Tektronix®

8560
MULTI-USER SOFTWARE
DEVELOPMENT UNIT
**NATIVE
PROGRAMMING
PACKAGE**
USERS MANUAL

Tektronix®

COMMITTED TO EXCELLENCE

This manual supports the
following TEKTRONIX products:

8560
Option   Product

4B   8560U02

These modules are
compatible with:

TNIX Version 1 (8560)

PLEASE CHECK FOR CHANGE INFORMATION
AT THE REAR OF THIS MANUAL.

8560
MULTI-USER SOFTWARE
DEVELOPMENT UNIT
NATIVE
PROGRAMMING
PACKAGE
USERS MANUAL

Tektronix, Inc.
P.O. Box 500
Beaverton, Oregon   97077

Serial Number _____

ABOUT WARRANTY AND SUPPORT FOR THIS PRODUCT

This product is provided by Tektronix as Category C software.

LIMITED RIGHTS LEGEND

RESTRICTED RIGHTS IN SOFTWARE

CONTENTS

Section 1
INTRODUCTION

TABLES

Section 1

INTRODUCTION

ABOUT THIS PRODUCT

The 8560 MUSDU Native Programming Package is a set of tools to develop software on the 8560. You can use these tools to create customized software tools and incorporate them into TNIX, the 8560's operating system. The following programming languages are provided: C, a system programming language; BASIC dialect, a simple applications language; and a relocatable assembler (no macros). The Native Programming also includes utilities for debugging, archiving, syntax checking, source formatting, stream editing, generating translators, and other software development tasks.

ABOUT THIS MANUAL

This users manual provides tutorial and reference material for use with the 8560 Native Programming Package. The following sections are included:

Installation. Tells you how to install the Native Programming Package software.

Technical Notes. Describes any limitations or special instructions for the programs, and any changes made to the programs by Tektronix.

The UNIX ® Assembler. Describes the usage and input syntax of as, the 8560 Assembler.

The C Programming Language. Describes the usage of the C programming language.

SED—A Non-Interactive Text Editor. Describes the usage of SED, a stream-oriented editor.

A Tutorial Introduction to ADB. Describes the usage of the UNIX TM. debugger ADB.

LINT—A C Program Checker. Discusses the usage and implementation of LINT.

YACC: A Compiler-Compiler. Describes YACC, a tool for describing the input to a computer program.

LEX—A Lexical Analyzer Generator. Describes the usage of LEX, a program generator designed for lexical processing of character input streams.

SOURCE OF DOCUMENTS

The tutorial and reference documents contained in Sections 4 through 9 of this manual are reprinted by permission of Bell Laboratories.

LIST OF COMMANDS

Table 1-1 contains a list of the commands included in this package, a brief description of the command's function, and a reference to more detailed information about the command.

Table 1-2 contains a list of libraries that contain useful functions and routines.

Table 1-1
8560 Native Programming Package Commands

Command	Description	Reference
adb	General purpose debugger for 8560 programs.	See section 7 of this manual; also see 8560 MUSDU Reference Manual Section 6.
ar	Archive and library maintenance program.	8560 MUSDU Reference Manual Section 6.
arcv	Converts UNIX Version 6 archives to TNIX archives.	8560 MUSDU Reference Manual Section 6.
as	8560 assembler.	See section 4 of this manual; also see 8560 MUSDU Reference Section 6.
bas	8560 BASIC (a dialect).	8560 MUSDU Reference Manual Section 6.
cb	C program formatter.	8560 MUSDU Reference Manual Section 6.
cc	8560 Native C Compiler	See section 5 of this manual; also see 8560 MUSDU Reference Manual Section 6.
join	Relational data-base operator.	8560 MUSDU Reference Manual Section 6.
ld	Loader.	8560 MUSDU Reference Manual Section 6.
lex	Generate lexical analysis programs.	See section 10 of this manual; also see 8560 MUSDU Reference Manual Section 6.
lint	A C program verifier.	See section 8 of this manual; also see 8560 MUSDU Reference Manual Section 6.

Table 1-1 (cont)

Command	Description	Reference
lorder	Find ordering relation for an object library.	8560 MUSDU Reference Manual Section 6.
nm	Print name list (symbol table).	8560 MUSDU Reference Manual Section 6.
prof	Display program execution profile data.	8560 MUSDU Reference Manual Section 6.
ranlib	Convert archives to random libraries.	8560 MUSDU Reference Manual Section 6.
sed	Stream-oriented editor.	See section 6 of this manual; also see 8560 MUSDU Reference Manual Section 6.
size	Print size of an object file.	8560 MUSDU Reference Manual Section 6.
strip	Remove symbol and relocation bits.	8560 MUSDU Reference Manual Section 6.
tsort	Topological sort.	8560 MUSDU Reference Manual Section 6.
yacc	A compiler-compiler.	See section 9 of this manual; also see 8560 MUSDU Reference Manual Section 6.

Table 1-2
8560 Native Programming Package Libraries

Library	Contents	Reference
libc	Standard routines for I/O, system calls, data manipulation, and debugging.	8560 MUSDU Reference Manual Section 3.
libm	Mathematical functions.	8560 MUSDU Reference Manual Section 3.

Section 2
INSTALLATION

TABLES

Section 2

INSTALLATION

INTRODUCTION

This section explains the procedure for installing the 8560 Native Programming Package on your 8560 system. The following information is included here: an explanation of the format of the installation disk, installation procedures, and a list of the files needed by each of the Native Programming Package commands.

INSTALLATION PROCEDURES

The Native Programming Package software resides on a flexible disk. The information on the disk consists of executable binary files in **fbr** format. You can load these programs onto your 8560 system disk as a group or you can install individual programs. To load the whole package, use the 8560 command **install**. The **install** command takes all of the information from a **fbr** format disk and loads it to the system disk. If you want to install a single program from the disk, the command install -f -x file loads the specified program from the **fbr** disk to the system disk.

For each of the Native Programming Package programs to execute properly, certain files must be on the system disk. Refer to the "Dependency Files" discussion later in this section for a complete list of these files. In order for these programs to be installed as system commands, they must be loaded while you are logged in as root.

Installing the Native Programming Package

The general procedure for installing the Native Programming Package is:

1. Log in to the 8560 as root. You must have superuser status to perform the installation.

2. Load the software installation disk into the disk drive.

3. Enter the following command to install the software:
     # install

Installing an Individual Program

The general procedure for installing a particular program from the installation disk is:

1. Log in to the 8560 as root. You must have superuser status to perform the installation.

2. Load the software installation disk into the disk drive.

3. Enter the following command to install the particular program:

   # install -f -x program

For example, to install **adb** you would enter:

   # install -f -x adb

DEPENDENCY FILES

Table 2-1 lists each program and the files that it needs for execution. These files may be installed separately to rebuild a command.

Table 2-1
Files Required for Native Programming Package Commands

Command	Files Required	
adb	/bin/adb	/bin/sh
ar	/bin/ar	/tmp
arcv	/bin/arcv	/tmp
as	/bin/as /tmp	/lib/as2
bas	/bin/bas /tmp	/bin/ed
cb	/bin/cb	
cc	/bin/cc /bin/ld /lib/c0 /tmp /lib/c2 /lib/mcrt0.o /lib/fmcrt0.o	/bin/as /lib/cpp /usr/include /lib/c1 /lib/crt0.o /lib/fcrt0.o /lib/libc.a
join	/bin/join	

Table 2-1 (cont)

Command	Files Required	
ld	/bin/ld /usr/lib/libmp.a /lib/libt4014.a /lib/libplot.a /lib/libt300s.a	/lib/libm.a /usr/lib/libdbm.a /tmp /lib/libt300.a /lib/libt450.a
lex	/bin/lex /bin/lint /usr/lib/lint2	/usr/lib/lex /usr/lib/lint1
lint	/lib/cpp /usr/lib/llib-lc /usr/lib/llib-port	/usr/tmp /usr/lib/llib-lm /usr/include
lorder	/bin/lorder /bin/echo /bin/sed /bin/join	/bin/rm /bin/nm /bin/sort
nm	/bin/nm	
prof	/bin/prof	
ranlib	/bin/ranlib /bin/sh	/bin/ar
sed	/bin/sed	
size	/bin/size	
strip	/bin/strip	/tmp
tsort	/bin/tsort	
yacc	/bin/yacc /lib/liby.a	/usr/lib/yaccpar

Section 3

TECHNICAL NOTES

This section is reserved for technical information about the 8560 MUSDU Native Programming Package. At the time of this writing, no technical notes are included. Technical notes will be incorporated into later versions of this manual, as needed.

Section 4

THE UNIX™ ASSEMBLER

INTRODUCTION

as, a PDP-11 assembler,was developed at Bell Laboratories and is licensed by Western Electric for use on the 8560. The remainder of this section is a reprint of an article describing *as*. The Technical Notes section of this manual describes the limitations of this program and any changes made to this program by Tektronix.

™UNIX is a Trademark of Bell Laboratories.

UNIX† Assembler Reference Manual

Dennis M. Ritchie

Bell Laboratories
Murray Hill, New Jersey 07974

0. Introduction

This document describes the usage and input syntax of the UNIX PDP-11 assembler *as*. The details of the PDP-11 are not described.

The input syntax of the UNIX assembler is generally similar to that of the DEC assembler PAL-11R, although its internal workings and output format are unrelated. It may be useful to read the publication DEC-11-ASDB-D, which describes PAL-11R, although naturally one must use care in assuming that its rules apply to *as*.

As is a rather ordinary assembler without macro capabilities. It produces an output file that contains relocation information and a complete symbol table; thus the output is acceptable to the UNIX link-editor *ld*, which may be used to combine the outputs of several assembler runs and to obtain object programs from libraries. The output format has been designed so that if a program contains no unresolved references to external symbols, it is executable without further processing.

1. Usage

as is used as follows:

as [−u] [−o *output*] *file₁* ...

If the optional "−u" argument is given, all undefined symbols in the current assembly will be made undefined-external. See the **.globl** directive below.

The other arguments name files which are concatenated and assembled. Thus programs may be written in several pieces and assembled together.

The output of the assembler is by default placed on the file *a.out* in the current directory; the "−o" flag causes the output to be placed on the named file. If there were no unresolved external references, and no errors detected, the output file is marked executable; otherwise, if it is produced at all, it is made non-executable.

2. Lexical conventions

Assembler tokens include identifiers (alternatively, "symbols" or "names"), temporary symbols, constants, and operators.

2.1 Identifiers

An identifier consists of a sequence of alphanumeric characters (including period ".", underscore "_", and tilde "˜" as alphanumeric) of which the first may not be numeric. Only the first eight characters are significant. When a name begins with a tilde, the tilde is discarded and that occurrence of the identifier generates a unique entry in the symbol table which can match no other occurrence of the identifier. This feature is used by the C compiler to place

---

† UNIX is a Trademark of Bell Laboratories.

names of local variables in the output symbol table without having to worry about making them unique.

2.2 Temporary symbols

A temporary symbol consists of a digit followed by "f" or "b". Temporary symbols are discussed fully in §5.1.

2.3 Constants

An octal constant consists of a sequence of digits; "8" and "9" are taken to have octal value 10 and 11. The constant is truncated to 16 bits and interpreted in two's complement notation.

A decimal constant consists of a sequence of digits terminated by a decimal point ".". The magnitude of the constant should be representable in 15 bits; i.e., be less than 32,768.

A single-character constant consists of a single quote "'" followed by an ASCII character not a new-line. Certain dual-character escape sequences are acceptable in place of the ASCII character to represent new-line and other non-graphics (see *String statements*, §5.5). The constant's value has the code for the given character in the least significant byte of the word and is null-padded on the left.

A double-character constant consists of a double quote "″" followed by a pair of ASCII characters not including new-line. Certain dual-character escape sequences are acceptable in place of either of the ASCII characters to represent new-line and other non-graphics (see *String statements*, §5.5). The constant's value has the code for the first given character in the least significant byte and that for the second character in the most significant byte.

2.4 Operators

There are several single- and double-character operators; see §6.

2.5 Blanks

Blank and tab characters may be interspersed freely between tokens, but may not be used within tokens (except character constants). A blank or tab is required to separate adjacent identifiers or constants not otherwise separated.

2.6 Comments

The character "/" introduces a comment, which extends through the end of the line on which it appears. Comments are ignored by the assembler.

3. Segments

Assembled code and data fall into three segments: the text segment, the data segment, and the bss segment. The text segment is the one in which the assembler begins, and it is the one into which instructions are typically placed. The UNIX system will, if desired, enforce the purity of the text segment of programs by trapping write operations into it. Object programs produced by the assembler must be processed by the link-editor *ld* (using its "−n" flag) if the text segment is to be write-protected. A single copy of the text segment is shared among all processes executing such a program.

The data segment is available for placing data or instructions which will be modified during execution. Anything which may go in the text segment may be put into the data segment. In programs with write-protected, sharable text segments, data segment contains the initialized but variable parts of a program. If the text segment is not pure, the data segment begins immediately after the text segment; if the text segment is pure, the data segment begins at the lowest 8K byte boundary after the text segment.

The bss segment may not contain any explicitly initialized code or data. The length of the

bss segment (like that of text or data) is determined by the high-water mark of the location counter within it. The bss segment is actually an extension of the data segment and begins immediately after it. At the start of execution of a program, the bss segment is set to 0. Typically the bss segment is set up by statements exemplified by

    lab: . = .+10

The advantage in using the bss segment for storage that starts off empty is that the initialization information need not be stored in the output file. See also *Location counter* and *Assignment statements* below.

4. The location counter

One special symbol, " . ", is the location counter. Its value at any time is the offset within the appropriate segment of the start of the statement in which it appears. The location counter may be assigned to, with the restriction that the current segment may not change; furthermore, the value of " . " may not decrease. If the effect of the assignment is to increase the value of " . ", the required number of null bytes are generated (but see *Segments* above).

5. Statements

A source program is composed of a sequence of *statements.* Statements are separated either by new-lines or by semicolons. There are five kinds of statements: null statements, expression statements, assignment statements, string statements, and keyword statements.

Any kind of statement may be preceded by one or more labels.

5.1 Labels

There are two kinds of label: name labels and numeric labels. A name label consists of a name followed by a colon (:). The effect of a name label is to assign the current value and type of the location counter " . " to the name. An error is indicated in pass 1 if the name is already defined; an error is indicated in pass 2 if the " . " value assigned changes the definition of the label.

A numeric label consists of a digit 0 to 9 followed by a colon (:). Such a label serves to define temporary symbols of the form "nb" and "nf", where n is the digit of the label. As in the case of name labels, a numeric label assigns the current value and type of " . " to the temporary symbol. However, several numeric labels with the same digit may be used within the same assembly. References of the form "nf" refer to the first numeric label "n:" *forward* from the reference; "nb" symbols refer to the first "n:" label *backward* from the reference. This sort of temporary label was introduced by Knuth [*The Art of Computer Programming, Vol 1: Fundamental Algorithms*]. Such labels tend to conserve both the symbol table space of the assembler and the inventive powers of the programmer.

5.2 Null statements

A null statement is an empty statement (which may, however, have labels). A null statement is ignored by the assembler. Common examples of null statements are empty lines or lines containing only a label.

5.3 Expression statements

An expression statement consists of an arithmetic expression not beginning with a keyword. The assembler computes its (16-bit) value and places it in the output stream, together with the appropriate relocation bits.

5.4 Assignment statements

An assignment statement consists of an identifier, an equals sign (=), and an expression. The value and type of the expression are assigned to the identifier. It is not required that the type or value be the same in pass 2 as in pass 1, nor is it an error to redefine any symbol by assignment.

Any external attribute of the expression is lost across an assignment. This means that it is not possible to declare a global symbol by assigning to it, and that it is impossible to define a symbol to be offset from a non-locally defined global symbol.

As mentioned, it is permissible to assign to the location counter " . ". It is required, however, that the type of the expression assigned be of the same type as " . ", and it is forbidden to decrease the value of " . ". In practice, the most common assignment to " . " has the form ". = . + n" for some number n; this has the effect of generating n null bytes.

5.5 String statements

A string statement generates a sequence of bytes containing ASCII characters. A string statement consists of a left string quote " < " followed by a sequence of ASCII characters not including newline, followed by a right string quote " > ". Any of the ASCII characters may be replaced by a two-character escape sequence to represent certain non-graphic characters, as follows:

\n	NL	(012)
\s	SP	(040)
\t	HT	(011)
\e	EOT	(004)
\0	NUL	(000)
\r	CR	(015)
\a	ACK	(006)
\p	PFX	(033)
\\	\	
\>	>	

The last two are included so that the escape character and the right string quote may be represented. The same escape sequences may also be used within single- and double-character constants (see §2.3 above).

5.6 Keyword statements

Keyword statements are numerically the most common type, since most machine instructions are of this sort. A keyword statement begins with one of the many predefined keywords of the assembler; the syntax of the remainder depends on the keyword. All the keywords are listed below with the syntax they require.

6. Expressions

An expression is a sequence of symbols representing a value. Its constituents are identifiers, constants, temporary symbols, operators, and brackets. Each expression has a type.

All operators in expressions are fundamentally binary in nature; if an operand is missing on the left, a 0 of absolute type is assumed. Arithmetic is two's complement and has 16 bits of precision. All operators have equal precedence, and expressions are evaluated strictly left to right except for the effect of brackets.

6.1 Expression operators

The operators are:

(blank) when there is no operand between operands, the effect is exactly the same as if a "+" had appeared.

+ addition

— subtraction

* multiplication

\/ division (note that plain " / " starts a comment)

8 bitwise **and**

| bitwise **or**

\> logical right shift

\< logical left shift

% modulo

! $a!b$ is a **or** (**not** b); i.e., the **or** of the first operand and the one's complement of the second; most common use is as a unary.

result has the value of first operand and the type of the second; most often used to define new machine instructions with syntax identical to existing instructions.

Expressions may be grouped by use of square brackets " [ ] ". (Round parentheses are reserved for address modes.)

6.2 Types

The assembler deals with a number of types of expressions. Most types are attached to keywords and used to select the routine which treats that keyword. The types likely to be met explicitly are:

undefined

Upon first encounter, each symbol is undefined. It may become undefined if it is assigned an undefined expression. It is an error to attempt to assemble an undefined expression in pass 2; in pass 1, it is not (except that certain keywords require operands which are not undefined).

undefined external

A symbol which is declared **.globl** but not defined in the current assembly is an undefined external. If such a symbol is declared, the link editor *ld* must be used to load the assembler's output with another routine that defines the undefined reference.

absolute An absolute symbol is defined ultimately from a constant. Its value is unaffected by any possible future applications of the link-editor to the output file.

text The value of a text symbol is measured with respect to the beginning of the text segment of the program. If the assembler output is link-edited, its text symbols may change in value since the program need not be the first in the link editor's output. Most text symbols are defined by appearing as labels. At the start of an assembly, the value of " . " is text 0.

data The value of a data symbol is measured with respect to the origin of the data segment of a program. Like text symbols, the value of a data symbol may change during a subsequent link-editor run since previously loaded programs may have data segments. After the first **.data** statement, the value of " . " is data 0.

bss The value of a bss symbol is measured from the beginning of the bss segment of a program. Like text and data symbols, the value of a bss symbol may change during a subsequent link-editor run, since previously loaded programs may have bss segments. After the first **.bss** statement, the value of " . " is bss 0.

external absolute, text, data, or bss

      symbols declared **.globl** but defined within an assembly as absolute, text, data, or bss symbols may be used exactly as if they were not declared **.globl**; however, their value and type are available to the link editor so that the program may be loaded with others that reference these symbols.

register

      The symbols

            r0 ... r5
            fr0 ... fr5
            sp
            pc

      are predefined as register symbols. Either they or symbols defined from them must be used to refer to the six general-purpose, six floating-point, and the 2 special-purpose machine registers. The behavior of the floating register names is identical to that of the corresponding general register names; the former are provided as a mnemonic aid.

other types

      Each keyword known to the assembler has a type which is used to select the routine which processes the associated keyword statement. The behavior of such symbols when not used as keywords is the same as if they were absolute.

6.3 Type propagation in expressions

When operands are combined by expression operators, the result has a type which depends on the types of the operands and on the operator. The rules involved are complex to state but were intended to be sensible and predictable. For purposes of expression evaluation the important types are

      undefined
      absolute
      text
      data
      bss
      undefined external
      other

The combination rules are then: If one of the operands is undefined, the result is undefined. If both operands are absolute, the result is absolute. If an absolute is combined with one of the "other types" mentioned above, or with a register expression, the result has the register or other type. As a consequence, one can refer to r3 as "r0+3". If two operands of "other type" are combined, the result has the numerically larger type An "other type" combined with an explicitly discussed type other than absolute acts like an absolute.

Further rules applying to particular operators are:

+   If one operand is text-, data-, or bss-segment relocatable, or is an undefined external, the result has the postulated type and the other operand must be absolute.

−   If the first operand is a relocatable text-, data-, or bss-segment symbol, the second operand may be absolute (in which case the result has the type of the first operand); or the second operand may have the same type as the first (in which case the result is absolute). If the first operand is external undefined, the second must be absolute. All other combinations are illegal.

^   This operator follows no other rule than that the result has the value of the first operand and the type of the second.

others

It is illegal to apply these operators to any but absolute symbols.

7. Pseudo-operations

The keywords listed below introduce statements that generate data in unusual forms or influence the later operations of the assembler. The metanotation

[stuff] ...

means that 0 or more instances of the given stuff may appear. Also, boldface tokens are literals, italic words are substitutable.

7.1 .byte *expression* [, *expression*] ...

The *expressions* in the comma-separated list are truncated to 8 bits and assembled in successive bytes. The expressions must be absolute. This statement and the string statement above are the only ones that assemble data one byte at at time.

7.2 .even

If the location counter ``.'' is odd, it is advanced by one so the next statement will be assembled at a word boundary.

7.3 .if *expression*

The *expression* must be absolute and defined in pass 1. If its value is nonzero, the .if is ignored; if zero, the statements between the .if and the matching .endif (below) are ignored. .if may be nested. The effect of .if cannot extend beyond the end of the input file in which it appears. (The statements are not totally ignored, in the following sense: .ifs and .endifs are scanned for, and moreover all names are entered in the symbol table. Thus names occurring only inside an .if will show up as undefined if the symbol table is listed.)

7.4 .endif

This statement marks the end of a conditionally-assembled section of code. See .if above.

7.5 .globl *name* [, *name*] ...

This statement makes the *names* external. If they are otherwise defined (by assignment or appearance as a label) they act within the assembly exactly as if the .globl statement were not given; however, the link editor *ld* may be used to combine this routine with other routines that refer these symbols.

Conversely, if the given symbols are not defined within the current assembly, the link editor can combine the output of this assembly with that of others which define the symbols. As discussed in §1, it is possible to force the assembler to make all otherwise undefined symbols external.

7.6 .text

7.7 .data

7.8 .bss

These three pseudo-operations cause the assembler to begin assembling into the text, data, or bss segment respectively. Assembly starts in the text segment. It is forbidden to assemble any code or data into the bss segment, but symbols may be defined and ``.'' moved about by assignment.

7.9 .comm *name* , *expression*

Provided the *name* is not defined elsewhere, this statement is equivalent to

.globl name
name = expression ^ name

That is, the type of *name* is "undefined external", and its value is *expression*. In fact the *name* behaves in the current assembly just like an undefined external. However, the link-editor *ld* has been special-cased so that all external symbols which are not otherwise defined, and which have a non-zero value, are defined to lie in the bss segment, and enough space is left after the symbol to hold *expression* bytes. All symbols which become defined in this way are located before all the explicitly defined bss-segment locations.

8. Machine instructions

Because of the rather complicated instruction and addressing structure of the PDP-11, the syntax of machine instruction statements is varied. Although the following sections give the syntax in detail, the machine handbooks should be consulted on the semantics.

8.1 Sources and Destinations

The syntax of general source and destination addresses is the same. Each must have one of the following forms, where *reg* is a register symbol, and *expr* is any sort of expression:

syntax	words	mode
reg	0	00 + *reg*
(*reg*) +	0	20 + *reg*
− (*reg*)	0	40 + *reg*
expr (*reg*)	1	60 + *reg*
(*reg*)	0	10 + *reg*
* *reg*	0	10 + *reg*
* (*reg*) +	0	30 + *reg*
* − (*reg*)	0	50 + *reg*
* (*reg*)	1	70 + *reg*
* *expr* (*reg*)	1	70 + *reg*
expr	1	67
$ *expr*	1	27
* *expr*	1	77
* $ *expr*	1	37

The *words* column gives the number of address words generated; the *mode* column gives the octal address-mode number. The syntax of the address forms is identical to that in DEC assemblers, except that "*" has been substituted for "@" and "$" for "#"; the UNIX typing conventions make "@" and "#" rather inconvenient.

Notice that mode "*reg" is identical to "(reg)"; that "*(reg)" generates an index word (namely, 0); and that addresses consisting of an unadorned expression are assembled as pc-relative references independent of the type of the expression. To force a non-relative reference, the form "*$expr" can be used, but notice that further indirection is impossible.

8.3 Simple machine instructions

The following instructions are defined as absolute symbols:

```
clc
clv
clz
cln
sec
sev
sez
sen
```

They therefore require no special syntax. The PDP-11 hardware allows more than one of the "clear" class, or alternatively more than one of the "set" class to be or-ed together; this may be expressed as follows:

```
clc | clv
```

8.4 Branch

The following instructions take an expression as operand. The expression must lie in the same segment as the reference, cannot be undefined-external, and its value cannot differ from the current location of " ." by more than 254 bytes:

br	blos	
bne	bvc	
beq	bvs	
bge	bhis	
blt	bec	(= bcc)
bgt	bcc	
ble	blo	
bpl	bcs	
bmi	bes	(= bcs)
bhi		

bes ("branch on error set") and bec ("branch on error clear") are intended to test the error bit returned by system calls (which is the c-bit).

8.5   Extended branch instructions

The following symbols are followed by an expression representing an address in the same segment as " .". If the target address is close enough, a branch-type instruction is generated; if the address is too far away, a jmp will be used.

jbr	jlos
jne	jvc
jeq	jvs
jge	jhis
jlt	jec
jgt	jcc
jle	jlo
jpl	jcs
jmi	jes
jhi	

jbr turns into a plain jmp if its target is too remote; the others (whose names are contructed by replacing the "b" in the branch instruction's name by "j") turn into the converse branch over a jmp to the target address.

8.6 Single operand instructions

The following symbols are names of single-operand machine instructions. The form of address expected is discussed in §8.1 above.

clr	sbcb
clrb	ror
com	rorb
comb	rol
inc	rolb
incb	asr
dec	asrb
decb	asl
neg	aslb
negb	jmp
adc	swab
adcb	tst
sbc	tstb

8.7 Double operand instructions

The following instructions take a general source and destination (§8.1), separated by a comma, as operands.

mov
movb
cmp
cmpb
bit
bitb
bic
bicb
bis
bisb
add
sub

8.8 Miscellaneous instructions

The following instructions have more specialized syntax. Here *reg* is a register name, *src* and *dst* a general source or destination (§8.1), and *expr* is an expression:

jsr	*reg,dst*	
rts	*reg*	
sys	*expr*	
ash	*src, reg*	(or, **als**)
ashc	*src, reg*	(or, **alsc**)
mul	*src, reg*	(or, **mpy**)
div	*src, reg*	(or, **dvd**)
xor	*reg, dst*	
sxt	*dst*	
mark	*expr*	
sob	*reg, expr*	

sys is another name for the **trap** instruction. It is used to code system calls. Its operand is required to be expressible in 6 bits. The expression in **mark** must be expressible in six bits, and the expression in **sob** must be in the same segment as " . ", must not be external-undefined, must be less than " . ", and must be within 510 bytes of " . ".

4-11

8.9 Floating-point unit instructions

The following floating-point operations are defined, with syntax as indicated:

cfcc		
setf		
setd		
seti		
setl		
clrf	*fdst*	
negf	*fdst*	
absf	*fdst*	
tstf	*fsrc*	
movf	*fsrc, freg*	(= ldf)
movf	*freg, fdst*	(= stf)
movif	*src, freg*	(= ldcif)
movfi	*freg, dst*	(= stcfi)
movof	*fsrc, freg*	(= ldcdf)
movfo	*freg, fdst*	(= stcfd)
movie	*src, freg*	(= ldexp)
movei	*freg, dst*	(= stexp)
addf	*fsrc, freg*	
subf	*fsrc, freg*	
mulf	*fsrc, freg*	
divf	*fsrc, freg*	
cmpf	*fsrc, freg*	
modf	*fsrc, freg*	
ldfps	*src*	
stfps	*dst*	
stst	*dst*	

fsrc, *fdst*, and *freg* mean floating-point source, destination, and register respectively. Their syntax is identical to that for their non-floating counterparts, but note that only floating registers 0-3 can be a *freg*.

The names of several of the operations have been changed to bring out an analogy with certain fixed-point instructions. The only strange case is **movf**, which turns into either **stf** or **ldf** depending respectively on whether its first operand is or is not a register. Warning: **ldf** sets the floating condition codes, **stf** does not.

9. Other symbols

9.1 ..

The symbol " .. " is the *relocation counter*. Just before each assembled word is placed in the output stream, the current value of this symbol is added to the word if the word refers to a text, data or bss segment location. If the output word is a pc-relative address word that refers to an absolute location, the value of " .. " is subtracted.

Thus the value of " .. " can be taken to mean the starting memory location of the program. The initial value of " .. " is 0.

The value of " .. " may be changed by assignment. Such a course of action is sometimes necessary, but the consequences should be carefully thought out. It is particularly ticklish to change " .. " midway in an assembly or to do so in a program which will be treated by the loader, which has its own notions of " .. ".

9.2 System calls

System call names are not predefined. They may be found in the file */usr/include/sys.s*

10. Diagnostics

When an input file cannot be read, its name followed by a question mark is typed and assembly ceases. When syntactic or semantic errors occur, a single-character diagnostic is typed out together with the line number and the file name in which it occurred. Errors in pass 1 cause cancellation of pass 2. The possible errors are:

)	parentheses error
]	parentheses error
>	string not terminated properly
*	indirection (*) used illegally
.	illegal assignment to " . "
A	error in address
B	branch address is odd or too remote
E	error in expression
F	error in local ("f" or "b") type symbol
G	garbage (unknown) character
I	end of file inside an .if
M	multiply defined symbol as label
O	word quantity assembled at odd address
P	phase error— " . " different in pass 1 and 2
R	relocation error
U	undefined symbol
X	syntax error

Section 5

THE C PROGRAMMING LANGUAGE

INTRODUCTION

The C programming language was developed at Bell Laboratories and is licensed by Western Electric for use on the 8560. The remainder of this section is a reprint of an article describing the C language. The Technical Notes section of this manual describes the limitations of this program and any changes made to this program by Tektronix.

The C Programming Language — Reference Manual

Dennis M. Ritchie

Bell Laboratories, Murray Hill, New Jersey

This manual is reprinted, with minor changes, from *The C Programming Language,* by Brian W. Kernighan and Dennis M. Ritchie, Prentice-Hall, Inc., 1978.

1. Introduction

This manual describes the C language on the DEC PDP-11, the DEC VAX-11, 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, these dependencies follow directly from the underlying 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, newlines, 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	8 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:

```
int       extern     else
char      register   for
float     typedef    do
double    static     while
struct    goto       switch
union     return     case
long      sizeof     default
short     break      entry
unsigned  continue
auto      if
```

The **entry** keyword is not currently implemented by any compiler but is reserved for future use. Some

---

† UNIX is a Trademark of Bell Laboratories.

implementations also reserve the words `fortran` and `asm.`

2.4 Constants

There are several kinds of constants, as listed below. Hardware characteristics which 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. The digits 8 and 9 have octal value 10 and 11 respectively. 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:

newline	NL (LF)	\n
horizontal tab	HT	\t
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.

2.5 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 an immediately following newline are ignored.

2.6 Hardware characteristics

The following table summarizes certain hardware properties which vary from machine to machine. Although these affect program portability, in practice they are less of a problem than might be thought *a priori*.

	DEC PDP-11	Honeywell 6000	IBM 370	Interdata 8/32
	ASCII	ASCII	EBCDIC	ASCII
char	8 bits	9 bits	8 bits	8 bits
int	16	36	32	32
short	16	36	16	16
long	32	36	32	32
float	32	36	32	32
double	64	72	64	64
range	$\pm10^{\pm38}$	$\pm10^{\pm38}$	$\pm10^{\pm76}$	$\pm10^{\pm76}$

The VAX-11 is identical to the PDP-11 except that integers have 32 bits.

3. Syntax notation

In the syntax notation used in this manual, syntactic categories are indicated by *italic* type, and literal words and characters in **bold** 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.

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 and int of all sizes will collectively be called *integral* types. float and double will collectively be called *floating* types.

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 by this manual, only the PDP-11 sign-extends. On the PDP-11, character variables range in value from −128 to 127; the characters of the ASCII alphabet are all positive. 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 integer or long integer 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.

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. All 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-lvalue* . *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 an lvalue naming a structure or a union, and the identifier must name a member of the structure or union. The result is an lvalue referring to the named member of the structure or union.

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. The rules given here for the use of structures and unions are not enforced strictly, in order to allow an escape from the typing mechanism. See §14.1.

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 integer 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.

> *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 (and is, on the PDP-11) 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, but the result is machine dependent unless 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.

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.

The compilers currently allow a pointer to be assigned to an integer, an integer to a pointer, and a pointer to a pointer of another type. The assignment is a pure copy operation, with no conversion. This usage is nonportable, and may produce pointers which cause addressing exceptions when used. However, it is guaranteed that assignment of the constant 0 to a pointer will produce a null pointer distinguishable from a pointer to any object.

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 a list 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-list$_{opt}$;

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-specifiers$_{opt}$
    sc-specifier decl-specifiers$_{opt}$

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, char, 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
>> *struct-or-union-specifier*
>> *typedef-name*

The words long, short, and unsigned may be thought of as adjectives; the following combinations are acceptable.

>> short int
>> long int
>> unsigned int
>> 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 and unions 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, structures, unions 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 3×5×7. 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 and union 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, 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. 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 may be the same as ordinary variables. However, names of tags and members must be mutually distinct.

Two structures may share a common initial sequence of members; that is, the same member may appear in two different structures if it has the same type in both and if all previous members are the same in both. (Actually, the compiler checks only that a name in two different structures has the same type and offset in both, but if preceding members differ the construction is nonportable.)

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.

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 become 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, *KLICKSP;
typedef struct ( double re, im; ) complex;
```

the constructions

```
MILES distance;
extern KLICKSP 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-list$_{opt}$ statement-list$_{opt}$*)
>
> *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.

5-17

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-1*$_{opt}$; *expression-2*$_{opt}$; *expression-3*$_{opt}$) *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 incrementation which 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-specifiers*$_{opt}$ *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-list*$_{opt}$)

> *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 ...". Finally, because structures, unions and functions cannot be passed to a function, it is useless to declare a formal parameter to be a structure, union or function (pointers to such objects are of course permitted).

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 blocks persists until the end of the block. The lexical scope of labels is the whole of the function in which they appear.

Because all references to the same external identifier refer to the same object (see §11.2) the compiler checks all declarations of the same external identifier for compatibility; in effect their scope is increased to the whole file 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*

(note: no trailing semicolon) causes the preprocessor to replace subsequent instances of the identifier with the given string of tokens. 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. Text inside a string or a character constant is not subject 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 standard places. Alternatively, a control line of the form

    #include *<filename>*

searches only the standard places, and not the directory of the source file.
#include's may be nested.

12.3 Conditional compilation

A compiler control line of the form

    #if *constant-expression*

checks whether the constant expression (see §15) evaluates to non-zero. 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 identifier
```

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, since auto functions are meaningless (C being incapable of compiling code into the stack); 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

There are only two things that can be done with a structure or union: name one of its members (by means of the . operator); or take its address (by unary &). Other operations, such as assigning from or to it or passing it as a parameter, draw an error message. In the future, it is expected that these operations, but not necessarily others, will be allowed.

§7.1 says that in a direct or indirect structure reference (with . or ->) the name on the right must be a member of the structure named or pointed to by the expression on the left. To allow an escape from the typing rules, this restriction is not firmly enforced by the compiler. In fact, any lvalue is allowed before ., and that lvalue is then assumed to have the form of the structure of which the name on the right is a member. Also, the expression before a -> is required only to be a pointer or an integer. If a pointer, it is assumed to point to a structure of which the name on the right is a member. If an integer, it is taken to be the absolute address, in machine storage units, of the appropriate structure.

Such constructions are non-portable.

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 $i \times j \times \cdots \times k$, then E appearing in an expression is converted to a pointer to an $(n-1)$-dimensional array with rank $j \times \cdots \times k$. 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 is measured in bytes. chars have no alignment requirements; everything else must have an even address.

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 just to their right. Thus char pointers are measured in 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 mod 2^{19}).

The IBM 370 and the Interdata 8/32 are similar. On both, addresses are measured in bytes; elementary objects must be aligned on a boundary equal to their length, so pointers to short must be 0 mod 2, to int and float 0 mod 4, and to double 0 mod 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, and sizeof expressions, possibly connected by the binary operators

$$+ \quad - \quad * \quad / \quad \% \quad \& \quad | \quad \hat{} \quad << \quad >> \quad == \quad != \quad < \quad > \quad <= \quad >=$$

or by the unary operators

$$- \quad \sim$$

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.

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

Since C is an evolving language, certain obsolete constructions may be found in older programs. Although most versions of the compiler support such anachronisms, ultimately they will disappear, 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 actually decrements x since the = and the − are adjacent, but which might easily be intended to assign −1 to 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+2)
```

resembles a function declaration closely enough to confuse the compilers.

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*
> & *expression*
> − *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*]
> *lvalue* . *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`
    struct-or-union-specifier
    typedef-name

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:
    type-decl-list function-statement

function-statement:
    (declaration-list$_{opt}$ statement-list)

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 identifier
```

Recent Changes to C

November 15, 1978

A few extensions have been made to the C language beyond what is described in the reference document ("The C Programming Language," Kernighan and Ritchie, Prentice-Hall, 1978).

1. Structure assignment

Structures may be assigned, passed as arguments to functions, and returned by functions. The types of operands taking part must be the same. Other plausible operators, such as equality comparison, have not been implemented.

There is a subtle defect in the PDP-11 implementation of functions that return structures: if an interrupt occurs during the return sequence, and the same function is called reentrantly during the interrupt, the value returned from the first call may be corrupted. The problem can occur only in the presence of true interrupts, as in an operating system or a user program that makes significant use of signals; ordinary recursive calls are quite safe.

2. Enumeration type

There is a new data type analogous to the scalar types of Pascal. To the type-specifiers in the syntax on p. 193 of the C book add

> *enum-specifier*

with syntax

> *enum-specifier:*
>   enum { *enum-list* }
>   enum *identifier* { *enum-list* }
>   enum *identifier*
>
> *enum-list:*
>   *enumerator*
>   *enum-list , enumerator*
>
> *enumerator:*
>   *identifier*
>   *identifier = constant-expression*

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, winedark };
...
enum color *cp, col;
```

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 identifiers in the enum-list are declared as constants, and may appear wherever constants are required. If no enumerators with = appear, then the values of the 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.

Enumeration tags and constants must all be distinct, and, unlike structure tags and members, are drawn from the same set as ordinary identifiers.

Objects of a given enumeration type are regarded as having a type distinct from objects of all other types, and *lint* flags type mismatches. In the PDP-11 implementation all enumeration variables are treated as if they were int.

Section 6

SED—A NON-INTERACTIVE TEXT EDITOR

INTRODUCTION

sed, a non-interactive text editor, was developed at Bell Laboratories and is licensed by Western Electric for use on the 8560. The remainder of this section is a reprint of an article describing *sed*. The Technical Notes section of this manual describes the limitations of this program and any changes made to this program by Tektronix.

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.*

August 15, 1978

---

†UNIX is a Trademark of Bell Laboratories.

SED — A Non-interactive Text Editor

Lee E. McMahon

Bell Laboratories
Murray Hill, New Jersey 07974

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.

Since only a few lines of the input reside in core 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.)

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 ('{ }')(Sec. 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 imbedded newline character, but not the newline at the end of the pattern space.

5) A period '.' matches any character except the terminal newline 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 newline 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 newlines must be hidden by a backslash character ('\') immediately preceding the new-line. The <text> argument is terminated by the first unhidden newline (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. The triggering 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 newline 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 Kubhla 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>

6-7

The <pattern> argument contains a pattern, exactly like the patterns in addresses (see 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 newline.

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 Kubhla 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 nocopy 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 nocopy mode):

    In XANadu did Kubhla Khan

and the command:

    /X/s/an/AN/gp

produces:

    In XANadu did Kubhla 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*

6-9

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 imbedded newlines; 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 imbedded newline. Pattern matches may extend across the imbedded newline(s).
>
> (2)D -- Delete first part of the pattern space
>
> > Delete up to and including the first newline character in the current pattern space. If the pattern space becomes empty (the only newline was the terminal newline), 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 newline in the pattern space.

The *P* and *D* functions are equivalent to their lower-case counterparts if there are no imbedded newlines 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* functions 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 newline.

(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 newline.

(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 adress 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]   Ken Thompson and Dennis M. Ritchie, *The UNIX Programmer's Manual.* Bell Laboratories, 1978.

Section 7

A TUTORIAL INTRODUCTION TO ADB

INTRODUCTION

adb, a debugger, was developed at Bell Laboratories and is licensed by Western Electric for use on the 8560. The remainder of this section is a reprint of an article describing *abd*. The Technical Notes section of this manual describes the limitations of this program and any changes made to this program by Tektronix.

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.

May 5, 1977

---

†UNIX is a Trademark of Bell Laboratories.

A Tutorial Introduction to ADB

J. F. Maranzano

S. R. Bourne

Bell Laboratories
Murray Hill, New Jersey 07974

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 filename minus (−) means ignore this argument as in:

adb − core

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

---

†UNIX is a Trademark of Bell Laboratories.

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 newline; it can be decremented by typing ˆ.

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 newline**
r	**print a blank space**
ˆ	**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** *a.out* **file**
/	**Print contents from** *core* **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 backtrace 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 backtrace 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 backtrace 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 \$C or the occurrence of a variable in the most recent call of a function. It is possible with the \$C request, however, to print the stack frame starting at some address as **address\$C**.

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 backtrace 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

7-7

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:

:c 0

is typed.

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

---

* Owing to a bug in early versions of ADB (including the version distributed in Generic 3 UNIX) 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.

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 is not detected until run time. At the location of the error ADB stops running the program.

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:

$$b1 \leqslant A \leqslant e1 \;\Rightarrow\; \text{file address} = (A - b1) + f1$$
$$b2 \leqslant A \leqslant e2 \;\Rightarrow\; \text{file address} = (A - b2) + f2$$

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 those variables not present are zero. Use can be made of these variables by expressions such as:

< b

in the address field. Similarly the value of the variable 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, −1/4o4ˆ8Cn

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 condition (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 newline.

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 Backtrace"
$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 Backtrace"

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 V) 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 flag could be set by the user through ADB and the program run. For example:

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.   K. Thompson and D. M. Ritchie, UNIX Programmer's Manual - 7th Edition, 1978.

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
_charp:     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
*charp/s
0124:           TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTLx        Nh@x&_
-
charp/s
_charp:         T

_charp+02:      this is a sentence.

_charp+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
$c
¯h(04452,04451)
¯g(04453,011124)
¯f(02,04451)
¯h(04450,04447)
¯g(04451,011120)
¯f(02,04447)
¯h(04446,04445)
¯g(04447,011114)
¯f(02,04445)
¯h(04444,04443)
HIT DEL KEY
adb
,5$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/"h = "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 8: ADB address maps

407 files

```
a.out    | hdr |          text+data
         |_____|_____|
              0                                 D


core     | hdr |          text+data          ......|  stack
         |_____|_____|      |
              0                              D  S        E
```

410 files (shared text)

```
a.out    | hdr |          text                |    data
         |_____|_____|_____|
              0                              T  B          D


core     | hdr |      data          ......|  stack
         |_____|_____|      |
              B                    D  S          E
```
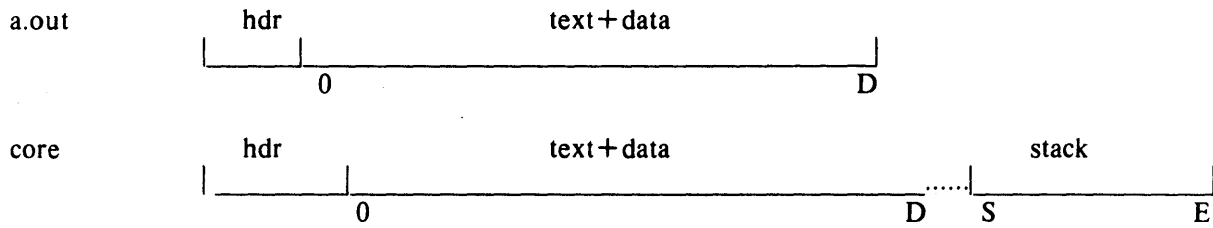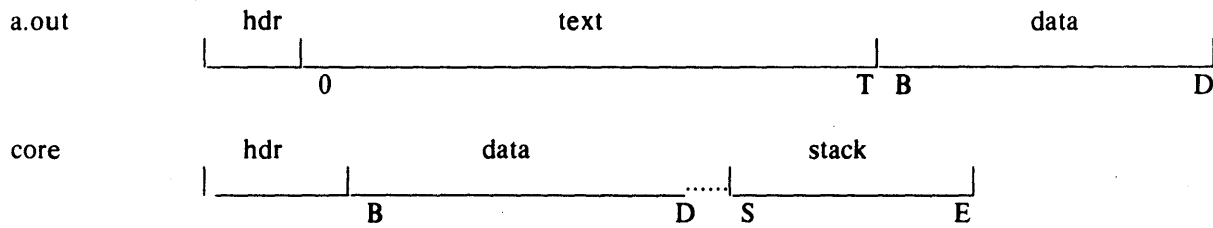
411 files (separated I and D space)

```
a.out    | hdr |          text                |    data
         |_____|_____|_____|
              0                              T  0          D


core     | hdr |      data          ......|  stack
         |_____|_____|      |
              0                    D  S          E
```
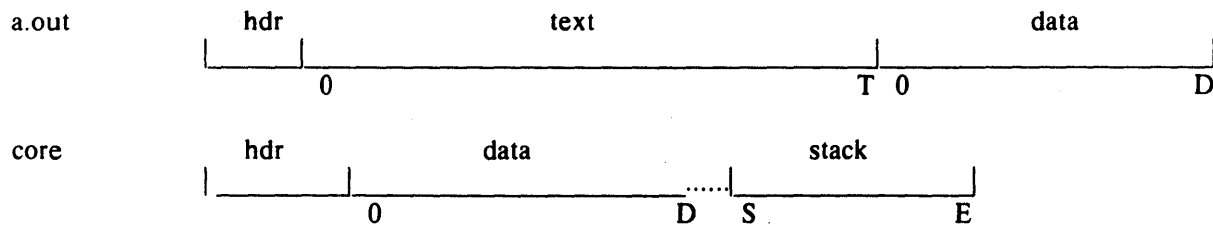
The following *adb* variables are set.

		407	410	411
b	base of data	0	B	0
d	length of data	D	D−B	D
s	length of stack	S	S	S
t	length of text	0	T	T

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      02322040240      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      02322040240      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      02322040240      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      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 −
=nt"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
         times1976 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
         times1976 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
         times1977 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 newline
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

Section 8

LINT—A C PROGRAM CHECKER

INTRODUCTION

lint, a C program checker, was developed at Bell Laboratories and is licensed by Western Electric for use on the 8560. The remainder of this section is a reprint of an article describing *lint*. The Technical Notes section of this manual describes the limitations of this program and any changes made to this program by Tektronix.

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.

July 26, 1978

Lint, a C Program Checker

S. C. Johnson

Bell Laboratories
Murray Hill, New Jersey 07974

Introduction and Usage

Suppose there are two C[1] 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 −> be a pointer to structure, the left operand of the . 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.

Nonportable 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 nonportable. 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 "nonportable character comparison".

A similar issue arises with bitfields; when assignments of constant values are made to bitfields, the field may be too small to hold the value. This is especially true because on some machines bitfields 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 bitfield 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

    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 ebcdic 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

---

†UNIX is a Trademark of Bell Laboratories.

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 Lint Up

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!)

A number of areas remain to be further developed. The checking of structures and arrays is rather inadequate; size incompatibilities go unchecked, and no attempt is made to match up structure and union declarations across files. Some stricter checking of the use of the 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," Comp. Sci. Tech. Rep. No. 32, Bell Laboratories, Murray Hill, New Jersey (July 1975).

3. M. E. Lesk, "Lex — A Lexical Analyzer Generator," Comp. Sci. Tech. Rep. No. 39, Bell Laboratories, Murray Hill, New Jersey (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,* (January 1978).

Appendix: Current Lint Options

The command currently has the form

lint [−options] files... library-descriptors...

The options are

h	Perform heuristic checks
p	Perform portability checks
v	Don't report unused arguments
u	Don't report unused or undefined externals
b	Report unreachable **break** statements.
x	Report unused external declarations
a	Report assignments of **long** to **int** or shorter.
c	Complain about questionable casts
n	No library checking is done
s	Same as **h** (for historical reasons)

Section 9

YACC—A COMPILER-COMPILER

INTRODUCTION

yacc, a compiler-compiler, was developed at Bell Laboratories and is licensed by Western Electric for use on the 8560. The remainder of this section is a reprint of an article describing *yacc*. The Technical Notes section of this manual describes the limitations of this program and any changes made to this program by Tektronix.

Yacc: Yet Another Compiler-Compiler

Stephen C. Johnson

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Computer program input generally has some structure; in fact, every computer program that does input can be thought of as defining an "input language" which it accepts. An input language may be as complex as a programming language, or as simple as a sequence of numbers. Unfortunately, usual input facilities are limited, difficult to use, and often are lax about checking their inputs for validity.

Yacc provides a general tool for describing the input to a computer program. The Yacc user specifies the structures of his input, together with code to be invoked as each such structure is recognized. Yacc turns such a specification into a subroutine that handles the input process; frequently, it is convenient and appropriate to have most of the flow of control in the user's application handled by this subroutine.

The input subroutine produced by Yacc calls a user-supplied routine to return the next basic input item. Thus, the user can specify his input in terms of individual input characters, or in terms of higher level constructs such as names and numbers. The user-supplied routine may also handle idiomatic features such as comment and continuation conventions, which typically defy easy grammatical specification.

Yacc is written in portable C. The class of specifications accepted is a very general one: LALR(1) grammars with disambiguating rules.

In addition to compilers for C, APL, Pascal, RATFOR, etc., Yacc has also been used for less conventional languages, including a phototypesetter language, several desk calculator languages, a document retrieval system, and a Fortran debugging system.

July 31, 1978

Yacc: Yet Another Compiler-Compiler

Stephen C. Johnson

Bell Laboratories
Murray Hill, New Jersey 07974

0: Introduction

Yacc provides a general tool for imposing structure on the input to a computer program. The Yacc user prepares a specification of the input process; this includes rules describing the input structure, code to be invoked when these rules are recognized, and a low-level routine to do the basic input. Yacc then generates a function to control the input process. This function, called a *parser*, calls the user-supplied low-level input routine (the *lexical analyzer*) to pick up the basic items (called *tokens*) from the input stream. These tokens are organized according to the input structure rules, called *grammar rules*; when one of these rules has been recognized, then user code supplied for this rule, an *action*, is invoked; actions have the ability to return values and make use of the values of other actions.

Yacc is written in a portable dialect of C[1] and the actions, and output subroutine, are in C as well. Moreover, many of the syntactic conventions of Yacc follow C.

The heart of the input specification is a collection of grammar rules. Each rule describes an allowable structure and gives it a name. For example, one grammar rule might be

        date : month_name day ',' year ;

Here, *date*, *month_name*, *day*, and *year* represent structures of interest in the input process; presumably, *month_name*, *day*, and *year* are defined elsewhere. The comma "," is enclosed in single quotes; this implies that the comma is to appear literally in the input. The colon and semicolon merely serve as punctuation in the rule, and have no significance in controlling the input. Thus, with proper definitions, the input

        July 4, 1776

might be matched by the above rule.

An important part of the input process is carried out by the lexical analyzer. This user routine reads the input stream, recognizing the lower level structures, and communicates these tokens to the parser. For historical reasons, a structure recognized by the lexical analyzer is called a *terminal symbol*, while the structure recognized by the parser is called a *nonterminal symbol*. To avoid confusion, terminal symbols will usually be referred to as *tokens*.

There is considerable leeway in deciding whether to recognize structures using the lexical analyzer or grammar rules. For example, the rules

        month_name : 'J' 'a' 'n' ;
        month_name : 'F' 'e' 'b' ;

            . . .

        month_name : 'D' 'e' 'c' ;

might be used in the above example. The lexical analyzer would only need to recognize individual letters, and *month_name* would be a nonterminal symbol. Such low-level rules tend to waste time and space, and may complicate the specification beyond Yacc's ability to deal with it. Usually, the lexical analyzer would recognize the month names, and return an indication that a

month_name was seen; in this case, *month_name* would be a token.

Literal characters such as "," must also be passed through the lexical analyzer, and are also considered tokens.

Specification files are very flexible. It is realively easy to add to the above example the rule

date : month '/' day '/' year ;

allowing

7 / 4 / 1776

as a synonym for

July 4, 1776

In most cases, this new rule could be "slipped in" to a working system with minimal effort, and little danger of disrupting existing input.

The input being read may not conform to the specifications. These input errors are detected as early as is theoretically possible with a left-to-right scan; thus, not only is the chance of reading and computing with bad input data substantially reduced, but the bad data can usually be quickly found. Error handling, provided as part of the input specifications, permits the reentry of bad data, or the continuation of the input process after skipping over the bad data.

In some cases, Yacc fails to produce a parser when given a set of specifications. For example, the specifications may be self contradictory, or they may require a more powerful recognition mechanism than that available to Yacc. The former cases represent design errors; the latter cases can often be corrected by making the lexical analyzer more powerful, or by rewriting some of the grammar rules. While Yacc cannot handle all possible specifications, its power compares favorably with similar systems; moreover, the constructions which are difficult for Yacc to handle are also frequently difficult for human beings to handle. Some users have reported that the discipline of formulating valid Yacc specifications for their input revealed errors of conception or design early in the program development.

The theory underlying Yacc has been described elsewhere.[2,3,4] Yacc has been extensively used in numerous practical applications, including *lint*,[5] the Portable C Compiler,[6] and a system for typesetting mathematics.[7]

The next several sections describe the basic process of preparing a Yacc specification; Section 1 describes the preparation of grammar rules, Section 2 the preparation of the user supplied actions associated with these rules, and Section 3 the preparation of lexical analyzers. Section 4 describes the operation of the parser. Section 5 discusses various reasons why Yacc may be unable to produce a parser from a specification, and what to do about it. Section 6 describes a simple mechanism for handling operator precedences in arithmetic expressions. Section 7 discusses error detection and recovery. Section 8 discusses the operating environment and special features of the parsers Yacc produces. Section 9 gives some suggestions which should improve the style and efficiency of the specifications. Section 10 discusses some advanced topics, and Section 11 gives acknowledgements. Appendix A has a brief example, and Appendix B gives a summary of the Yacc input syntax. Appendix C gives an example using some of the more advanced features of Yacc, and, finally, Appendix D describes mechanisms and syntax no longer actively supported, but provided for historical continuity with older versions of Yacc.

1: Basic Specifications

Names refer to either tokens or nonterminal symbols. Yacc requires token names to be declared as such. In addition, for reasons discussed in Section 3, it is often desirable to include the lexical analyzer as part of the specification file; it may be useful to include other programs as well. Thus, every specification file consists of three sections: the *declarations, (grammar)*

rules, and *programs*. The sections are separated by double percent "%%" marks. (The percent "%" is generally used in Yacc specifications as an escape character.)

In other words, a full specification file looks like

```
declarations
%%
rules
%%
programs
```

The declaration section may be empty. Moreover, if the programs section is omitted, the second %% mark may be omitted also; thus, the smallest legal Yacc specification is

```
%%
rules
```

Blanks, tabs, and newlines are ignored except that they may not appear in names or multi-character reserved symbols. Comments may appear wherever a name is legal; they are enclosed in /* . . . */, as in C and PL/I.

The rules section is made up of one or more grammar rules. A grammar rule has the form:

A : BODY ;

A represents a nonterminal name, and BODY represents a sequence of zero or more names and literals. The colon and the semicolon are Yacc punctuation.

Names may be of arbitrary length, and may be made up of letters, dot ".", underscore "_", and non-initial digits. Upper and lower case letters are distinct. The names used in the body of a grammar rule may represent tokens or nonterminal symbols.

A literal consists of a character enclosed in single quotes "''". As in C, the backslash "\" is an escape character within literals, and all the C escapes are recognized. Thus

'\n'	newline
'\r'	return
'\''	single quote "'"
'\\'	backslash "\"
'\t'	tab
'\b'	backspace
'\f'	form feed
'\xxx'	"xxx" in octal

For a number of technical reasons, the NUL character ('\0' or 0) should never be used in grammar rules.

If there are several grammar rules with the same left hand side, the vertical bar "|" can be used to avoid rewriting the left hand side. In addition, the semicolon at the end of a rule can be dropped before a vertical bar. Thus the grammar rules

```
A      :      B C D ;
A      :      E F ;
A      :      G ;
```

can be given to Yacc as

```
A      :      B C D
|      E F
|      G
;
```

It is not necessary that all grammar rules with the same left side appear together in the grammar rules section, although it makes the input much more readable, and easier to change.

If a nonterminal symbol matches the empty string, this can be indicated in the obvious way:

empty : ;

Names representing tokens must be declared; this is most simply done by writing

%token name1 name2 . . .

in the declarations section. (See Sections 3 , 5, and 6 for much more discussion). Every name not defined in the declarations section is assumed to represent a nonterminal symbol. Every nonterminal symbol must appear on the left side of at least one rule.

Of all the nonterminal symbols, one, called the *start symbol,* has particular importance. The parser is designed to recognize the start symbol; thus, this symbol represents the largest, most general structure described by the grammar rules. By default, the start symbol is taken to be the left hand side of the first grammar rule in the rules section. It is possible, and in fact desirable, to declare the start symbol explicitly in the declarations section using the %start keyword:

%start   symbol

The end of the input to the parser is signaled by a special token, called the *endmarker.* If the tokens up to, but not including, the endmarker form a structure which matches the start symbol, the parser function returns to its caller after the endmarker is seen; it *accepts* the input. If the endmarker is seen in any other context, it is an error.

It is the job of the user-supplied lexical analyzer to return the endmarker when appropriate; see section 3, below. Usually the endmarker represents some reasonably obvious I/O status, such as "end-of-file" or "end-of-record".

2: Actions

With each grammar rule, the user may associate actions to be performed each time the rule is recognized in the input process. These actions may return values, and may obtain the values returned by previous actions. Moreover, the lexical analyzer can return values for tokens, if desired.

An action is an arbitrary C statement, and as such can do input and output, call subprograms, and alter external vectors and variables. An action is specified by one or more statements, enclosed in curly braces "{" and "}". For example,

```
A     :      '(' B ')'
             {       hello( 1, "abc" ); }
```

and

```
XXX   :      YYY ZZZ
             {       printf("a message\n");
                     flag = 25; }
```

are grammar rules with actions.

To facilitate easy communication between the actions and the parser, the action statements are altered slightly. The symbol "dollar sign" "$" is used as a signal to Yacc in this context.

To return a value, the action normally sets the pseudo-variable "$$" to some value. For example, an action that does nothing but return the value 1 is

```
{ $$ = 1; }
```

To obtain the values returned by previous actions and the lexical analyzer, the action may use the pseudo-variables $1, $2, . . ., which refer to the values returned by the components of the right side of a rule, reading from left to right. Thus, if the rule is

```
A     :     B  C  D  ;
```

for example, then $2 has the value returned by C, and $3 the value returned by D.

As a more concrete example, consider the rule

```
expr    :        '(' expr ')' ;
```

The value returned by this rule is usually the value of the *expr* in parentheses. This can be indicated by

```
expr    :        '(' expr ')'          { $$ = $2 ; }
```

By default, the value of a rule is the value of the first element in it ($1). Thus, grammar rules of the form

```
A      :      B   ;
```

frequently need not have an explicit action.

In the examples above, all the actions came at the end of their rules. Sometimes, it is desirable to get control before a rule is fully parsed. Yacc permits an action to be written in the middle of a rule as well as at the end. This rule is assumed to return a value, accessible through the usual mechanism by the actions to the right of it. In turn, it may access the values returned by the symbols to its left. Thus, in the rule

```
A      :      B
                      { $$ = 1; }
              C
                      { x = $2;  y = $3; }
              ;
```

the effect is to set *x* to 1, and *y* to the value returned by C.

Actions that do not terminate a rule are actually handled by Yacc by manufacturing a new nonterminal symbol name, and a new rule matching this name to the empty string. The interior action is the action triggered off by recognizing this added rule. Yacc actually treats the above example as if it had been written:

```
$ACT   :        /* empty */
                        { $$ = 1; }
                ;


A      :        B $ACT C
                        { x = $2;  y = $3; }
                ;
```

In many applications, output is not done directly by the actions; rather, a data structure, such as a parse tree, is constructed in memory, and transformations are applied to it before output is generated. Parse trees are particularly easy to construct, given routines to build and maintain the tree structure desired. For example, suppose there is a C function *node*, written so that the call

```
node( L, n1, n2 )
```

creates a node with label L, and descendants n1 and n2, and returns the index of the newly created node. Then parse tree can be built by supplying actions such as:

9-7

```
expr    :       expr '+' expr
                { $$ = node( '+', $1, $3 ); }
```

in the specification.

The user may define other variables to be used by the actions. Declarations and definitions can appear in the declarations section, enclosed in the marks "%{" and "%}". These declarations and definitions have global scope, so they are known to the action statements and the lexical analyzer. For example,

```
%{   int variable = 0;   %}
```

could be placed in the declarations section, making *variable* accessible to all of the actions. The Yacc parser uses only names beginning in "yy"; the user should avoid such names.

In these examples, all the values are integers: a discussion of values of other types will be found in Section 10.

3: Lexical Analysis

The user must supply a lexical analyzer to read the input stream and communicate tokens (with values, if desired) to the parser. The lexical analyzer is an integer-valued function called *yylex*. The function returns an integer, the *token number*, representing the kind of token read. If there is a value associated with that token, it should be assigned to the external variable *yylval*.

The parser and the lexical analyzer must agree on these token numbers in order for communication between them to take place. The numbers may be chosen by Yacc, or chosen by the user. In either case, the "# define" mechanism of C is used to allow the lexical analyzer to return these numbers symbolically. For example, suppose that the token name DIGIT has been defined in the declarations section of the Yacc specification file. The relevant portion of the lexical analyzer might look like:

```
yylex(){
        extern int yylval;
        int c;
        . . .
        c = getchar();
        . . .
        switch( c ) {
                . . .
        case '0':
        case '1':
                . . .
        case '9':
                yylval = c-'0';
                return( DIGIT );
                . . .
        }
        . . .
```

The intent is to return a token number of DIGIT, and a value equal to the numerical value of the digit. Provided that the lexical analyzer code is placed in the programs section of the specification file, the identifier DIGIT will be defined as the token number associated with the token DIGIT.

This mechanism leads to clear, easily modified lexical analyzers; the only pitfall is the need to avoid using any token names in the grammar that are reserved or significant in C or the parser; for example, the use of token names *if* or *while* will almost certainly cause severe difficulties when the lexical analyzer is compiled. The token name *error* is reserved for error

handling, and should not be used naively (see Section 7).

As mentioned above, the token numbers may be chosen by Yacc or by the user. In the default situation, the numbers are chosen by Yacc. The default token number for a literal character is the numerical value of the character in the local character set. Other names are assigned token numbers starting at 257.

To assign a token number to a token (including literals), the first appearance of the token name or literal *in the declarations section* can be immediately followed by a nonnegative integer. This integer is taken to be the token number of the name or literal. Names and literals not defined by this mechanism retain their default definition. It is important that all token numbers be distinct.

For historical reasons, the endmarker must have token number 0 or negative. This token number cannot be redefined by the user; thus, all lexical analyzers should be prepared to return 0 or negative as a token number upon reaching the end of their input.

A very useful tool for constructing lexical analyzers is the *Lex* program developed by Mike Lesk.[8] These lexical analyzers are designed to work in close harmony with Yacc parsers. The specifications for these lexical analyzers use regular expressions instead of grammar rules. Lex can be easily used to produce quite complicated lexical analyzers, but there remain some languages (such as FORTRAN) which do not fit any theoretical framework, and whose lexical analyzers must be crafted by hand.

4: How the Parser Works

Yacc turns the specification file into a C program, which parses the input according to the specification given. The algorithm used to go from the specification to the parser is complex, and will not be discussed here (see the references for more information). The parser itself, however, is relatively simple, and understanding how it works, while not strictly necessary, will nevertheless make treatment of error recovery and ambiguities much more comprehensible.

The parser produced by Yacc consists of a finite state machine with a stack. The parser is also capable of reading and remembering the next input token (called the *lookahead* token). The *current state* is always the one on the top of the stack. The states of the finite state machine are given small integer labels; initially, the machine is in state 0, the stack contains only state 0, and no lookahead token has been read.

The machine has only four actions available to it, called *shift, reduce, accept*, and *error*. A move of the parser is done as follows:

1. Based on its current state, the parser decides whether it needs a lookahead token to decide what action should be done; if it needs one, and does not have one, it calls *yylex* to obtain the next token.

2. Using the current state, and the lookahead token if needed, the parser decides on its next action, and carries it out. This may result in states being pushed onto the stack, or popped off of the stack, and in the lookahead token being processed or left alone.

The *shift* action is the most common action the parser takes. Whenever a shift action is taken, there is always a lookahead token. For example, in state 56 there may be an action:

      IF   shift 34

which says, in state 56, if the lookahead token is IF, the current state (56) is pushed down on the stack, and state 34 becomes the current state (on the top of the stack). The lookahead token is cleared.

The *reduce* action keeps the stack from growing without bounds. Reduce actions are appropriate when the parser has seen the right hand side of a grammar rule, and is prepared to announce that it has seen an instance of the rule, replacing the right hand side by the left hand side. It may be necessary to consult the lookahead token to decide whether to reduce, but usually it is not; in fact, the default action (represented by a ".") is often a reduce action.

Reduce actions are associated with individual grammar rules. Grammar rules are also given small integer numbers, leading to some confusion. The action

.   reduce 18

refers to *grammar rule* 18, while the action

IF   shift 34

refers to *state* 34.

Suppose the rule being reduced is

A   :   x y z   ;

The reduce action depends on the left hand symbol (A in this case), and the number of symbols on the right hand side (three in this case). To reduce, first pop off the top three states from the stack (In general, the number of states popped equals the number of symbols on the right side of the rule). In effect, these states were the ones put on the stack while recognizing *x*, *y*, and *z*, and no longer serve any useful purpose. After popping these states, a state is uncovered which was the state the parser was in before beginning to process the rule. Using this uncovered state, and the symbol on the left side of the rule, perform what is in effect a shift of A. A new state is obtained, pushed onto the stack, and parsing continues. There are significant differences between the processing of the left hand symbol and an ordinary shift of a token, however, so this action is called a *goto* action. In particular, the lookahead token is cleared by a shift, and is not affected by a goto. In any case, the uncovered state contains an entry such as:

A   goto 20

causing state 20 to be pushed onto the stack, and become the current state.

In effect, the reduce action "turns back the clock" in the parse, popping the states off the stack to go back to the state where the right hand side of the rule was first seen. The parser then behaves as if it had seen the left side at that time. If the right hand side of the rule is empty, no states are popped off of the stack: the uncovered state is in fact the current state.

The reduce action is also important in the treatment of user-supplied actions and values. When a rule is reduced, the code supplied with the rule is executed before the stack is adjusted. In addition to the stack holding the states, another stack, running in parallel with it, holds the values returned from the lexical analyzer and the actions. When a shift takes place, the external variable *yylval* is copied onto the value stack. After the return from the user code, the reduction is carried out. When the *goto* action is done, the external variable *yyval* is copied onto the value stack. The pseudo-variables $1, $2, etc., refer to the value stack.

The other two parser actions are conceptually much simpler. The *accept* action indicates that the entire input has been seen and that it matches the specification. This action appears only when the lookahead token is the endmarker, and indicates that the parser has successfully done its job. The *error* action, on the other hand, represents a place where the parser can no longer continue parsing according to the specification. The input tokens it has seen, together with the lookahead token, cannot be followed by anything that would result in a legal input. The parser reports an error, and attempts to recover the situation and resume parsing: the error recovery (as opposed to the detection of error) will be covered in Section 7.

It is time for an example! Consider the specification

```
%token  DING  DONG  DELL
%%
rhyme   :       sound  place
        ;
sound   :       DING  DONG
        ;
place   :       DELL
        ;
```

When Yacc is invoked with the —v option, a file called *y.output* is produced, with a human-readable description of the parser. The *y.output* file corresponding to the above grammar (with some statistics stripped off the end) is:

state 0

    $accept : _rhyme $end

    DING shift 3
    . error

    rhyme goto 1
    sound goto 2

state 1

    $accept : rhyme_$end

    $end accept
    . error

state 2

    rhyme : sound_place

    DELL shift 5
    . error

    place goto 4

state 3

    sound : DING_DONG

    DONG shift 6
    . error

state 4

    rhyme : sound place_   (1)

    . reduce 1

state 5

    place : DELL_   (3)

    . reduce 3

state 6

    sound : DING DONG_   (2)

    . reduce 2

Notice that, in addition to the actions for each state, there is a description of the parsing rules being processed in each state. The _ character is used to indicate what has been seen, and what is yet to come, in each rule. Suppose the input is

DING DONG DELL

It is instructive to follow the steps of the parser while processing this input.

    Initially, the current state is state 0. The parser needs to refer to the input in order to decide between the actions available in state 0, so the first token, *DING*, is read, becoming the lookahead token. The action in state 0 on *DING* is is "shift 3", so state 3 is pushed onto the stack, and the lookahead token is cleared. State 3 becomes the current state. The next token, *DONG*, is read, becoming the lookahead token. The action in state 3 on the token *DONG* is

"shift 6", so state 6 is pushed onto the stack, and the lookahead is cleared. The stack now contains 0, 3, and 6. In state 6, without even consulting the lookahead, the parser reduces by rule 2.

$$\text{sound} \quad : \quad \text{DING} \quad \text{DONG}$$

This rule has two symbols on the right hand side, so two states, 6 and 3, are popped off of the stack, uncovering state 0. Consulting the description of state 0, looking for a goto on *sound*,

$$\text{sound} \quad \text{goto} \quad 2$$

is obtained; thus state 2 is pushed onto the stack, becoming the current state.

In state 2, the next token, *DELL*, must be read. The action is "shift 5", so state 5 is pushed onto the stack, which now has 0, 2, and 5 on it, and the lookahead token is cleared. In state 5, the only action is to reduce by rule 3. This has one symbol on the right hand side, so one state, 5, is popped off, and state 2 is uncovered. The goto in state 2 on *place*, the left side of rule 3, is state 4. Now, the stack contains 0, 2, and 4. In state 4, the only action is to reduce by rule 1. There are two symbols on the right, so the top two states are popped off, uncovering state 0 again. In state 0, there is a goto on *rhyme* causing the parser to enter state 1. In state 1, the input is read; the endmarker is obtained, indicated by "$end" in the *y.output* file. The action in state 1 when the endmarker is seen is to accept, successfully ending the parse.

The reader is urged to consider how the parser works when confronted with such incorrect strings as *DING DONG DONG, DING DONG, DING DONG DELL DELL*, etc. A few minutes spend with this and other simple examples will probably be repaid when problems arise in more complicated contexts.

5: Ambiguity and Conflicts

A set of grammar rules is *ambiguous* if there is some input string that can be structured in two or more different ways. For example, the grammar rule

$$\text{expr} \quad : \quad \text{expr} \quad '-' \quad \text{expr}$$

is a natural way of expressing the fact that one way of forming an arithmetic expression is to put two other expressions together with a minus sign between them. Unfortunately, this grammar rule does not completely specify the way that all complex inputs should be structured. For example, if the input is

$$\text{expr} \quad - \quad \text{expr} \quad - \quad \text{expr}$$

the rule allows this input to be structured as either

$$(\quad \text{expr} \quad - \quad \text{expr} \quad) \quad - \quad \text{expr}$$

or as

$$\text{expr} \quad - \quad (\quad \text{expr} \quad - \quad \text{expr} \quad)$$

(The first is called *left association*, the second *right association*).

Yacc detects such ambiguities when it is attempting to build the parser. It is instructive to consider the problem that confronts the parser when it is given an input such as

$$\text{expr} \quad - \quad \text{expr} \quad - \quad \text{expr}$$

When the parser has read the second expr, the input that it has seen:

$$\text{expr} \quad - \quad \text{expr}$$

matches the right side of the grammar rule above. The parser could *reduce* the input by applying this rule; after applying the rule; the input is reduced to *expr*(the left side of the rule). The parser would then read the final part of the input:

```
          —  expr
```

and again reduce. The effect of this is to take the left associative interpretation.

Alternatively, when the parser has seen

```
     expr  —  expr
```

it could defer the immediate application of the rule, and continue reading the input until it had seen

```
     expr  —  expr  —  expr
```

It could then apply the rule to the rightmost three symbols, reducing them to *expr* and leaving

```
     expr  —  expr
```

Now the rule can be reduced once more; the effect is to take the right associative interpretation. Thus, having read

```
     expr  —  expr
```

the parser can do two legal things, a shift or a reduction, and has no way of deciding between them. This is called a *shift / reduce conflict*. It may also happen that the parser has a choice of two legal reductions; this is called a *reduce / reduce conflict*. Note that there are never any "Shift/shift" conflicts.

When there are shift/reduce or reduce/reduce conflicts, Yacc still produces a parser. It does this by selecting one of the valid steps wherever it has a choice. A rule describing which choice to make in a given situation is called a *disambiguating rule.*

Yacc invokes two disambiguating rules by default:

1.   In a shift/reduce conflict, the default is to do the shift.

2.   In a reduce/reduce conflict, the default is to reduce by the *earlier* grammar rule (in the input sequence).

Rule 1 implies that reductions are deferred whenever there is a choice, in favor of shifts. Rule 2 gives the user rather crude control over the behavior of the parser in this situation, but reduce/reduce conflicts should be avoided whenever possible.

Conflicts may arise because of mistakes in input or logic, or because the grammar rules, while consistent, require a more complex parser than Yacc can construct. The use of actions within rules can also cause conflicts, if the action must be done before the parser can be sure which rule is being recognized. In these cases, the application of disambiguating rules is inappropriate, and leads to an incorrect parser. For this reason, Yacc always reports the number of shift/reduce and reduce/reduce conflicts resolved by Rule 1 and Rule 2.

In general, whenever it is possible to apply disambiguating rules to produce a correct parser, it is also possible to rewrite the grammar rules so that the same inputs are read but there are no conflicts. For this reason, most previous parser generators have considered conflicts to be fatal errors. Our experience has suggested that this rewriting is somewhat unnatural, and produces slower parsers; thus, Yacc will produce parsers even in the presence of conflicts.

As an example of the power of disambiguating rules, consider a fragment from a programming language involving an "if-then-else" construction:

```
     stat     :      IF  '('  cond  ')'  stat
              |      IF  '('  cond  ')'  stat  ELSE  stat
              ;
```

In these rules, *IF* and *ELSE* are tokens, *cond* is a nonterminal symbol describing conditional (logical) expressions, and *stat* is a nonterminal symbol describing statements. The first rule will be called the *simple-if* rule, and the second the *if-else* rule.

These two rules form an ambiguous construction, since input of the form

IF (C1) IF (C2) S1 ELSE S2

can be structured according to these rules in two ways:

IF (C1) {
        IF (C2) S1
        }
ELSE S2

or

IF (C1) {
        IF (C2) S1
        ELSE S2
        }

The second interpretation is the one given in most programming languages having this construct. Each *ELSE* is associated with the last preceding "un-*ELSE*'d' *IF*. In this example, consider the situation where the parser has seen

IF (C1) IF (C2) S1

and is looking at the *ELSE*. It can immediately reduce by the simple-if rule to get

IF (C1) stat

and then read the remaining input,

ELSE S2

and reduce

IF (C1) stat ELSE S2

by the if-else rule. This leads to the first of the above groupings of the input.

On the other hand, the *ELSE* may be shifted, *S2* read, and then the right hand portion of

IF (C1) IF (C2) S1 ELSE S2

can be reduced by the if-else rule to get

IF (C1) stat

which can be reduced by the simple-if rule. This leads to the second of the above groupings of the input, which is usually desired.

Once again the parser can do two valid things — there is a shift/reduce conflict. The application of disambiguating rule 1 tells the parser to shift in this case, which leads to the desired grouping.

This shift/reduce conflict arises only when there is a particular current input symbol, *ELSE*, and particular inputs already seen, such as

IF (C1) IF (C2) S1

In general, there may be many conflicts, and each one will be associated with an input symbol and a set of previously read inputs. The previously read inputs are characterized by the state of the parser.

The conflict messages of Yacc are best understood by examining the verbose (−v) option output file. For example, the output corresponding to the above conflict state might be:

23: shift/reduce conflict (shift 45, reduce 18) on ELSE

state 23

```
        stat : IF ( cond ) stat_        (18)
        stat : IF ( cond ) stat_ELSE stat

    ELSE    shift 45
      .     reduce 18
```

The first line describes the conflict, giving the state and the input symbol. The ordinary state description follows, giving the grammar rules active in the state, and the parser actions. Recall that the underline marks the portion of the grammar rules which has been seen. Thus in the example, in state 23 the parser has seen input corresponding to

        IF (cond) stat

and the two grammar rules shown are active at this time. The parser can do two possible things. If the input symbol is *ELSE*, it is possible to shift into state 45. State 45 will have, as part of its description, the line

        stat : IF (cond) stat ELSE_stat

since the *ELSE* will have been shifted in this state. Back in state 23, the alternative action, described by ".", is to be done if the input symbol is not mentioned explicitly in the above actions; thus, in this case, if the input symbol is not *ELSE*, the parser reduces by grammar rule 18:

        stat : IF '(' cond ')' stat

Once again, notice that the numbers following "shift" commands refer to other states, while the numbers following "reduce" commands refer to grammar rule numbers. In the *y.output* file, the rule numbers are printed after those rules which can be reduced. In most one states, there will be at most reduce action possible in the state, and this will be the default command. The user who encounters unexpected shift/reduce conflicts will probably want to look at the verbose output to decide whether the default actions are appropriate. In really tough cases, the user might need to know more about the behavior and construction of the parser than can be covered here. In this case, one of the theoretical references[2, 3, 4] might be consulted; the services of a local guru might also be appropriate.

6: Precedence

There is one common situation where the rules given above for resolving conflicts are not sufficient; this is in the parsing of arithmetic expressions. Most of the commonly used constructions for arithmetic expressions can be naturally described by the notion of *precedence* levels for operators, together with information about left or right associativity. It turns out that ambiguous grammars with appropriate disambiguating rules can be used to create parsers that are faster and easier to write than parsers constructed from unambiguous grammars. The basic notion is to write grammar rules of the form

        expr : expr OP expr

and

        expr : UNARY expr

for all binary and unary operators desired. This creates a very ambiguous grammar, with many parsing conflicts. As disambiguating rules, the user specifies the precedence, or binding strength, of all the operators, and the associativity of the binary operators. This information is sufficient to allow Yacc to resolve the parsing conflicts in accordance with these rules, and

construct a parser that realizes the desired precedences and associativities.

The precedences and associativities are attached to tokens in the declarations section. This is done by a series of lines beginning with a Yacc keyword: %left, %right, or %nonassoc, followed by a list of tokens. All of the tokens on the same line are assumed to have the same precedence level and associativity; the lines are listed in order of increasing precedence or binding strength. Thus,

```
%left  '+'  '−'
%left  '*'  '/'
```

describes the precedence and associativity of the four arithmetic operators. Plus and minus are left associative, and have lower precedence than star and slash, which are also left associative. The keyword %right is used to describe right associative operators, and the keyword %nonassoc is used to describe operators, like the operator .LT. in Fortran, that may not associate with themselves; thus,

A .LT. B .LT. C

is illegal in Fortran, and such an operator would be described with the keyword %nonassoc in Yacc. As an example of the behavior of these declarations, the description

```
%right  '='
%left   '+'  '−'
%left   '*'  '/'


%%


expr    :       expr  '='  expr
        |       expr  '+'  expr
        |       expr  '−'  expr
        |       expr  '*'  expr
        |       expr  '/'  expr
        |       NAME
        ;
```

might be used to structure the input

a = b = c*d − e − f*g

as follows:

a = (b = (((c*d)−e) − (f*g)))

When this mechanism is used, unary operators must, in general, be given a precedence. Sometimes a unary operator and a binary operator have the same symbolic representation, but different precedences. An example is unary and binary '−'; unary minus may be given the same strength as multiplication, or even higher, while binary minus has a lower strength than multiplication. The keyword, %prec, changes the precedence level associated with a particular grammar rule. %prec appears immediately after the body of the grammar rule, before the action or closing semicolon, and is followed by a token name or literal. It causes the precedence of the grammar rule to become that of the following token name or literal. For example, to make unary minus have the same precedence as multiplication the rules might resemble:

```
%left '+' '-'
%left '*' '/' .

%%

expr     :       expr '+' expr
         |       expr '-' expr
         |       expr '*' expr
         |       expr '/' expr
         |       '-' expr     %prec '*'
         |       NAME
         ;
```

A token declared by %left, %right, and %nonassoc need not be, but may be, declared by %token as well.

The precedences and associativities are used by Yacc to resolve parsing conflicts; they give rise to disambiguating rules. Formally, the rules work as follows:

1. The precedences and associativities are recorded for those tokens and literals that have them.

2. A precedence and associativity is associated with each grammar rule; it is the precedence and associativity of the last token or literal in the body of the rule. If the %prec construction is used, it overrides this default. Some grammar rules may have no precedence and associativity associated with them.

3. When there is a reduce/reduce conflict, or there is a shift/reduce conflict and either the input symbol or the grammar rule has no precedence and associativity, then the two disambiguating rules given at the beginning of the section are used, and the conflicts are reported.

4. If there is a shift/reduce conflict, and both the grammar rule and the input character have precedence and associativity associated with them, then the conflict is resolved in favor of the action (shift or reduce) associated with the higher precedence. If the precedences are the same, then the associativity is used; left associative implies reduce, right associative implies shift, and nonassociating implies error.

Conflicts resolved by precedence are not counted in the number of shift/reduce and reduce/reduce conflicts reported by Yacc. This means that mistakes in the specification of precedences may disguise errors in the input grammar; it is a good idea to be sparing with precedences, and use them in an essentially "cookbook" fashion, until some experience has been gained. The *y.output* file is very useful in deciding whether the parser is actually doing what was intended.

7: Error Handling

Error handling is an extremely difficult area, and many of the problems are semantic ones. When an error is found, for example, it may be necessary to reclaim parse tree storage, delete or alter symbol table entries, and, typically, set switches to avoid generating any further output.

It is seldom acceptable to stop all processing when an error is found; it is more useful to continue scanning the input to find further syntax errors. This leads to the problem of getting the parser "restarted" after an error. A general class of algorithms to do this involves discarding a number of tokens from the input string, and attempting to adjust the parser so that input can continue.

To allow the user some control over this process, Yacc provides a simple, but reasonably general, feature. The token name "error" is reserved for error handling. This name can be used in grammar rules; in effect, it suggests places where errors are expected, and recovery might take place. The parser pops its stack until it enters a state where the token "error" is

legal. It then behaves as if the token "error" were the current lookahead token, and performs the action encountered. The lookahead token is then reset to the token that caused the error. If no special error rules have been specified, the processing halts when an error is detected.

In order to prevent a cascade of error messages, the parser, after detecting an error, remains in error state until three tokens have been successfully read and shifted. If an error is detected when the parser is already in error state, no message is given, and the input token is quietly deleted.

As an example, a rule of the form

        stat   :   error

would, in effect, mean that on a syntax error the parser would attempt to skip over the statement in which the error was seen. More precisely, the parser will scan ahead, looking for three tokens that might legally follow a statement, and start processing at the first of these; if the beginnings of statements are not sufficiently distinctive, it may make a false start in the middle of a statement, and end up reporting a second error where there is in fact no error.

Actions may be used with these special error rules. These actions might attempt to reinitialize tables, reclaim symbol table space, etc.

Error rules such as the above are very general, but difficult to control. Somewhat easier are rules such as

        stat   :   error ';'

Here, when there is an error, the parser attempts to skip over the statement, but will do so by skipping to the next ';'. All tokens after the error and before the next ';' cannot be shifted, and are discarded. When the ';' is seen, this rule will be reduced, and any "cleanup" action associated with it performed.

Another form of error rule arises in interactive applications, where it may be desirable to permit a line to be reentered after an error. A possible error rule might be

        input   :   error '\n' { printf("Reenter last line: "); } input
                        { $$ = $4; }

There is one potential difficulty with this approach; the parser must correctly process three input tokens before it admits that it has correctly resynchronized after the error. If the reentered line contains an error in the first two tokens, the parser deletes the offending tokens, and gives no message; this is clearly unacceptable. For this reason, there is a mechanism that can be used to force the parser to believe that an error has been fully recovered from. The statement

        yyerrok ;

in an action resets the parser to its normal mode. The last example is better written

        input   :   error '\n'
                        {   yyerrok;
                                printf("Reenter last line: ");   }
                        input
                        { $$ = $4; }
                ;

As mentioned above, the token seen immediately after the "error" symbol is the input token at which the error was discovered. Sometimes, this is inappropriate; for example, an error recovery action might take upon itself the job of finding the correct place to resume input. In this case, the previous lookahead token must be cleared. The statement

        yyclearin ;

in an action will have this effect. For example, suppose the action after error were to call some

9-19

sophisticated resynchronization routine, supplied by the user, that attempted to advance the input to the beginning of the next valid statement. After this routine was called, the next token returned by yylex would presumably be the first token in a legal statement; the old, illegal token must be discarded, and the error state reset. This could be done by a rule like

```
stat    :       error
                {       resynch();
                        yyerrok ;
                        yyclearin ;   }

        ;
```

These mechanisms are admittedly crude, but do allow for a simple, fairly effective recovery of the parser from many errors; moreover, the user can get control to deal with the error actions required by other portions of the program.

8: The Yacc Environment

When the user inputs a specification to Yacc, the output is a file of C programs, called *y.tab.c* on most systems (due to local file system conventions, the names may differ from installation to installation). The function produced by Yacc is called *yyparse*; it is an integer valued function. When it is called, it in turn repeatedly calls *yylex*, the lexical analyzer supplied by the user (see Section 3) to obtain input tokens. Eventually, either an error is detected, in which case (if no error recovery is possible) *yyparse* returns the value 1, or the lexical analyzer returns the endmarker token and the parser accepts. In this case, *yyparse* returns the value 0.

The user must provide a certain amount of environment for this parser in order to obtain a working program. For example, as with every C program, a program called *main* must be defined, that eventually calls *yyparse*. In addition, a routine called *yyerror* prints a message when a syntax error is detected.

These two routines must be supplied in one form or another by the user. To ease the initial effort of using Yacc, a library has been provided with default versions of *main* and *yyerror*. The name of this library is system dependent; on many systems the library is accessed by a −ly argument to the loader. To show the triviality of these default programs, the source is given below:

```
main(){
        return( yyparse() );
        }
```

and

```
# include <stdio.h>

yyerror(s) char *s; {
        fprintf( stderr, "%s\n", s );
        }
```

The argument to *yyerror* is a string containing an error message, usually the string "syntax error". The average application will want to do better than this. Ordinarily, the program should keep track of the input line number, and print it along with the message when a syntax error is detected. The external integer variable *yychar* contains the lookahead token number at the time the error was detected; this may be of some interest in giving better diagnostics. Since the *main* program is probably supplied by the user (to read arguments, etc.) the Yacc library is useful only in small projects, or in the earliest stages of larger ones.

The external integer variable *yydebug* is normally set to 0. If it is set to a nonzero value, the parser will output a verbose description of its actions, including a discussion of which input symbols have been read, and what the parser actions are. Depending on the operating environment, it may be possible to set this variable by using a debugging system.

9: Hints for Preparing Specifications

This section contains miscellaneous hints on preparing efficient, easy to change, and clear specifications. The individual subsections are more or less independent.

Input Style

It is difficult to provide rules with substantial actions and still have a readable specification file. The following style hints owe much to Brian Kernighan.

a. Use all capital letters for token names, all lower case letters for nonterminal names. This rule comes under the heading of "knowing who to blame when things go wrong."

b. Put grammar rules and actions on separate lines. This allows either to be changed without an automatic need to change the other.

c. Put all rules with the same left hand side together. Put the left hand side in only once, and let all following rules begin with a vertical bar.

d. Put a semicolon only after the last rule with a given left hand side, and put the semicolon on a separate line. This allows new rules to be easily added.

e. Indent rule bodies by two tab stops, and action bodies by three tab stops.

The example in Appendix A is written following this style, as are the examples in the text of this paper (where space permits). The user must make up his own mind about these stylistic questions; the central problem, however, is to make the rules visible through the morass of action code.

Left Recursion

The algorithm used by the Yacc parser encourages so called "left recursive" grammar rules: rules of the form

        name   :   name rest_of_rule ;

These rules frequently arise when writing specifications of sequences and lists:

        list   :   item
                | list ',' item
                ;

and

        seq   :   item
                | seq item
                ;

In each of these cases, the first rule will be reduced for the first item only, and the second rule will be reduced for the second and all succeeding items.

With right recursive rules, such as

        seq   :   item
                | item seq
                ;

the parser would be a bit bigger, and the items would be seen, and reduced, from right to left. More seriously, an internal stack in the parser would be in danger of overflowing if a very long sequence were read. Thus, the user should use left recursion wherever reasonable.

It is worth considering whether a sequence with zero elements has any meaning, and if so, consider writing the sequence specification with an empty rule:

```
seq     :       /* empty */
        |       seq  item
        ;
```

Once again, the first rule would always be reduced exactly once, before the first item was read, and then the second rule would be reduced once for each item read. Permitting empty sequences often leads to increased generality. However, conflicts might arise if Yacc is asked to decide which empty sequence it has seen, when it hasn't seen enough to know!

Lexical Tie-ins

Some lexical decisions depend on context. For example, the lexical analyzer might want to delete blanks normally, but not within quoted strings. Or names might be entered into a symbol table in declarations, but not in expressions.

One way of handling this situation is to create a global flag that is examined by the lexical analyzer, and set by actions. For example, suppose a program consists of 0 or more declarations, followed by 0 or more statements. Consider:

```
%{
        int dflag;
%}
        ... other declarations ...

%%

prog    :       decls  stats
        ;

decls   :       /* empty */
                        {       dflag = 1;  }
        |       decls  declaration
        ;

stats   :       /* empty */
                        {       dflag = 0;  }
        |       stats  statement
        ;

        ... other rules ...
```

The flag *dflag* is now 0 when reading statements, and 1 when reading declarations, *except for the first token in the first statement.* This token must be seen by the parser before it can tell that the declaration section has ended and the statements have begun. In many cases, this single token exception does not affect the lexical scan.

This kind of "backdoor" approach can be elaborated to a noxious degree. Nevertheless, it represents a way of doing some things that are difficult, if not impossible, to do otherwise.

Reserved Words

Some programming languages permit the user to use words like "if", which are normally reserved, as label or variable names, provided that such use does not conflict with the legal use of these names in the programming language. This is extremely hard to do in the framework of Yacc; it is difficult to pass information to the lexical analyzer telling it "this instance of 'if' is a keyword, and that instance is a variable". The user can make a stab at it, using the mechanism described in the last subsection, but it is difficult.

A number of ways of making this easier are under advisement. Until then, it is better
that the keywords be *reserved*; that is, be forbidden for use as variable names. There are

powerful stylistic reasons for preferring this, anyway.

10: Advanced Topics

This section discusses a number of advanced features of Yacc.

Simulating Error and Accept in Actions

The parsing actions of error and accept can be simulated in an action by use of macros YYACCEPT and YYERROR. YYACCEPT causes *yyparse* to return the value 0; YYERROR causes the parser to behave as if the current input symbol had been a syntax error; *yyerror* is called, and error recovery takes place. These mechanisms can be used to simulate parsers with multiple endmarkers or context-sensitive syntax checking.

Accessing Values in Enclosing Rules.

An action may refer to values returned by actions to the left of the current rule. The mechanism is simply the same as with ordinary actions, a dollar sign followed by a digit, but in this case the digit may be 0 or negative. Consider

```
sent    :       adj noun verb adj noun
                        { look at the sentence . . . }

        ;

adj     :       THE             {       $$ = THE; }
        |       YOUNG           {       $$ = YOUNG; }
        . . .
        ;

noun    :       DOG
                        {       $$ = DOG; }
        |       CRONE
                        {       if( $0 = = YOUNG ){
                                        printf( "what?\n" );
                                }
                        $$ = CRONE;
                        }
        ;
        . . .
```

In the action following the word CRONE, a check is made that the preceding token shifted was not YOUNG. Obviously, this is only possible when a great deal is known about what might precede the symbol *noun* in the input. There is also a distinctly unstructured flavor about this. Nevertheless, at times this mechanism will save a great deal of trouble, especially when a few combinations are to be excluded from an otherwise regular structure.

Support for Arbitrary Value Types

By default, the values returned by actions and the lexical analyzer are integers. Yacc can also support values of other types, including structures. In addition, Yacc keeps track of the types, and inserts appropriate union member names so that the resulting parser will be strictly type checked. The Yacc value stack (see Section 4) is declared to be a *union* of the various types of values desired. The user declares the union, and associates union member names to each token and nonterminal symbol having a value. When the value is referenced through a $$ or $n construction, Yacc will automatically insert the appropriate union name, so that no unwanted conversions will take place. In addition, type checking commands such as *Lint*[5] will be far more silent.

There are three mechanisms used to provide for this typing. First, there is a way of defining the union; this must be done by the user since other programs, notably the lexical analyzer, must know about the union member names. Second, there is a way of associating a union member name with tokens and nonterminals. Finally, there is a mechanism for describing the type of those few values where Yacc can not easily determine the type.

To declare the union, the user includes in the declaration section:

```
%union {
        body of union ...
        }
```

This declares the Yacc value stack, and the external variables *yylval* and *yyval*, to have type equal to this union. If Yacc was invoked with the **—d** option, the union declaration is copied onto the *y.tab.h* file. Alternatively, the union may be declared in a header file, and a typedef used to define the variable YYSTYPE to represent this union. Thus, the header file might also have said:

```
typedef union {
        body of union ...
        } YYSTYPE;
```

The header file must be included in the declarations section, by use of %{ and %}.

Once YYSTYPE is defined, the union member names must be associated with the various terminal and nonterminal names. The construction

```
< name >
```

is used to indicate a union member name. If this follows one of the keywords %token, %left, %right, and %nonassoc, the union member name is associated with the tokens listed. Thus, saying

```
%left  <optype>  '+' '—'
```

will cause any reference to values returned by these two tokens to be tagged with the union member name *optype*. Another keyword, %type, is used similarly to associate union member names with nonterminals. Thus, one might say

```
%type  <nodetype>  expr  stat
```

There remain a couple of cases where these mechanisms are insufficient. If there is an action within a rule, the value returned by this action has no *a priori* type. Similarly, reference to left context values (such as $0 — see the previous subsection) leaves Yacc with no easy way of knowing the type. In this case, a type can be imposed on the reference by inserting a union member name, between < and >, immediately after the first $. An example of this usage is

```
rule    :       aaa { $<intval>$  =  3; } bbb
                {       fun( $<intval>2, $<other>0 ); }
        ;
```

This syntax has little to recommend it, but the situation arises rarely.

A sample specification is given in Appendix C. The facilities in this subsection are not triggered until they are used: in particular, the use of %type will turn on these mechanisms. When they are used, there is a fairly strict level of checking. For example, use of $n or $$ to refer to something with no defined type is diagnosed. If these facilities are not triggered, the Yacc value stack is used to hold *int*'s, as was true historically.

11: Acknowledgements

Yacc owes much to a most stimulating collection of users, who have goaded me beyond my inclination, and frequently beyond my ability, in their endless search for "one more feature". Their irritating unwillingness to learn how to do things my way has usually led to my doing things their way; most of the time, they have been right. B. W. Kernighan, P. J. Plauger, S. I. Feldman, C. Imagna, M. E. Lesk, and A. Snyder will recognize some of their ideas in the current version of Yacc. C. B. Haley contributed to the error recovery algorithm. D. M. Ritchie, B. W. Kernighan, and M. O. Harris helped translate this document into English. Al Aho also deserves special credit for bringing the mountain to Mohammed, and other favors.

References

1. B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice-Hall, Englewood Cliffs, New Jersey (1978).

2. A. V. Aho and S. C. Johnson, "LR Parsing," *Comp. Surveys* **6**(2) pp. 99-124 (June 1974).

3. A. V. Aho, S. C. Johnson, and J. D. Ullman, "Deterministic Parsing of Ambiguous Grammars," *Comm. Assoc. Comp. Mach.* **18**(8) pp. 441-452 (August 1975).

4. A. V. Aho and J. D. Ullman, *Principles of Compiler Design,* Addison-Wesley, Reading, Mass. (1977).

5. S. C. Johnson, "Lint, a C Program Checker," Comp. Sci. Tech. Rep. No. 65 (December 1977).

6. S. C. Johnson, "A Portable Compiler: Theory and Practice," *Proc. 5th ACM Symp. on Principles of Programming Languages,* (January 1978).

7. B. W. Kernighan and L. L. Cherry, "A System for Typesetting Mathematics," *Comm. Assoc. Comp. Mach.* **18** pp. 151-157 (March 1975).

8. M. E. Lesk, "Lex — A Lexical Analyzer Generator," Comp. Sci. Tech. Rep. No. 39, Bell Laboratories, Murray Hill, New Jersey (October 1975).

Appendix A: A Simple Example

This example gives the complete Yacc specification for a small desk calculator; the desk calculator has 26 registers, labeled "a" through "z", and accepts arithmetic expressions made up of the operators +, −, *, /, % (mod operator), & (bitwise and), | (bitwise or), and assignment. If an expression at the top level is an assignment, the value is not printed; otherwise it is. As in C, an integer that begins with 0 (zero) is assumed to be octal; otherwise, it is assumed to be decimal.

As an example of a Yacc specification, the desk calculator does a reasonable job of showing how precedences and ambiguities are used, and demonstrating simple error recovery. The major oversimplifications are that the lexical analysis phase is much simpler than for most applications, and the output is produced immediately, line by line. Note the way that decimal and octal integers are read in by the grammar rules; This job is probably better done by the lexical analyzer.

```
%{
# include <stdio.h>
# include <ctype.h>

int  regs[26];
int  base;

%}

%start  list

%token  DIGIT  LETTER

%left  '|'
%left  '&'
%left  '+'  '−'
%left  '*'  '/'  '%'
%left  UMINUS      /* supplies  precedence  for  unary  minus */

%%      /* beginning  of  rules  section */

list    :       /* empty */
        |       list  stat  '\n'
        |       list  error  '\n'
                        {       yyerrok; }
        ;

stat    :       expr
                        {       printf( "%d\n", $1 ); }
        |       LETTER  '='  expr
                        {       regs[$1]  =  $3; }
        ;

expr    :       '(' expr ')'
                        {       $$  =  $2; }
        |       expr  '+'  expr
                        {       $$  =  $1  +  $3; }
        |       expr  '−'  expr
                        {       $$  =  $1  −  $3; }
```

```
    |       expr '*' expr
                    {       $$ = $1 * $3; }
    |       expr '/' expr
                    {       $$ = $1 / $3; }
    |       expr '%' expr
                    {       $$ = $1 % $3; }
    |       expr '&' expr
                    {       $$ = $1 & $3; }
    |       expr 'l' expr
                    {       $$ = $1 | $3; }
    |       '—' expr        %prec UMINUS
                    {       $$ = — $2; }
    |       LETTER
                    {       $$ = regs[$1]; }
    |       number
    ;

number:     DIGIT
                    {       $$ = $1;   base = ($1==0) ? 8 : 10; }
    |       number DIGIT
                    {       $$ = base * $1 + $2; }
    ;

%%     /* start of programs */

yylex() {                   /* lexical analysis routine */
        /* returns LETTER for a lower case letter, yylval = 0 through 25 */
        /* return DIGIT for a digit, yylval = 0 through 9 */
        /* all other characters are returned immediately */

        int c;

        while( (c=getchar()) == '' ) {/* skip blanks */ }

        /* c is now nonblank */

        if( islower( c ) ) {
                yylval = c — 'a';
                return ( LETTER );
                }
        if( isdigit( c ) ) {
                yylval = c — '0';
                return( DIGIT );
                }
        return( c );
        }
```

Appendix B: Yacc Input Syntax

This Appendix has a description of the Yacc input syntax, as a Yacc specification. Context dependencies, etc., are not considered. Ironically, the Yacc input specification language is most naturally specified as an LR(2) grammar; the sticky part comes when an identifier is seen in a rule, immediately following an action. If this identifier is followed by a colon, it is the start of the next rule; otherwise it is a continuation of the current rule, which just happens to have an action embedded in it. As implemented, the lexical analyzer looks ahead after seeing an identifier, and decide whether the next token (skipping blanks, newlines, comments, etc.) is a colon. If so, it returns the token C_IDENTIFIER. Otherwise, it returns IDENTIFIER. Literals (quoted strings) are also returned as IDENTIFIERS, but never as part of C_IDENTIFIERs.

```
              /* grammar for the input to Yacc */

              /* basic entities */
%token  IDENTIFIER       /*  includes identifiers  and literals */
%token  C_IDENTIFIER  /*  identifier (but not literal) followed by colon   */
%token  NUMBER            /*   [0-9]+   */

              /* reserved words:  %type  => TYPE, %left  => LEFT, etc. */

%token  LEFT RIGHT NONASSOC TOKEN PREC TYPE START UNION

%token  MARK  /* the %% mark */
%token  LCURL  /* the %{ mark */
%token  RCURL  /* the %} mark */

              /* ascii character literals stand for themselves */

%start  spec

%%

spec      :         defs MARK rules tail
          ;

tail      :         MARK {  In this action, eat up the rest of the file  }
          |         /* empty: the second MARK is optional */
          ;

defs      :         /* empty */
          |         defs def
          ;

def       :         START IDENTIFIER
          |         UNION { Copy union definition to output }
          |         LCURL { Copy C code to output file } RCURL
          |         ndefs rword tag nlist
          ;

rword     :         TOKEN
          |         LEFT
          |         RIGHT
```

```
              |       NONASSOC
              |       TYPE
              ;


tag           :       /* empty: union tag is optional */
              |       '<' IDENTIFIER '>'
              ;


nlist         :       nmno
              |       nlist nmno
              |       nlist ',' nmno
              ;


nmno          :       IDENTIFIER              /* NOTE: literal illegal with %type */
              |       IDENTIFIER NUMBER       /* NOTE: illegal with %type */
              ;


      /* rules section */

rules         :       C_IDENTIFIER rbody prec
              |       rules rule
              ;


rule          :       C_IDENTIFIER rbody prec
              |       '|' rbody prec
              ;


rbody         :       /* empty */
              |       rbody IDENTIFIER
              |       rbody act
              ;


act           :       '{' { Copy action, translate $$, etc. } '}'
              ;


prec          :       /* empty */
              |       PREC IDENTIFIER
              |       PREC IDENTIFIER act
              |       prec ';'
              ;
```

Appendix C: An Advanced Example

This Appendix gives an example of a grammar using some of the advanced features discussed in Section 10. The desk calculator example in Appendix A is modified to provide a desk calculator that does floating point interval arithmetic. The calculator understands floating point constants, the arithmetic operations +, −, *, /, unary −, and = (assignment), and has 26 floating point variables, "a" through "z". Moreover, it also understands *intervals*, written

$$(x , y)$$

where x is less than or equal to y. There are 26 interval valued variables "A" through "Z" that may also be used. The usage is similar to that in Appendix A; assignments return no value, and print nothing, while expressions print the (floating or interval) value.

This example explores a number of interesting features of Yacc and C. Intervals are represented by a structure, consisting of the left and right endpoint values, stored as *double*'s. This structure is given a type name, INTERVAL, by using *typedef*. The Yacc value stack can also contain floating point scalars, and integers (used to index into the arrays holding the variable values). Notice that this entire strategy depends strongly on being able to assign structures and unions in C. In fact, many of the actions call functions that return structures as well.

It is also worth noting the use of YYERROR to handle error conditions: division by an interval containing 0, and an interval presented in the wrong order. In effect, the error recovery mechanism of Yacc is used to throw away the rest of the offending line.

In addition to the mixing of types on the value stack, this grammar also demonstrates an interesting use of syntax to keep track of the type (e.g. scalar or interval) of intermediate expressions. Note that a scalar can be automatically promoted to an interval if the context demands an interval value. This causes a large number of conflicts when the grammar is run through Yacc: 18 Shift/Reduce and 26 Reduce/Reduce. The problem can be seen by looking at the two input lines:

$$2.5 + (3.5 − 4.)$$

and

$$2.5 + (3.5 , 4.)$$

Notice that the 2.5 is to be used in an interval valued expression in the second example, but this fact is not known until the "," is read; by this time, 2.5 is finished, and the parser cannot go back and change its mind. More generally, it might be necessary to look ahead an arbitrary number of tokens to decide whether to convert a scalar to an interval. This problem is evaded by having two rules for each binary interval valued operator: one when the left operand is a scalar, and one when the left operand is an interval. In the second case, the right operand must be an interval, so the conversion will be applied automatically. Despite this evasion, there are still many cases where the conversion may be applied or not, leading to the above conflicts. They are resolved by listing the rules that yield scalars first in the specification file; in this way, the conflicts will be resolved in the direction of keeping scalar valued expressions scalar valued until they are forced to become intervals.

This way of handling multiple types is very instructive, but not very general. If there were many kinds of expression types, instead of just two, the number of rules needed would increase dramatically, and the conflicts even more dramatically. Thus, while this example is instructive, it is better practice in a more normal programming language environment to keep the type information as part of the value, and not as part of the grammar.

Finally, a word about the lexical analysis. The only unusual feature is the treatment of floating point constants. The C library routine *atof* is used to do the actual conversion from a character string to a double precision value. If the lexical analyzer detects an error, it responds by returning a token that is illegal in the grammar, provoking a syntax error in the parser, and thence error recovery.

```
%{

# include  <stdio.h>
# include  <ctype.h>

typedef  struct  interval  {
        double  lo,  hi;
        }  INTERVAL;

INTERVAL  vmul(),  vdiv();

double  atof();

double  dreg[ 26 ];
INTERVAL  vreg[ 26 ];

%}

%start    lines

%union   {
        int  ival;
        double  dval;
        INTERVAL  vval;
        }

%token  <ival>  DREG VREG        /* indices into dreg, vreg arrays */

%token  <dval>  CONST            /* floating point constant */

%type  <dval>  dexp         /* expression */

%type  <vval>  vexp         /* interval expression */

        /* precedence information about the operators */

%left    '+' '−'
%left    '*' '/'
%left    UMINUS       /* precedence for unary minus */

%%

lines   :       /* empty */
        |       lines line
        ;

line    :       dexp '\n'
                {       printf( "%15.8f\n", $1 ); }
        |       vexp '\n'
                {       printf( "(%15.8f , %15.8f )\n", $1.lo, $1.hi ); }
        |       DREG '=' dexp '\n'
                {       dreg[$1] = $3; }
        |       VREG '=' vexp '\n'
```

```
                           {        vreg[$1]  =  $3;  }
           |     error  '\n'
                           {        yyerrok;  }

           ;


dexp    :      CONST
        |      DREG
                           {        $$  =  dreg[$1];  }
        |      dexp  '+'  dexp
                           {        $$  =  $1  +  $3;  }
        |      dexp  '-'  dexp
                           {        $$  =  $1  -  $3;  }
        |      dexp  '*'  dexp
                           {        $$  =  $1  *  $3;  }
        |      dexp  '/'  dexp
                           {        $$  =  $1  /  $3;  }
        |      '-'  dexp        %prec  UMINUS
                           {        $$  =  - $2;  }
        |      '('  dexp  ')'
                           {        $$  =  $2;  }

        ;


vexp    :      dexp
                           {        $$.hi  =  $$.lo  ==  $1;  }
        |      '('  dexp  ','  dexp  ')'
                           {
                           $$.lo  =  $2;
                           $$.hi  =  $4;
                           if( $$.lo  >  $$.hi ){
                                   printf( "interval  out  of  order\n" );
                                   YYERROR;
                                   }
                           }
        |      VREG
                           {        $$  =  vreg[$1];    }
        |      vexp  '+'  vexp
                           {        $$.hi  =  $1.hi  +  $3.hi;
                                    $$.lo  =  $1.lo  +  $3.lo;    }
        |      dexp  '+'  vexp
                           {        $$.hi  =  $1  +  $3.hi;
                                    $$.lo  =  $1  +  $3.lo;    }
        |      vexp  '-'  vexp
                           {        $$.hi  =  $1.hi  -  $3.lo;
                                    $$.lo  =  $1.lo  -  $3.hi;    }
        |      dexp  '-'  vexp
                           {        $$.hi  =  $1  -  $3.lo;
                                    $$.lo  =  $1  -  $3.hi;    }
        |      vexp  '*'  vexp
                           {        $$  =  vmul( $1.lo,  $1.hi,  $3 );  }
        |      dexp  '*'  vexp
                           {        $$  =  vmul( $1,  $1,  $3 );  }
        |      vexp  '/'  vexp
                           {        if( dcheck( $3 ) )  YYERROR;
                                    $$  =  vdiv( $1.lo,  $1.hi,  $3 );  }
```

```
    |       dexp '/' vexp
                    {       if( dcheck( $3 ) ) YYERROR;
                            $$ = vdiv( $1, $1, $3 ); }
    |       '-' vexp        %prec UMINUS
                    {       $$.hi = -$2.lo;   $$.lo = -$2.hi;   }
    |       '(' vexp ')'
                    {       $$ = $2; }

    ;


%%

# define BSZ 50       /* buffer size for floating point numbers */

        /* lexical analysis */

yylex(){
        register c;

        while( (c=getchar()) == ' ' ){ /* skip over blanks */ }

        if( isupper( c ) ){
                yylval.ival = c - 'A';
                return( VREG );
                }
        if( islower( c ) ){
                yylval.ival = c - 'a';
                return( DREG );
                }

        if( isdigit( c ) || c=='.' ){
                /* gobble up digits, points, exponents */

                char buf[BSZ+1], *cp = buf;
                int dot = 0, exp = 0;

                for( ; (cp-buf)<BSZ ; ++cp,c=getchar() ){

                        *cp = c;
                        if( isdigit( c ) ) continue;
                        if( c == '.' ){
                                if( dot++ || exp ) return( '.' );   /* will cause syntax error */
                                continue;
                                }

                        if( c == 'e' ){
                                if( exp++ ) return( 'e' );   /* will cause syntax error */
                                continue;
                                }

                        /* end of number */
                        break;
                        }
                *cp = '\0';
                if( (cp-buf) >= BSZ ) printf( "constant too long: truncated\n" );
```

```
                else ungetc( c, stdin );   /* push back last char read */
                yylval.dval = atof( buf );
                return( CONST );
                }
        return( c );
        }


INTERVAL hilo( a, b, c, d ) double a, b, c, d; {
        /* returns the smallest interval containing a, b, c, and d */
        /* used by *, / routines */
        INTERVAL v;

        if( a>b ) { v.hi = a;   v.lo = b; }
        else { v.hi = b;   v.lo = a; }

        if( c>d ) {
                if( c>v.hi ) v.hi = c;
                if( d<v.lo ) v.lo = d;
                }
        else {
                if( d>v.hi ) v.hi = d;
                if( c<v.lo ) v.lo = c;
                }
        return( v );
        }


INTERVAL vmul( a, b, v ) double a, b;   INTERVAL v; {
        return( hilo( a*v.hi, a*v.lo, b*v.hi, b*v.lo ) );
        }


dcheck( v ) INTERVAL v; {
        if( v.hi >= 0. && v.lo <= 0. ){
                printf( "divisor interval contains 0.\n" );
                return( 1 );
                }
        return( 0 );
        }


INTERVAL vdiv( a, b, v ) double a, b;   INTERVAL v; {
        return( hilo( a/v.hi, a/v.lo, b/v.hi, b/v.lo ) );
        }
```

Appendix D: Old Features Supported but not Encouraged

This Appendix mentions synonyms and features which are supported for historical continuity, but, for various reasons, are not encouraged.

1. Literals may also be delimited by double quotes ''"''.

2. Literals may be more than one character long. If all the characters are alphabetic, numeric, or _, the type number of the literal is defined, just as if the literal did not have the quotes around it. Otherwise, it is difficult to find the value for such literals.

    The use of multi-character literals is likely to mislead those unfamiliar with Yacc, since it suggests that Yacc is doing a job which must be actually done by the lexical analyzer.

3. Most places where % is legal, backslash ''\'' may be used. In particular, \\ is the same as %%, \left the same as %left, etc.

4. There are a number of other synonyms:

    %< is the same as %left
    %> is the same as %right
    %binary and %2 are the same as %nonassoc
    %0 and %term are the same as %token
    %= is the same as %prec

5. Actions may also have the form

    ={ ... }

    and the curly braces can be dropped if the action is a single C statement.

6. C code between %{ and %} used to be permitted at the head of the rules section, as well as in the declaration section.

Section 10

LEX—A LEXICAL ANALYZER GENERATOR

INTRODUCTION

lex, a lexical analyzer generator, was developed at Bell Laboratories and is licensed by Western Electric for use on the 8560. The remainder of this section is a reprint of an article describing *lex*. The Technical Notes section of this manual describes the limitations of this program and any changes made to this program by Tektronix.

Lex - A Lexical Analyzer Generator

M. E. Lesk and E. Schmidt

Bell Laboratories

Murray Hill, New Jersey 07974

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.

Lex source is a table of regular expressions and corresponding program fragments. The table is translated to a program which reads an input stream, copying it to an output stream and partitioning the input into strings which match the given expressions. As each such string is recognized the corresponding program fragment is executed. The recognition of the expressions is performed by a deterministic finite automaton generated by Lex. The program fragments written by the user are executed in the order in which the corresponding regular expressions occur in the input stream.

The lexical analysis programs written with Lex accept ambiguous specifications and choose the longest match possible at each input point. If necessary, substantial lookahead is performed on the input, but the input stream will be backed up to the end of the current partition, so that the user has general freedom to manipulate it.

Lex can be used to generate analyzers in either C or Ratfor, a language which can be translated automatically to portable Fortran. It is available on the PDP-11 UNIX, Honeywell GCOS, and IBM OS systems. Lex is designed to simplify interfacing with Yacc, for those with access to this compiler-compiler system.

Table of Contents

1 Introduction.

Lex is a program generator designed for lexical processing of character input streams. It accepts a high-level, problem oriented specification for character string matching, and produces a program in a general purpose language which recognizes regular expressions. The regular expressions are specified by the user in the source specifications given to Lex. The Lex written code recognizes these expressions in an input stream and partitions the input stream into strings matching the expressions. At the boundaries between strings program sections provided by the user are executed. The Lex source file associates the regular expressions and the program fragments. As each expression appears in the input to the program written by Lex, the corresponding fragment is executed.

The user supplies the additional code beyond expression matching needed to complete his tasks, possibly including code written by other generators. The program that recognizes the expressions is generated in the general purpose programming language employed for the user's program fragments. Thus, a high level expression language is provided to write the string expressions to be matched while the user's freedom to write actions is unimpaired. This avoids forcing the user who wishes to use a string manipulation language for input analysis to

Source → [Lex] → yylex

Input → [yylex] → Output
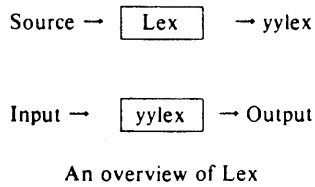
An overview of Lex

Figure 1

write processing programs in the same and often inappropriate string handling language.

Lex is not a complete language, but rather a generator representing a new language feature which can be added to different programming languages, called "host languages." Just as general purpose languages can produce code to run on different computer hardware, Lex can write code in different host languages. The host language is used for the output code generated by Lex and also for the program fragments added by the user. Compatible run-time libraries for the different host languages are also provided. This makes Lex adaptable to different environments and different users. Each application may be directed to the combination of hardware and host language appropriate to the task, the user's background, and the properties of local implementations. At present there are only two host languages, C[1] and Fortran (in the form of the Ratfor language[2]). Lex itself exists on UNIX, GCOS, and OS/370; but the code generated by Lex may be taken anywhere the appropriate compilers exist.

Lex turns the user's expressions and actions (called *source* in this memo) into the host general-purpose language; the generated program is named *yylex*. The *yylex* program will recognize expressions in a stream (called *input* in this memo) and perform the specified actions for each expression as it is detected. See Figure 1.

For a trivial example, consider a program to delete from the input all blanks or tabs at the ends of lines.

```
%%
[ \t]+$   ;
```

is all that is required. The program contains a %% delimiter to mark the beginning of the rules, and one rule.

This rule contains a regular expression which matches one or more instances of the characters blank or tab (written \t for visibility, in accordance with the C language convention) just prior to the end of a line. The brackets indicate the character class made of blank and tab; the + indicates "one or more ..."; and the $ indicates "end of line," as in QED. No action is specified, so the program generated by Lex (yylex) will ignore these characters. Everything else will be copied. To change any remaining string of blanks or tabs to a single blank, add another rule:

```
%%
[ \t]+$    ;
[ \t]+     printf(" ");
```

The finite automaton generated for this source will scan for both rules at once, observing at the termination of the string of blanks or tabs whether or not there is a newline character, and executing the desired rule action. The first rule matches all strings of blanks or tabs at the end of lines, and the second rule all remaining strings of blanks or tabs.

Lex can be used alone for simple transformations, or for analysis and statistics gathering on a lexical level. Lex can also be used with a parser generator to perform the lexical analysis phase; it is particularly easy to interface Lex and Yacc [3]. Lex programs recognize only regular expressions; Yacc writes parsers that accept a large class of context free grammars, but require a lower level analyzer to recognize input tokens. Thus, a combination of Lex and Yacc is often appropriate. When used as a preprocessor for a later parser generator, Lex is used to partition the input stream, and the parser generator assigns structure to the resulting pieces. The flow of control in such a case (which might be the first half of a compiler, for example) is shown in Figure 2. Additional programs, written by other generators or by hand, can be added easily to programs written by Lex. Yacc users will realize that the name *yylex* is what Yacc expects its lexical analyzer to be named, so that the use of this name by Lex simplifies interfacing.

Lex generates a deterministic finite automaton from the regular expressions in the source [4]. The automaton is interpreted, rather than compiled, in order to save space. The result is still a fast analyzer. In particular, the time

lexical   grammar
rules   rules
  ↓   ↓
[Lex]   [Yacc]
  ↓   ↓
Input → [yylex] → [yyparse] → Parsed input
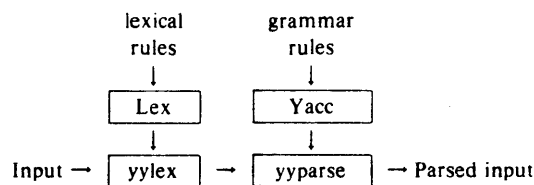
Lex with Yacc

Figure 2

10-3

taken by a Lex program to recognize and partition an input stream is proportional to the length of the input. The number of Lex rules or the complexity of the rules is not important in determining speed, unless rules which include forward context require a significant amount of rescanning. What does increase with the number and complexity of rules is the size of the finite automaton, and therefore the size of the program generated by Lex.

In the program written by Lex, the user's fragments (representing the *actions* to be performed as each regular expression is found) are gathered as cases of a switch (in C) or branches of a computed GOTO (in Ratfor). The automaton interpreter directs the control flow. Opportunity is provided for the user to insert either declarations or additional statements in the routine containing the actions, or to add subroutines outside this action routine.

Lex is not limited to source which can be interpreted on the basis of one character lookahead. For example, if there are two rules, one looking for *ab* and another for *abcdefg*, and the input stream is *abcdefh*, Lex will recognize *ab* and leave the input pointer just before *cd. . .* Such backup is more costly than the processing of simpler languages.

2 Lex Source.

The general format of Lex source is:

```
{definitions}
%%
{rules}
%%
{user subroutines}
```

where the definitions and the user subroutines are often omitted. The second %% is optional, but the first is required to mark the beginning of the rules. The absolute minimum Lex program is thus

```
%%
```

(no definitions, no rules) which translates into a program which copies the input to the output unchanged.

In the outline of Lex programs shown above, the *rules* represent the user's control decisions; they are a table, in which the left column contains *regular expressions* (see section 3) and the right column contains *actions,* program fragments to be executed when the expressions are recognized. Thus an individual rule might appear

```
integer    printf("found keyword INT");
```

to look for the string *integer* in the input stream and print the message "found keyword INT" whenever it appears. In this example the host procedural language is C and the C library function *printf* is used to print the string. The end of the expression is indicated by the first blank or tab character. If the action is merely a single C expression, it can just be given on the right side of the line; if it is compound, or takes more than a line, it should be enclosed in

braces. As a slightly more useful example, suppose it is desired to change a number of words from British to American spelling. Lex rules such as

```
colour       printf("color");
mechanise    printf("mechanize");
petrol       printf("gas");
```

would be a start. These rules are not quite enough, since the word *petroleum* would become *gaseum*, a way of dealing with this will be described later.

3 Lex Regular Expressions.

The definitions of regular expressions are very similar to those in QED [5]. A regular expression specifies a set of strings to be matched. It contains text characters (which match the corresponding characters in the strings being compared) and operator characters (which specify repetitions, choices, and other features). The letters of the alphabet and the digits are always text characters; thus the regular expression

integer

matches the string *integer* wherever it appears and the expression

a57D

looks for the string *a57D*.

Operators. The operator characters are

" \ [ ] ^ - ? . * + | ( ) $ / { } % < >

and if they are to be used as text characters, an escape should be used. The quotation mark operator (") indicates that whatever is contained between a pair of quotes is to be taken as text characters. Thus

xyz"++"

matches the string *xyz++* when it appears. Note that a part of a string may be quoted. It is harmless but unnecessary to quote an ordinary text character; the expression

"xyz++"

is the same as the one above. Thus by quoting every non-alphanumeric character being used as a text character, the user can avoid remembering the list above of current operator characters, and is safe should further extensions to Lex lengthen the list.

An operator character may also be turned into a text character by preceding it with \ as in

xyz\+\+

which is another, less readable, equivalent of the above

expressions. Another use of the quoting mechanism is to get a blank into an expression; normally, as explained above, blanks or tabs end a rule. Any blank character not contained within [] (see below) must be quoted. Several normal C escapes with \ are recognized: \n is newline, \t is tab, and \b is backspace. To enter \ itself, use \\. Since newline is illegal in an expression, \n must be used; it is not required to escape tab and backspace. Every character but blank, tab, newline and the list above is always a text character.

Character classes. Classes of characters can be specified using the operator pair []. The construction *[ab]* matches a single character, which may be *a*, *b*, or *c*. Within square brackets, most operator meanings are ignored. Only three characters are special: these are \ — and ^. The — character indicates ranges. For example,

$$[a-z0-9<>\_]$$

indicates the character class containing all the lower case letters, the digits, the angle brackets, and underline. Ranges may be given in either order. Using — between any pair of characters which are not both upper case letters, both lower case letters, or both digits is implementation dependent and will get a warning message. (E.g., [0-z] in ASCII is many more characters than it is in EBCDIC). If it is desired to include the character — in a character class, it should be first or last; thus

$$[-+0-9]$$

matches all the digits and the two signs.

In character classes, the ^ operator must appear as the first character after the left bracket; it indicates that the resulting string is to be complemented with respect to the computer character set. Thus

$$[\hat{}abc]$$

matches all characters except a, b, or c, including all special or control characters; or

$$[\hat{}a-zA-Z]$$

is any character which is not a letter. The \ character provides the usual escapes within character class brackets.

Arbitrary character. To match almost any character, the operator character

is the class of all characters except newline. Escaping into octal is possible although non-portable:

$$[\40-\176]$$

matches all printable characters in the ASCII character set, from octal 40 (blank) to octal 176 (tilde).

Optional expressions. The operator ? indicates an optional element of an expression. Thus

$$ab?c$$

matches either *ac* or *abc*.

Repeated expressions. Repetitions of classes are indicated by the operators * and +.

$$a*$$

is any number of consecutive *a* characters, including zero; while

$$a+$$

is one or more instances of *a*. For example,

$$[a-z]+$$

is all strings of lower case letters. And

$$[A-Za-z][A-Za-z0-9]*$$

indicates all alphanumeric strings with a leading alphabetic character. This is a typical expression for recognizing identifiers in computer languages.

Alternation and Grouping. The operator | indicates alternation:

$$(ab|cd)$$

matches either *ab* or *cd*. Note that parentheses are used for grouping, although they are not necessary on the outside level;

$$ab|cd$$

would have sufficed. Parentheses can be used for more complex expressions:

$$(ab|cd+)?(ef)*$$

matches such strings as *abefef, efefef, cdef,* or *cddd*; but not *abc, abcd,* or *abcdef.*

Context sensitivity. Lex will recognize a small amount of surrounding context. The two simplest operators for this are ^ and $. If the first character of an expression is ^, the expression will only be matched at the beginning of a line (after a newline character, or at the beginning of the input stream). This can never conflict with the other meaning of ^, complementation of character classes, since that only applies within the [] operators. If the very last character is $, the expression will only be matched at the end of a line (when immediately followed by newline). The latter operator is a special case of the / operator character, which indicates trailing context. The expression

$$ab/cd$$

matches the string *ab*, but only if followed by *cd*. Thus

ab$

is the same as

ab/\n

Left context is handled in Lex by *start conditions* as explained in section 10. If a rule is only to be executed when the Lex automaton interpreter is in start condition *x*, the rule should be prefixed by

<x>

using the angle bracket operator characters. If we considered "being at the beginning of a line" to be start condition *ONE*, then the ^ operator would be equivalent to

<ONE>

Start conditions are explained more fully later.

Repetitions and Definitions. The operators {} specify either repetitions (if they enclose numbers) or definition expansion (if they enclose a name). For example

{digit}

looks for a predefined string named *digit* and inserts it at that point in the expression. The definitions are given in the first part of the Lex input, before the rules. In contrast,

a{1,5}

looks for 1 to 5 occurrences of *a*.

Finally, initial % is special, being the separator for Lex source segments.

4 Lex Actions.

When an expression written as above is matched, Lex executes the corresponding action. This section describes some features of Lex which aid in writing actions. Note that there is a default action, which consists of copying the input to the output. This is performed on all strings not otherwise matched. Thus the Lex user who wishes to absorb the entire input, without producing any output, must provide rules to match everything. When Lex is being used with Yacc, this is the normal situation. One may consider that actions are what is done instead of copying the input to the output; thus, in general, a rule which merely copies can be omitted. Also, a character combination which is omitted from the rules and which appears as input is likely to be printed on the output, thus calling attention to the gap in the rules.

One of the simplest things that can be done is to ignore the input. Specifying a C null statement, *;* as an action causes this result. A frequent rule is

[\t\n]   ;

which causes the three spacing characters (blank, tab, and newline) to be ignored.

Another easy way to avoid writing actions is the action character |, which indicates that the action for this rule is the action for the next rule. The previous example could also have been written

    " "
    "\t"
    "\n"

with the same result, although in different style. The quotes around \n and \t are not required.

In more complex actions, the user will often want to know the actual text that matched some expression like *[a—z]+*. Lex leaves this text in an external character array named *yytext*. Thus, to print the name found, a rule like

    [a-z]+   printf("%s", yytext);

will print the string in *yytext*. The C function *printf* accepts a format argument and data to be printed; in this case, the format is "print string" (% indicating data conversion, and *s* indicating string type), and the data are the characters in *yytext*. So this just places the matched string on the output. This action is so common that it may be written as ECHO:

    [a-z]+   ECHO;

is the same as the above. Since the default action is just to print the characters found, one might ask why give a rule, like this one, which merely specifies the default action? Such rules are often required to avoid matching some other rule which is not desired. For example, if there is a rule which matches *read* it will normally match the instances of *read* contained in *bread* or *readjust*, to avoid this, a rule of the form *[a—z]+* is needed. This is explained further below.

Sometimes it is more convenient to know the end of what has been found; hence Lex also provides a count *yyleng* of the number of characters matched. To count both the number of words and the number of characters in words in the input, the user might write

    [a-zA-Z]+   {words++; chars += yyleng;}

which accumulates in *chars* the number of characters in the words recognized. The last character in the string matched can be accessed by

yytext[yyleng-1]

in C or

yytext(yyleng)

in Ratfor.

Occasionally, a Lex action may decide that a rule has not recognized the correct span of characters. Two routines are provided to aid with this situation. First, *yymore()* can be called to indicate that the next input expression recognized is to be tacked on to the end of this input. Normally, the next input string would overwrite the current entry in *yytext*. Second, *yyless (n)* may be called to indicate that not all the characters matched by the currently successful expression are wanted right now. The argument *n* indicates the number of characters in *yytext* to be retained. Further characters previously matched are returned to the input. This provides the same sort of lookahead offered by the / operator, but in a different form.

Example: Consider a language which defines a string as a set of characters between quotation (") marks, and provides that to include a " in a string it must be preceded by a \. The regular expression which matches that is somewhat confusing, so that it might be preferable to write

```
\"[^"]*    {
           if (yytext[yyleng-1] == '\\')
               yymore();
           else
               ... normal user processing
           }
```

which will, when faced with a string such as *"abc\"def"* first match the five characters *"abc*; then the call to *yymore()* will cause the next part of the string, *"def*, to be tacked on the end. Note that the final quote terminating the string should be picked up in the code labeled "normal processing".

The function *yyless()* might be used to reprocess text in various circumstances. Consider the C problem of distinguishing the ambiguity of "=−a". Suppose it is desired to treat this as "=− a" but print a message. A rule might be

```
=-[a-zA-Z]    {
              printf("Operator (=−) ambiguous\n");
              yyless(yyleng-1);
              ... action for =− ...
              }
```

which prints a message, returns the letter after the operator to the input stream, and treats the operator as "=−". Alternatively it might be desired to treat this as "= −a". To do this, just return the minus sign as well as the letter to the input:

```
=-[a-zA-Z]    {
              printf("Operator (=−) ambiguous\n");
              yyless(yyleng-2);
              ... action for = ...
              }
```

will perform the other interpretation. Note that the expressions for the two cases might more easily be written

$$=-/[A-Za-z]$$

in the first case and

$$=/-[A-Za-z]$$

in the second; no backup would be required in the rule action. It is not necessary to recognize the whole identifier to observe the ambiguity. The possibility of "=−3", however, makes

$$=-/[^ \t\n]$$

a still better rule.

In addition to these routines, Lex also permits access to the I/O routines it uses. They are:

1) *input()* which returns the next input character;

2) *output(c)* which writes the character *c* on the output; and

3) *unput(c)* pushes the character *c* back onto the input stream to be read later by *input()*.

By default these routines are provided as macro definitions, but the user can override them and supply private versions. There is another important routine in Ratfor, named *lexshf*, which is described below under "Character Set". These routines define the relationship between external files and internal characters, and must all be retained or modified consistently. They may be redefined, to cause input or output to be transmitted to or from strange places, including other programs or internal memory; but the character set used must be consistent in all routines; a value of zero returned by *input* must mean end of file; and the relationship between *unput* and *input* must be retained or the Lex lookahead will not work. Lex does not look ahead at all if it does not have to, but every rule ending in + * ? or $ or containing / implies lookahead. Lookahead is also necessary to match an expression that is a prefix of another expression. See below for a discussion of the character set used by Lex. The standard Lex library imposes a 100 character limit on backup.

Another Lex library routine that the user will sometimes want to redefine is *yywrap()* which is called whenever Lex reaches an end-of-file. If *yywrap* returns a 1, Lex continues with the normal wrapup on end of input. Sometimes, however, it is convenient to arrange for more input to arrive from a new source. In this case, the user should provide a *yywrap* which arranges for new input and returns 0. This instructs Lex to continue processing. The default *yywrap* always returns 1.

This routine is also a convenient place to print tables, summaries, etc. at the end of a program. Note that it is not possible to write a normal rule which recognizes end-of-file; the only access to this condition is through *yywrap*. In fact, unless a private version of *input()* is supplied a file containing nulls cannot be handled, since a value of 0 returned by *input* is taken to be end-of-file.

In Ratfor all of the standard I/O library routines, *input,*

10-7

output, unput, yywrap, and *lexshf,* are defined as integer functions. This requires *input* and *yywrap* to be called with arguments. One dummy argument is supplied and ignored.

5 Ambiguous Source Rules.

Lex can handle ambiguous specifications. When more than one expression can match the current input, Lex chooses as follows:

1) The longest match is preferred.

2) Among rules which matched the same number of characters, the rule given first is preferred.

Thus, suppose the rules

> integer   keyword action ...;
> [a-z]+   identifier action ...;

to be given in that order. If the input is *integers,* it is taken as an identifier, because *[a-z]+* matches 8 characters while *integer* matches only 7. If the input is *integer,* both rules match 7 characters, and the keyword rule is selected because it was given first. Anything shorter (e.g. *int*) will not match the expression *integer* and so the identifier interpretation is used.

The principle of preferring the longest match makes rules containing expressions like .* dangerous. For example,

> '.*'

might seem a good way of recognizing a string in single quotes. But it is an invitation for the program to read far ahead, looking for a distant single quote. Presented with the input

> 'first' quoted string here, 'second' here

the above expression will match

> 'first' quoted string here, 'second'

which is probably not what was wanted. A better rule is of the form

> '[^\n]*'

which, on the above input, will stop after *'first'.* The consequences of errors like this are mitigated by the fact that the . operator will not match newline. Thus expressions like .* stop on the current line. Don't try to defeat this with expressions like [.\n]+ or equivalents; the Lex generated program will try to read the entire input file, causing internal buffer overflows.

Note that Lex is normally partitioning the input stream, not searching for all possible matches of each expression. This means that each character is accounted for once and only once. For example, suppose it is desired to count occurrences of both *she* and *he* in an input text. Some

Lex rules to do this might be

> she   s++;
> he   h++;
> \n |
> .   ;

where the last two rules ignore everything besides *he* and *she.* Remember that . does not include newline. Since *she* includes *he,* Lex will normally *not* recognize the instances of *he* included in *she,* since once it has passed a *she* those characters are gone.

Sometimes the user would like to override this choice. The action REJECT means "go do the next alternative." It causes whatever rule was second choice after the current rule to be executed. The position of the input pointer is adjusted accordingly. Suppose the user really wants to count the included instances of *he:*

> she   {s++; REJECT;}
> he   {h++; REJECT;}
> \n |
> .   ;

these rules are one way of changing the previous example to do just that. After counting each expression, it is rejected; whenever appropriate, the other expression will then be counted. In this example, of course, the user could note that *she* includes *he* but not vice versa, and omit the REJECT action on *he,* in other cases, however, it would not be possible a priori to tell which input characters were in both classes.

Consider the two rules

> a[bc]+   { ... ; REJECT;}
> a[cd]+   { ... ; REJECT;}

If the input is *ab,* only the first rule matches, and on *ad* only the second matches. The input string *accb* matches the first rule for four characters and then the second rule for three characters. In contrast, the input *accd* agrees with the second rule for four characters and then the first rule for three.

In general, REJECT is useful whenever the purpose of Lex is not to partition the input stream but to detect all examples of some items in the input, and the instances of these items may overlap or include each other. Suppose a digram table of the input is desired; normally the digrams overlap, that is the word *the* is considered to contain both *th* and *he.* Assuming a two-dimensional array named *digram* to be incremented, the appropriate source is

```
%%
[a-z][a-z]    {digram[yytext[0]][yytext[1]]++; REJECT;}
\n           ;
```

where the REJECT is necessary to pick up a letter pair beginning at every character, rather than at every other character.

6 Lex Source Definitions.

Remember the format of the Lex source:

```
{definitions}
%%
{rules}
%%
{user routines}
```

So far only the rules have been described. The user needs additional options, though, to define variables for use in his program and for use by Lex. These can go either in the definitions section or in the rules section.

Remember that Lex is turning the rules into a program. Any source not intercepted by Lex is copied into the generated program. There are three classes of such things.

1) Any line which is not part of a Lex rule or action which begins with a blank or tab is copied into the Lex generated program. Such source input prior to the first %% delimiter will be external to any function in the code; if it appears immediately after the first %%, it appears in an appropriate place for declarations in the function written by Lex which contains the actions. This material must look like program fragments, and should precede the first Lex rule.

As a side effect of the above, lines which begin with a blank or tab, and which contain a comment, are passed through to the generated program. This can be used to include comments in either the Lex source or the generated code. The comments should follow the host language convention.

2) Anything included between lines containing only %{ and %} is copied out as above. The delimiters are discarded. This format permits entering text like preprocessor statements that must begin in column 1, or copying lines that do not look like programs.

3) Anything after the third %% delimiter, regardless of formats, etc., is copied out after the Lex output.

Definitions intended for Lex are given before the first %% delimiter. Any line in this section not contained between %{ and %}, and begining in column 1, is assumed to define Lex substitution strings. The format of such lines is

name translation

and it causes the string given as a translation to be associated with the name. The name and translation must be separated by at least one blank or tab, and the name must begin with a letter. The translation can then be called out by the {name} syntax in a rule. Using {D} for the digits and {E} for an exponent field, for example, might abbreviate rules to recognize numbers:

```
D           [0-9]
E           [TEde][-+]?{D}+
%%
{D}+                    printf("integer");
{D}+"."{D}*({E})?    |
{D}*"."{D}+({E})?    |
{D}+{E}
```

Note the first two rules for real numbers; both require a decimal point and contain an optional exponent field, but the first requires at least one digit before the decimal point and the second requires at least one digit after the decimal point. To correctly handle the problem posed by a Fortran expression such as *35.EQ.1*, which does not contain a real number, a context-sensitive rule such as

```
[0-9]+/"."EQ    printf("integer");
```

could be used in addition to the normal rule for integers.

The definitions section may also contain other commands, including the selection of a host language, a character set table, a list of start conditions, or adjustments to the default size of arrays within Lex itself for larger source programs. These possibilities are discussed below under "Summary of Source Format," section 12.

7 Usage.

There are two steps in compiling a Lex source program. First, the Lex source must be turned into a generated program in the host general purpose language. Then this program must be compiled and loaded, usually with a library of Lex subroutines. The generated program is on a file named *lex.yy.c* for a C host language source and *lex.yy.r* for a Ratfor host environment. There are two I/O libraries, one for C defined in terms of the C standard library [6], and the other defined in terms of Ratfor. To indicate that a Lex source file is intended to be used with the Ratfor host language, make the first line of the file *%R*.

The C programs generated by Lex are slightly different on OS/370, because the OS compiler is less powerful than the UNIX or GCOS compilers, and does less at compile time. C programs generated on GCOS and UNIX are the same. The C host language is default, but may be explicitly requested by making the first line of the source file *%C*.

The Ratfor generated by Lex is the same on all systems, but can not be compiled directly on TSO. See below for instructions. The Ratfor I/O library, however, varies slightly because the different Fortrans disagree on the method of indicating end-of-input and the name of the library routine for logical AND. The Ratfor I/O library, dependent on Fortran character I/O, is quite slow. In particular it reads all input lines as 80A1 format; this will truncate any longer line, discarding your data, and pads any shorter line with blanks. The library version of *input* removes the padding (including any trailing blanks from the original input) before processing. Each source

file using a Ratfor host should begin with the "%R" command.

UNIX. The libraries are accessed by the loader flags *-llc* for C and *-llr* for Ratfor; the C name may be abbreviated to *-ll.* So an appropriate set of commands is

C Host	Ratfor Host
lex source	lex source
cc lex.yy.c -ll -lS	rc -2 lex.yy.r -llr

The resulting program is placed on the usual file *a.out* for later execution. To use Lex with Yacc see below. Although the default Lex I/O routines use the C standard library, the Lex automata themselves do not do so; if private versions of *input, output* and *unput* are given, the library can be avoided. Note the "-2" option in the Ratfor compile command; this requests the larger version of the compiler, a useful precaution.

GCOS. The Lex commands on GCOS are stored in the "." library. The appropriate command sequences are:

C Host	Ratfor Host
./lex source	./lex source
./cc lex.yy.c ./lexclib h=	./rc a= lex.yy.r ./lexrlib h=

The resulting program is placed on the usual file *.program* for later execution (as indicated by the "h=" option); it may be copied to a permanent file if desired. Note the "a=" option in the Ratfor compile command; this indicates that the Fortran compiler is to run in ASCII mode.

TSO. Lex is just barely available on TSO. Restrictions imposed by the compilers which must be used with its output make it rather inconvenient. To use the C version, type

    exec 'dot.lex.clist(lex)' 'sourcename'
    exec 'dot.lex.clist(cload)' 'libraryname membername'

The first command analyzes the source file and writes a C program on file *lex.yy.text.* The second command runs this file through the C compiler and links it with the Lex C library (stored on 'hr289.lcl.load') placing the object program in your file *libraryname.LOAD(membername)* as a completely linked load module. The compiling command uses a special version of the C compiler command on TSO which provides an unusually large intermediate assembler file to compensate for the unusual bulk of C-compiled Lex programs on the OS system. Even so, almost any Lex source program is too big to compile, and must be split.

The same Lex command will compile Ratfor Lex programs, leaving a file *lex.yy.rat* instead of *lex.yy.text* in your directory. The Ratfor program must be edited, however, to compensate for peculiarities of IBM Ratfor. A command sequence to do this, and then compile and load, is available. The full commands are:

    exec 'dot.lex.clist(lex)' 'sourcename'

    exec 'dot.lex.clist(rload)' 'libraryname membername'

with the same overall effect as the C language commands. However, the Ratfor commands will run in a 150K byte partition, while the C commands require 250K bytes to operate.

The steps involved in processing the generated Ratfor program are:

a.   Edit the Ratfor program.

1.   Remove all tabs.

2.   Change all lower case letters to upper case letters.

3.   Convert the file to an 80-column card image file.

b.   Process the Ratfor through the Ratfor preprocessor to get Fortran code.

c.   Compile the Fortran.

d.   Load with the libraries 'hr289.lrl.load' and 'sys1.fortlib'.

The final load module will only read input in 80-character fixed length records. *Warning:* Work is in progress on the IBM C compiler, and Lex and its availability on the IBM 370 are subject to change without notice.

8 Lex and Yacc.

If you want to use Lex with Yacc, note that what Lex writes is a program named *yylex(),* the name required by Yacc for its analyzer. Normally, the default main program on the Lex library calls this routine, but if Yacc is loaded, and its main program is used, Yacc will call *yylex().* In this case each Lex rule should end with

    return(token);

where the appropriate token value is returned. An easy way to get access to Yacc's names for tokens is to compile the Lex output file as part of the Yacc output file by placing the line

    # include "lex.yy.c"

in the last section of Yacc input. Supposing the grammar to be named "good" and the lexical rules to be named "better" the UNIX command sequence can just be:

    yacc good
    lex better
    cc y.tab.c -ly -ll -lS

The Yacc library (-ly) should be loaded before the Lex library, to obtain a main program which invokes the Yacc parser. The generations of Lex and Yacc programs can be done in either order.

9 Examples.

As a trivial problem, consider copying an input file while adding 3 to every positive number divisible by 7. Here is a suitable Lex source program

```
%%
          int k;
[0-9] +   {
          scanf(-1, yytext, "%d", &k);
          if (k%7 == 0)
                printf("%d", k+3);
          else
                printf("%d",k);
          }
```

ɔ do just that. The rule [0-9]+ recognizes strings of di-
its; *scanf* converts the digits to binary and stores the
esult in *k*. The operator % (remainder) is used to check
/hether *k* is divisible by 7; if it is, it is incremented by 3
s it is written out. It may be objected that this program
/ill alter such input items as *49.63* or *X7*. Furthermore,
t increments the absolute value of all negative numbers
livisible by 7. To avoid this, just add a few more rules
fter the active one, as here:

```
%%
              int k;
?[0-9] +      {
              scanf(-1, yytext, "%d", &k);
              printf("%d", k%7 == 0 ? k+3 : k);
              }
?[0-9.] +     ECHO;
.A-Za-z][A-Za-z0-9] +     ECHO;
```

Numerical strings containing a "." or preceded by a letter
/ill be picked up by one of the last two rules, and not
:hanged. The *if-else* has been replaced by a C conditional
:xpression to save space; the form *a?b:c* means "if *a*
hen *b* else *c*".

For an example of statistics gathering, here is a pro-
gram which histograms the lengths of words, where a
word is defined as a string of letters.

```
              int lengs[100];
%%
[a-z] +       lengs[yyleng] + +;
.             |
\n            ;
%%
yywrap ()
{
int i;
printf("Length  No. words\n");
for(i=0; i<100; i++)
    if (lengs[i] > 0)
        printf("%5d%10d\n",i,lengs[i]);
return(1);
}
```

This program accumulates the histogram, while producing
no output. At the end of the input it prints the table.
The final statement *return(1);* indicates that Lex is to per-
form wrapup. If *yywrap* returns zero (false) it implies
that further input is available and the program is to con-
tinue reading and processing. To provide a *yywrap* that

never returns true causes an infinite loop.

As a larger example, here are some parts of a program
written by N. L. Schryer to convert double precision For-
tran to single precision Fortran. Because Fortran does
not distinguish upper and lower case letters, this routine
begins by defining a set of classes including both cases of
each letter:

```
a    [aA]
b    [bB]
c    [cC]
...
z    [zZ]
```

An additional class recognizes white space:

```
W    [ \t]*
```

The first rule changes "double precision" to "real", or
"DOUBLE PRECISION" to "REAL".

```
{d}{o}{u}{b}{l}{e}{W}{p}{r}{e}{c}{i}{s}{i}{o}{n} {
    printf(yytext[0] == 'd'? "real" : "REAL");
    }
```

Care is taken throughout this program to preserve the
case (upper or lower) of the original program. The condi-
tional operator is used to select the proper form of the
keyword. The next rule copies continuation card indica-
tions to avoid confusing them with constants:

```
^"    "[^ 0]    ECHO;
```

In the regular expression, the quotes surround the blanks.
It is interpreted as "beginning of line, then five blanks,
then anything but blank or zero." Note the two different
meanings of ^. There follow some rules to change double
precision constants to ordinary floating constants.

```
[0-9] +{W}{d}{W}[+-]?{W}[0-9] +      |
[0-9] +{W}"."{W}{d}{W}[+-]?{W}[0-9] +      |
"."{W}[0-9] +{W}{d}{W}[+-]?{W}[0-9] +      {
    /* convert constants */
    for(p=yytext; *p != 0; p++)
        {
        if (*p == 'd' | *p == 'D')
            *p =+ 'e'- 'd';
        ECHO;
        }
```

After the floating point constant is recognized, it is
scanned by the *for* loop to find the letter *d* or *D*. The
program than adds *'e'-'d'*, which converts it to the next
letter of the alphabet. The modified constant, now
single-precision, is written out again. There follow a
series of names which must be respelled to remove their
initial *d*. By using the array *yytext* the same action
suffices for all the names (only a sample of a rather long
list is given here).

10-11

```
{d}{s}{i}{n}          |
{d}{c}{o}{s}          |
{d}{s}{q}{r}{t}       |
{d}{a}{t}{a}{n}       |
...
{d}{f}{l}{o}{a}{t}    printf("%s",yytext+1);
```

Another list of names must have initial *d* changed to initial *a*:

```
{d}{l}{o}{g}          |
{d}{l}{o}{g}10        |
{d}{m}{i}{n}1         |
{d}{m}{a}{x}1         {
                      yytext[0] =+ 'a' - 'd';
                      ECHO;
                      }
```

And one routine must have initial *d* changed to initial *r*.

```
{d}1{m}{a}{c}{h}      {yytext[0] =+ 'r' - 'd';
```

To avoid such names as *dsinx* being detected as instances of *dsin*, some final rules pick up longer words as identifiers and copy some surviving characters:

```
[A-Za-z][A-Za-z0-9]*  |
[0-9]+                |
\n                    |
                      ECHO;
```

Note that this program is not complete; it does not deal with the spacing problems in Fortran or with the use of keywords as identifiers.

10 Left Context Sensitivity.

Sometimes it is desirable to have several sets of lexical rules to be applied at different times in the input. For example, a compiler preprocessor might distinguish preprocessor statements and analyze them differently from ordinary statements. This requires sensitivity to prior context, and there are several ways of handling such problems. The ^ operator, for example, is a prior context operator, recognizing immediately preceding left context just as $ recognizes immediately following right context. Adjacent left context could be extended, to produce a facility similar to that for adjacent right context, but it is unlikely to be as useful, since often the relevant left context appeared some time earlier, such as at the beginning of a line.

This section describes three means of dealing with different environments: a simple use of flags, when only a few rules change from one environment to another, the use of *start conditions* on rules, and the possibility of making multiple lexical analyzers all run together. In each case, there are rules which recognize the need to change the environment in which the following input text

is analyzed, and set some parameter to reflect the change. This may be a flag explicitly tested by the user's action code; such a flag is the simplest way of dealing with the problem, since Lex is not involved at all. It may be more convenient, however, to have Lex remember the flags as initial conditions on the rules. Any rule may be associated with a start condition. It will only be recognized when Lex is in that start condition. The current start condition may be changed at any time. Finally, if the sets of rules for the different environments are very dissimilar, clarity may be best achieved by writing several distinct lexical analyzers, and switching from one to another as desired.

Consider the following problem: copy the input to the output, changing the word *magic* to *first* on every line which began with the letter *a*, changing *magic* to *second* on every line which began with the letter *b*, and changing *magic* to *third* on every line which began with the letter *c*. All other words and all other lines are left unchanged.

These rules are so simple that the easiest way to do this job is with a flag:

```
           int flag;
%%
^a         {flag = 'a'; ECHO;}
^b         {flag = 'b'; ECHO;}
^c         {flag = 'c'; ECHO;}
\n         {flag =  0 ; ECHO;}
magic      {
           switch (flag)
           {
           case 'a': printf("first"); break;
           case 'b': printf("second"); break;
           case 'c': printf("third"); break;
           default: ECHO; break;
           }
           }
```

should be adequate.

To handle the same problem with start conditions, each start condition must be introduced to Lex in the definitions section with a line reading

```
%Start    name1 name2 ...
```

where the conditions may be named in any order. The word *Start* may be abbreviated to *s* or *S*. The conditions may be referenced at the head of a rule with the < > brackets:

```
<name1>expression
```

is a rule which is only recognized when Lex is in the start condition *name1*. To enter a start condition, execute the action statement

```
BEGIN name1;
```

which changes the start condition to *name1*. To resume the normal state,

BEGIN 0;

resets the initial condition of the Lex automaton interpreter. A rule may be active in several start conditions:

<name1,name2,name3>

is a legal prefix. Any rule not beginning with the <> prefix operator is always active.

The same example as before can be written:

```
%START AA BB CC
%%
^a                      {ECHO; BEGIN AA;}
^b                      {ECHO; BEGIN BB;}
^c                      {ECHO; BEGIN CC;}
\n                      {ECHO; BEGIN 0;}
<AA>magic               printf("first");
<BB>magic               printf("second");
<CC>magic               printf("third");
```

where the logic is exactly the same as in the previous method of handling the problem, but Lex does the work rather than the user's code.

11 Character Set.

The programs generated by Lex handle character I/O only through the routines *input, output,* and *unput.* Thus the character representation provided in these routines is accepted by Lex and employed to return values in *yytext.* For internal use a character is represented as a small integer which, if the standard library is used, has a value equal to the integer value of the bit pattern representing the character on the host computer. In C, the I/O routines are assumed to deal directly in this representation. In Ratfor, it is anticipated that many users will prefer left-adjusted rather than right-adjusted characters; thus the routine *lexshf* is called to change the representation delivered by *input* into a right-adjusted integer. If the user changes the I/O library, the routine *lexshf* should also be changed to a compatible version. The Ratfor library I/O system is arranged to represent the letter *a* as in the Fortran value *1Ha* while in C the letter *a* is represented as the character constant *'a'*. If this interpretation is changed, by providing I/O routines which translate the characters, Lex must be told about it, by giving a translation table. This table must be in the definitions section, and must be bracketed by lines containing only "%T". The table contains lines of the form

{integer} {character string}

which indicate the value associated with each character. Thus the next example maps the lower and upper case letters together into the integers 1 through 26, newline into 27, + and - into 28 and 29, and the digits into 30 through 39. Note the escape for newline. If a table is supplied, every character that is to appear either in the

```
%T
1       Aa
2       Bb
...
26      Zz
27      \n
28      +
29      -
30      0
31      1
...
39      9
%T
```

Sample character table.

rules or in any valid input must be included in the table. No character may be assigned the number 0, and no character may be assigned a bigger number than the size of the hardware character set.

It is not likely that C users will wish to use the character table feature; but for Fortran portability it may be essential.

Although the contents of the Lex Ratfor library routines for input and output run almost unmodified on UNIX, GCOS, and OS/370, they are not really machine independent, and would not work with CDC or Burroughs Fortran compilers. The user is of course welcome to replace *input, output, unput* and *lexshf* but to replace them by completely portable Fortran routines is likely to cause a substantial decrease in the speed of Lex Ratfor programs. A simple way to produce portable routines would be to leave *input* and *output* as routines that read with 80A1 format, but replace *lexshf* by a table lookup routine.

12 Summary of Source Format.

The general form of a Lex source file is:

```
{definitions}
%%
{rules}
%%
{user subroutines}
```

The definitions section contains a combination of

1)   Definitions, in the form "name space translation".

2)   Included code, in the form "space code".

3)   Included code, in the form

```
%{
code
%}
```

10-13

4) Start conditions, given in the form

%S name1 name2 ...

5) Character set tables, in the form

%T
number space character-string
...
%T

6) A language specifier, which must also precede any rules or included code, in the form "%C" for C or "%R" for Ratfor.

7) Changes to internal array sizes, in the form

%*x* *nnn*

where *nnn* is a decimal integer representing an array size and *x* selects the parameter as follows:

Letter	Parameter
p	positions
n	states
e	tree nodes
a	transitions
k	packed character classes
o	output array size

Lines in the rules section have the form "expression action" where the action may be continued on succeeding lines by using braces to delimit it.

Regular expressions in Lex use the following operators:

x	the character "x"
"x"	an "x", even if x is an operator.
\x	an "x", even if x is an operator.
[xy]	the character x or y.
[x-z]	the characters x, y or z.
[^x]	any character but x.
.	any character but newline.
^x	an x at the beginning of a line.
<y>x	an x when Lex is in start condition y.
x$	an x at the end of a line.
x?	an optional x.
x*	0,1,2, ... instances of x.
x+	1,2,3, ... instances of x.
x|y	an x or a y.
(x)	an x.
x/y	an x but only if followed by y.
{xx}	the translation of xx from the definitions section.
x{m,n}	*m* through *n* occurrences of x

13 Caveats and Bugs.

There are pathological expressions which produce exponential growth of the tables when converted to deterministic machines; fortunately, they are rare.

REJECT does not rescan the input; instead it remembers the results of the previous scan. This means that if a rule with trailing context is found, and REJECT executed, the user must not have used *unput* to change the characters forthcoming from the input stream. This is the only restriction on the user's ability to manipulate the not-yet-processed input.

TSO Lex is an older version. Among the non-supported features are REJECT, start conditions, or variable length trailing context, And any significant Lex source is too big for the IBM C compiler when translated.

14 Acknowledgments.

As should be obvious from the above, the outside of Lex is patterned on Yacc and the inside on Aho's string matching routines. Therefore, both S. C. Johnson and A. V. Aho are really originators of much of Lex, as well as debuggers of it. Many thanks are due to both.

The code of the current version of Lex was designed, written, and debugged by Eric Schmidt.

15 References.

1. B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice-Hall, N. J. (1978).

2. B. W. Kernighan, *Ratfor: A Preprocessor for a Rational Fortran,* Software — Practice and Experience, 5, pp. 395-496 (1975).

3. S. C. Johnson, *Yacc: Yet Another Compiler Compiler,* Computing Science Technical Report No. 32, 1975, Bell Laboratories, Murray Hill, NJ 07974.

4. A. V. Aho and M. J. Corasick, *Efficient String Matching: An Aid to Bibliographic Search,* Comm. ACM **18**, 333-340 (1975).

5. B. W. Kernighan, D. M. Ritchie and K. L. Thompson, *QED Text Editor,* Computing Science Technical Report No. 5, 1972, Bell Laboratories, Murray Hill, NJ 07974.

6. D. M. Ritchie, private communication. See also M. E. Lesk, *The Portable C Library,* Computing Science Technical Report No. 31, Bell Laboratories, Murray Hill, NJ 07974.