f2 = 05500
*cheap/s
0124:          TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTLx   Nh@x &_
~
cheap/s
_cheap:        T
_cheap+02:     this is a sentence.
_cheap+026:    Input file missing
main.argc/d
0177756:      2
*main.argv/3o
0177762:      0177770 0177776 0177777
0177770/s
0177770:      a.out
*main.argv/3o
0177762:      0177770 0177776 0177777
*/s
0177770:      a.out
.=o
              0177770
.-10/d
0177756:      2
$Q

```

Figure 3: Multiple function C program for stack trace illustration

```
int    fcnt,gcnt,hcnt;
h(x,y)
{
    int hi; register int hr;
    hi = x+1;
    hr = x-y+1;
    hcnt++;
    hj:
    f(hr,hi);
}

g(p,q)
{
    int gi; register int gr;
    gi = q-p;
    gr = q-p+1;
    gcnt++;
    gj:
    h(gr,gi);
}

f(a,b)
{
    int fi; register int fr;
    fi = a+2*b;
    fr = a+b;
    fcnt++;
    fj:
    g(fr,fi);
}

main()
{
    f(1,1);
}
```

Figure 4: ADB output for C program of Figure 3

```

adb
$c
~h(04452,04451)
~g(04453,011124)
~f(02,04451)
~h(04450,04447)
~g(04451,011120)
~f(02,04447)
~h(04446,04445)
~g(04447,011114)
~f(02,04445)
~h(04444,04443)
HIT DEL KEY
adb
,$$C
~h(04452,04451)
    x:      04452
    y:      04451
    hi:     ?
~g(04453,011124)
    p:      04453
    q:      011124
    gi:     04451
    gr:     ?
~f(02,04451)
    a:      02
    b:      04451
    fi:     011124
    fr:     04453
~h(04450,04447)
    x:      04450
    y:      04447
    hi:     04451
    hr:     02
~g(04451,011120)
    p:      04451
    q:      011120
    gi:     04447
    gr:     04450
fcnt/d
_fcnt:      1173
gcnt/d
_gcnt:      1173
hcnt/d
_hcnt:      1172
h.x/d
022004:     2346
$q

```

Figure 5: C program to decode tabs

```

#define MAXLINE    80
#define YES       1
#define NO        0
#define TABSP     8

char  input[] "data";
char  ibuf[518];
int   tabs[MAXLINE];

main()
{
    int col, *ptab;
    char c;

    ptab = tabs;
    settab(ptab); /* Set initial tab stops */
    col = 1;
    if(fopen(input,ibuf) < 0) {
        printf("%s : not found\n",input);
        exit(8);
    }
    while((c = getc(ibuf)) != -1) {
        switch(c) {
            case '\t': /* TAB */
                while(tabpos(col) != YES) {
                    putchar(' '); /* put BLANK */
                    col++;
                }
                break;
            case '\n': /* NEWLINE */
                putchar('\n');
                col = 1;
                break;
            default:
                putchar(c);
                col++;
        }
    }
}

/* Tabpos return YES if col is a tab stop */
tabpos(col)
int col;
{
    if(col > MAXLINE)
        return(YES);
    else
        return(tabs[col]);
}

/* Settab - Set initial tab stops */
settab(tabp)
int *tabp;
{
    int i;
    for(i = 0; i <= MAXLINE; i++)
        (i%TABSP) ? (tabs[i] = NO) : (tabs[i] = YES);
}

```

Figure 6a: ADB output for C program of Figure 5

```

adb a.out —
settab+4:b
fopen+4:b
getc+4:b
tabpos+4:b
$b
breakpoints
count  bkpt      command
1      ~tabpos+04
1      _getc+04
1      _fopen+04
1      ~settab+04
settab,5?ia
~settab:      jsr      r5, csv
~settab+04:   tst      -(sp)
~settab+06:   clr      0177770(r5)
~settab+012:  cmp      $0120,0177770(r5)
~settab+020:  blt      ~settab+076
~settab+022:
settab,5?i
~settab:      jsr      r5, csv
              tst      -(sp)
              clr      0177770(r5)
              cmp      $0120,0177770(r5)
              blt      ~settab+076

:r
a.out: running
breakpoint   ~settab+04:   tst      -(sp)
settab+4:d
:c
a.out: running
breakpoint   _fopen+04:   mov      04(r5), nulstr+012
$C
_fopen(02302,02472)
~main(01,0177770)
      col:      01
      c:        0
      ptab:     03500
tabs,3/8o
03500:      01      0      0      0      0      0      0      0
            01      0      0      0      0      0      0      0
            01      0      0      0      0      0      0      0

```

Figure 6b: ADB output for C program of Figure 5

```

:c
a.out: running
breakpoint  _getc+04:      mov    04(r5),r1
ibuf+6/20c
__cleanu+0202:      This  is    a test  of
:c
a.out: running
breakpoint  ~tabpos+04:    cmp    $0120,04(r5)
tabpos+4:d
settab+4:b  settab,5?ia
settab+4:b  settab,5?ia; 0
getc+4,3:b  main.c?C; 0
settab+4:b  settab,5?ia; ptab/o; 0
$b
breakpoints
count  bkpt          command
1      ~tabpos+04
3      _getc+04      main.c?C;0
1      _fopen+04
1      ~settab+04    settab,5?ia;ptab?o;0
~settab:   jsr    r5,csv
~settab+04: bpt
~settab+06: clr    0177770(r5)
~settab+012: cmp   $0120,0177770(r5)
~settab+020: blt   ~settab+076
~settab+022:
0177766:    0177770
0177744:    @~
T0177744:   T
h0177744:   h
i0177744:   i
s0177744:   s

```

Figure 7: ADB output for C program with breakpoints

```

adb ex3 -
h+4:b hcnt/d; h.hi/; h.hr/
g+4:b gcnt/d; g.gi/; g.gr/
f+4:b fcnt/d; f.fi/; f.fr/
:r
ex3: running
_fcnt: 0
0177732: 214
symbol not found
f+4:b fcnt/d; f.a/; f.b/; f.fi/
g+4:b gcnt/d; g.p/; g.q/; g.gi/
h+4:b hcnt/d; h.x/; h.y/; h.hi/
:c
ex3: running
_fcnt: 0
0177746: 1
0177750: 1
0177732: 214
_gcnt: 0
0177726: 2
0177730: 3
0177712: 214
_hcnt: 0
0177706: 2
0177710: 1
0177672: 214
_fcnt: 1
0177666: 2
0177670: 3
0177652: 214
_gcnt: 1
0177646: 5
0177650: 8
0177632: 214
HIT DEL
f+4:b fcnt/d; f.a/"a = "d; f.b/"b = "d; f.fi/"fi = "d
g+4:b gcnt/d; g.p/"p = "d; g.q/"q = "d; g.gi/"gi = "d
h+4:b hcnt/d; h.x/"x = "d; h.y/"y = "d; h.hi/"hi = "d
:r
ex3: running
_fcnt: 0
0177746: a = 1
0177750: b = 1
0177732: fi = 214
_gcnt: 0
0177726: p = 2
0177730: q = 3
0177712: gi = 214
_hcnt: 0
0177706: x = 2
0177710: y = 1
0177672: hi = 214
_fcnt: 1
0177666: a = 2
0177670: b = 3
0177652: fi = 214
HIT DEL
$q

```


Figure 9: ADB output for maps

```

adb map407 core407
$m
text map `map407`
b1 = 0          e1    = 0256          f1 = 020
b2 = 0          e2    = 0256          f2 = 020
data map `core407`
b1 = 0          e1    = 0300          f1 = 02000
b2 = 0175400   e2    = 0200000       f2 = 02300
$V
variables
d = 0300
m = 0407
s = 02400
$Q

```

```

adb map410 core410
$m
text map `map410`
b1 = 0          e1    = 0200          f1 = 020
b2 = 020000    e2    = 020116       f2 = 0220
data map `core410`
b1 = 020000    e1    = 020200       f1 = 02000
b2 = 0175400   e2    = 0200000       f2 = 02200
$V
variables
b = 020000
d = 0200
m = 0410
s = 02400
t = 0200
$Q

```

```

adb map411 core411
$m
text map `map411`
b1 = 0          e1    = 0200          f1 = 020
b2 = 0          e2    = 0116          f2 = 0220
data map `core411`
b1 = 0          e1    = 0200          f1 = 02000
b2 = 0175400   e2    = 0200000       f2 = 02200
$V
variables
d = 0200
m = 0411
s = 02400
t = 0200
$Q

```

Figure 10: Simple C program for illustrating formatting and patching

```
char  str1[]  "This is a character string";
int   one    1;
int   number 456;
long  lnum   1234;
float fpt    1.25;
char  str2[]  "This is the second character string";
main()
{
    one = 2;
}
```

Figure 11: ADB output illustrating fancy formats

```

adb map410 core410
<b,-1/8ona
020000:      0  064124  071551  064440  020163  020141  064143  071141
_str1+016:   061541  062564  020162  072163  064562  063556  0  02
_number:
_number: 0710 0  02322  040240  0  064124  071551  064440
_str2+06: 020163  064164  020145  062563  067543  062156  061440  060550
_str2+026: 060562  072143  071145  071440  071164  067151  0147 0
savr5+02: 0  0  0  0  0  0  0  0

<b,20/4o4^8Cn
020000:      0  064124  071551  064440  @`@`This i
           020163  020141  064143  071141  s a char
           061541  062564  020162  072163  acter st
           064562  063556  0  02  ring@`@`@b@`
_number: 0710 0  02322  040240  H@a@`@`R@d @@
           0  064124  071551  064440  @`@`This i
           020163  064164  020145  062563  s the se
           067543  062156  061440  060550  cond cha
           060562  072143  071145  071440  racter s
           071164  067151  0147 0  tring@`@`@`
           0  0  0  0  @`@`@`@`@`@`@`@`
           0  0  0  0  @`@`@`@`@`@`@`@`

data address not found
<b,20/4o4^8t8cna
020000:      0  064124  071551  064440  This i
_str1+06: 020163  020141  064143  071141  s a char
_str1+016: 061541  062564  020162  072163  acter st
_str1+026: 064562  063556  0  02  ring
_number:
_number: 0710 0  02322  040240  HR
_fpt+02: 0  064124  071551  064440  This i
_str2+06: 020163  064164  020145  062563  s the se
_str2+016: 067543  062156  061440  060550  cond cha
_str2+026: 060562  072143  071145  071440  racter s
_str2+036: 071164  067151  0147 0  tring
savr5+02: 0  0  0
savr5+012: 0  0  0  0

data address not found
<b,10/2b8t^2cn
020000:      0  0

_str1:      0124 0150  Th
           0151 0163  is
           040 0151  i
           0163 040  s
           0141 040  a
           0143 0150  ch
           0141 0162  ar
           0141 0143  ac
           0164 0145  te
    
```

\$Q

Figure 12: Directory and inode dumps

```

adb dir -
-ni'Inode't'Name'
0,-1?ut14cn

      Inode   Name
0:    652    .
      82     ..
      5971   cap.c
      5323   cap
      0      pp

adb /dev/src -
02000>b
?m<b
new map    ^/dev/src
b1 = 02000    e1    = 0100000000    f1 = 0
b2 = 0        e2    = 0          f2 = 0
$V
variables
b = 02000
<b,-1?'flags'8ton'links,uid,gid'8t3bn'size'8tbrdn'addr'8t8un'times'8t2Y2na
02000:      flags 073145
           links,uid,gid  0163 0164 0141
           size 0162 10356
           addr 28770    8236 25956    27766    25455    8236 25956    25206
           times 1976 Feb 5 08:34:56  1975 Dec 28 10:55:15

02040:      flags 024555
           links,uid,gid  012  0163 0164
           size 0162 25461
           addr 8308 30050    8294 25130    15216    26890    29806    10784
           times 1976 Aug 17 12:16:51 1976 Aug 17 12:16:51

02100:      flags 05173
           links,uid,gid  011  0162 0145
           size 0147 29545
           addr 25972    8306 28265    8308 25642    15216    2314 25970
           times 1977 Apr 2 08:58:01  1977 Feb 5 10:21:44

```

ADB Summary

Command Summary

a) formatted printing

? *format* print from *a.out* file according to *format*

/ *format* print from *core* file according to *format*

= *format* print the value of *dot*

?w *expr* write expression into *a.out* file

/w *expr* write expression into *core* file

?l *expr* locate expression in *a.out* file

b) breakpoint and program control

:b set breakpoint at *dot*

:c continue running program

:d delete breakpoint

:k kill the program being debugged

:r run *a.out* file under ADB control

:s single step

c) miscellaneous printing

\$b print current breakpoints

\$c C stack trace

\$e external variables

\$f floating registers

\$m print ADB segment maps

\$q exit from ADB

\$r general registers

\$s set offset for symbol match

\$v print ADB variables

\$w set output line width

d) calling the shell

! call *shell* to read rest of line

e) assignment to variables

> *name* assign dot to variable or register *name*

Format Summary

a the value of dot

b one byte in octal

c one byte as a character

d one word in decimal

f two words in floating point

i PDP 11 instruction

o one word in octal

n print a new-line

r print a blank space

s a null terminated character string

nt move to next *n* space tab

u one word as unsigned integer

x hexadecimal

Y date

^ backup dot

"..." print string

Expression Summary

a) expression components

decimal integer e.g. 256

octal integer e.g. 0277

hexadecimal e.g. #ff

symbols e.g. flag _main main.argc

variables e.g. <b

registers e.g. <pc <r0

(expression) expression grouping

b) dyadic operators

+ add

- subtract

* multiply

% integer division

& bitwise and

| bitwise or

round up to the next multiple

c) monadic operators

~ not

* contents of location

- integer negate

January 1981