UCSD Pascal™

Software Library Manual

Texas Instruments Professional Computer

Cover Part No. 2232347-0001

UCSD Pascal
TI Part No. 2232401-0001
Original Issue: 15 April 1983

Preface

This manual is a reference for the UCSD Pascal programming language on the Texas Instruments Professional Computer. UCSD Pascal™ differs in some details from the Pascal language as originally designed by Professor Niklaus Wirth. Compared to the original language, it is better adapted for interactive use on a microcomputer.

The first part of this manual contains a detailed description of UCSD Pascal. After an overview of the language for those not already familiar with it, it takes a bottom-up approach, starting with lexical details, data types, syntax and statements, and then more advanced topics. The descriptions of Pascal written by Niklaus Wirth follow: we have found no reason to alter his approach. The wealth of data types, Pascal's most salient and subtle features, are presented as soon as possible.

The second part of this book is less formal. With the aid of examples, we have attempted to teach UCSD Pascal to a reader with programming experience, but no knowledge of any Pascal dialect. It should be useful to any readers with more knowledge than this minimum, but by no means is it intended as a course for beginning programmers. The text takes a conversational, problem-solving approach to the presentation of the language. Since we have assumed a variety of readers, and tried to cover as much of the language as possible, our examples include both numerical and non-numerical problems. Not all of our examples are immediately practical programs, but all of them attempt to illustrate practical techniques.

The sample programs in the second part of the book are printed from program source that was actually run and tested. To the best of our knowledge, they are correct.

Trademark of the Regents of the University of California.

Although we describe aspects of the UCSD p-System™ throughout this manual, it is about the UCSD Pascal language. For more complete information on the UCSD p-System itself, please refer to the *UCSD p-System Operating System Reference Manual* (2232395-0001).

Tutorials on the filer, editor, and UCSD Pascal can be found in *Personal Computing with UCSD p-System* (2232418-0001) by Overgaard and Stringfellow.

Trademark of the Regents of the University of California.

Contents

PART 2 A GUIDE FOR UCSD PASCAL PROGRAMMERS

Part 1

A Definition of UCSD Pascal

1

A Synopsis

This chapter is an overview of Pascal for readers who are not already familiar with the language. Those who have used UCSD Pascal before can skip it. Those who have used only other languages, or other dialects of Pascal, should read it to receive a general understanding for the language and the UCSD implementation.

A Pascal program is relatively formal, when compared with other languages such as BASIC, FORTRAN, or PL/I. Pascal's restrictions require the programmer to be more disciplined, but the advantages include:

- Syntax that is easily understood
- Implicit error-checking during compilation and runtime
- Freedom to concentrate on the algorithm rather than tricks of the language
- Modularity of the program's structure
- Readability

All of these things tend to make a program less error-prone and easier to maintain. Correctness and maintainability are the goals of structured programming, and Pascal was designed to promote such a style.

This chapter is informal, and is meant to convey the *flavor* of a Pascal program without going into details. What we describe here should give you an idea of what to expect in the chapters that follow.

In our examples of Pascal source (and elsewhere), reserved words are printed in **boldface** and predeclared identifiers are <u>UNDERLINED</u> and capitalized. This stresses the difference between reserved, predeclared, and user-created identifiers. We realize that program listings do not normally look like this.

THE FORM OF A PROGRAM

The text of a Pascal program begins with a heading, which is followed by an optional sequence of declarations, and a list of statements enclosed in the words **begin** and **end.** It ends with a period.

Here is a simple program:

```
program FirstOne;

begin
   WRITELN('Hello, out there!');
end.
```

DECLARATION

All data in a Pascal program has a given *type*. Some types are part of the language (they are said to be *predeclared*). There are the numeric types INTEGER, REAL, and long INTEGER. There is the logical type BOOLEAN, and the character type CHAR. Sequences of characters can be represented by the type STRING.

These are constants of various types:

1523	INTEGER
3.14159	REAL
'G'	CHAR
'Hello, out there!'	STRING

Constants can be given symbolic names (*identifiers*):

```
const
   OneDate = 1523;
     pi = 3.14159;
   initial = 'G';
   message = 'Hello, out there!';
```

In a Pascal program, all variables must be declared. The type of a variable never changes:

```
var
    Year: INTEGER;
    factor: REAL;
    letter: CHAR;
    greeting: STRING;
```

(The type of a variable in Pascal is much stricter than in some other languages (for example, FORTRAN, PL/I). For a program to compile successfully, variables must be treated according to the rules for their type.)

The programmer can define new scalar types. A scalar is a finite sequence of values with symbolic names:

```
var   WeekDay: (Sun, Mon, Tue, Wed, Thu, Fri, Sat);
```

Subranges of scalars are also available:

```
WorkDay: Mon .. Fri;   Subrange of WeekDay
Rating: 1 .. 10;        Subrange of INTEGER
```

In addition to the simple types, scalars, and subranges, Pascal provides the structured types **array**, **record**, and **set**. An array is a table of values (all of one type) grouped under a single name. A record is a group of values, possibly of different types. A set is a collection of values taken from a single base type.

Dynamic structures such as linked lists and search trees can be created using *pointers*. A pointer is a variable that points to another variable of a given type. The variable pointed to has no name; it is allocated or deallocated as needed.

Finally, long sequences of values can be contained in a file. In UCSD Pascal, a file is associated with a physical entity such as a disk file or a peripheral device.

EVALUATION

New values can be calculated by combining existing values in *expressions*. The result of an expression is of a particular type. Various operands are defined for different types.

Numeric expressions and numeric operators follow common algebraic conventions:

```
a + b + c
1.5 * ((i−1) * 2)
1 + 2 − 3 * 4
```

Relational operators can be used. Expressions with relational operators return a result of type BOOLEAN:

```
date = today
date <> yesterday      <> means "not equal"
profit > 10000
```

Pascal also provides some intrinsic functions, which appear in expressions:

```
pi * SQR(radius)
SQRT(SQR(b) − 4*a*c)

ODD(factor)            A Boolean function
```

Functions can also be defined by the user.

To initialize a variable or change its value, an assignment can be used. The symbol for assignment is :=. For example:

```
IsOdd:= ODD(factor);
area:= width * height;
RoundArea := pi*SQR(radius);
```

Each of these lines is called an *assignment statement*. The semicolon (;) is used to separate statements, as shown.

ACTION

In general, statements in Pascal are executed starting with the first statement in the main program, and proceeding to the end. Flow-of-control statements can be used to vary the order in which statements are executed by providing for decisions and repetition. Functions and procedures can be used to break the program into intelligible portions and improve its organization.

The **if** statement is used for a simple two-way (true/false) decision:

```
if profit > 10000 then
  PlanParty;

if profit > 10000 then
  PlanParty
else
  CallMeeting;
```

The **case** statement is used for multiple-way decisions:

```
case month of
  Apr, Jun, Sep, Nov: days:= 30;
  Jan, Mar, May, Jul, Aug, Oct, Dec: days:= 31;
  Feb: if leapyear then
          days:= 29
        else
          days:= 28;
end;
```

There are three loop constructs in Pascal. The **while** statement is the most general form:

```
while NotDone do
    FixIt;
```

The **repeat** statement is like a **while**, but always executes at least once:

```
repeat
    FixIt
until fixed;
```

The **for** statement uses a control variable, and repeats a given number of times:

```
for i:= 1 to 10 do
  clear;
```

Pascal also includes ways to perform branching as in BASIC or FORTRAN, but these are typically used only in emergency situations, if at all.

A **procedure** is a self-contained portion of code. The form of a procedure is much like the form of a program:

```
procedure Greet;

begin
    WRITELN('Hello, out there!');
end;
```

It is called by a statement that simply consists of the procedure's name:

```
Greet;
```

Parameters can be passed to a procedure when it is called:

```
procedure area (height, width: INTEGER; var result:
                INTEGER);
    begin
      result:= height * width;
    end;
```

This could be called by a statement such as:

```
area(12, 5, rectangle);
```

In this example, width and height are *value* parameters, while result is a *variable* parameter. In the code that calls the procedure, value parameters are unaffected, while variable parameters may be changed by the procedure (in the example, *rectangle* will be set to the area that the procedure calculates).

A **function** is much like a procedure, except that it returns a result:

```
function area(height, width: INTEGER): INTEGER;
    begin
      area:= height * width;
    end;
```

As we have seen, a function is called by using it in an expression:

```
rectangle:= area(12, 5);
```

or:

```
if area(NewHeight, NewWidth) <> 0 then PrintResult;
```

Pascal allows recursion. A procedure or function can call itself. Some algorithms are much more elegant when expressed recursively; Pascal can implement them directly.

COMMUNICATION

Simple (character) input/output (I/O) is accomplished with the intrinsic procedures READ, READLN, WRITE, and WRITELN. The intrinsic Boolean functions EOF (end of file) and EOLN (end of line) are provided for control.

If no other file is specified, these I/O intrinsics refer to the standard files INPUT and OUTPUT. In UCSD Pascal, INPUT and OUTPUT are both equivalent to the system's console.

File (record) I/O uses READ, WRITE, EOF, PUT, and GET. In UCSD Pascal, a file used within a program may refer to a system disk file or a peripheral I/O device. Random access is available with the intrinsic SEEK.

UCSD Pascal allows devices to be controlled directly (and swiftly) by a number of intrinsics including UNITREAD and UNITWRITE.

Also in UCSD Pascal, a file can be declared without a type and manipulated by the intrinsics BLOCKREAD and BLOCKWRITE. These are mainly used to transfer large portions of data swiftly.
of data swiftly.

MODULARITY

Code that is to be used by more than one program can be separately compiled. Such a code package is called a **unit**. A unit consists of an **interface** part, an **implementation** part, and optional initialization/termination code (units are a UCSD Pascal extension).

An interface part contains declarations and procedure or function headings. These can be used by the program (or other unit) that uses the unit.

An implementation part contains other declarations, and the code for the procedures and functions that were declared in the interface part. All of this information is strictly private to the unit.

It is possible to change the implementation part of a unit (to improve an algorithm, for example) and recompile it. If the interface part has *not* been changed, programs and units that use the unit may continue to do so; there is no need for them to be recompiled as well.

More than one unit can be grouped together in a single code file (often called a library). This can be useful, but is outside the scope of this book. Library handling is discussed in the *UCSD p-System Operating System Reference Manual.*

Units can also be useful for breaking up a program that is too long to compile in one piece.

MISCELLANY

This section describes some capabilities that pertain only to UCSD Pascal.

A procedure or function can be declared a **segment** procedure or function, in which case the system can swap it independently of the main program. This can be useful when running large programs that occupy a lot of space at execution time.

It is possible to write a routine that runs concurrently (that is, it shares processor time with other routines or programs). A routine that runs concurrently is called a **process**. A process looks like a procedure. It is not called, but an instance of it is set into execution by a call to the intrinsic <u>START</u>. Processes may be coordinated by the intrinsics <u>SIGNAL</u> and <u>WAIT</u>. They are especially useful for I/O and interrupt handling.

It is possible to write a procedure or function in assembly language, and call it from a Pascal program. This is done by declaring it **external**. The assembly language (*native code*) routine is responsible for conforming to Pascal's calling conventions. Such internal information is outside the scope of this handbook.

Finally, there are a number of compiler options that allow the following:

- Control of the compiled listing
- Control of the compiler's output
- Insertion of a copyright notice
- Conditional compilation
- Turning I/O checking off/on
- Turning range checking off/on
- Specifying a library
- Control of include files

EXECUTION

After a UCSD Pascal program has been compiled, it is run on the p-System by using the R(un or eX(ecute commands.

While a program is running, a number of things can cause a *runtime error*. A runtime error can be caused by a bug in the program, or by a mistake made by the person operating the program. When a runtime error occurs, the system aborts the program and displays a message that looks something like this:

```
Divide by zero Segment TEST   Proc 1   Offset 6 Type
   <space> to continue
```

If the program is at fault, it must be fixed. If the program's operator is at fault, the program can usually be started again by the U(ser-restart command.

Descriptions of the p-System commands and the full list of possible runtime errors can be found in the *UCSD p-System Operating System Reference Manual.* In this manual we indicate situations that will cause runtime errors, but the reader should be aware that the text and format of such messages are subject to change.

2

Lexical Conventions

A UCSD Pascal source program is essentially a stream of characters contained in a text file. For the program to compile successfully, it must conform to both the lexical standards of the language, and the syntactic (grammatical) rules. This chapter describes lexical standards which cover the legal character set, the formation of identifiers, the set of symbols and reserved words, the formation of comments, and a few other topics.

The term *token* is a compiler-writer's term that refers to a single symbol or name in the source program. It can be a special-character symbol, a reserved word, a constant, or an identifier. Since it is a useful general-purpose word, it appears throughout this chapter.

For a description of text files as they are maintained in the UCSD p-System, please refer to the *UCSD p-System Operating System Reference Manual.*

CONVENTIONS OF THIS HANDBOOK

We have used few conventions throughout this book. The intent has been to clarify our topics, not obscure them. In some cases, we use a form of EBNF (Extended Backus-Naur Form), already familiar to many readers. A description of EBNF appears at the end of this section.

A concise description of Pascal syntax in the form of railroad diagrams can be found in Appendix B.

Since UCSD Pascal is a particular dialect of the language, we have attempted to indicate where it differs from standard Pascal. When we say *standard Pascal*, we are referring to those features of Pascal that are common to both the original definition by Jensen and Wirth and the current American National Standards Institute (ANSI) draft standard. Where these two sources differ from each other, we have attempted to state that explicitly. As of this writing, the International Standards Organization (ISO) has a draft standard that is very similar to the ANSI draft.

When we describe a subrange, we shorten the conventional ellipsis (...) to .., as in the Pascal language itself. In other words, 1..6 represents the digits (or integers) one through six, as 1...6 would indicate in common mathematical notation. The book uses this notation because Pascal does.

In informal descriptions of syntax (and *not* in Pascal programs themselves), words enclosed in angle brackets (< >) are names of things; usually they represent non-printing characters such as <return> and <esc>.

EBNF

As we use it in this book, an EBNF expression is a description of some portion of Pascal syntax such as the following:

```
assignment-statement =
     variable-name ":=" expression
```

The EBNF expression consists of the name of some syntactic object, followed by an equal sign (=) and a description of the object.

Within the description, anything in quotes must appear literally in the Pascal program. A name that is *not* in quotes is the name of a syntactic object that is described elsewhere. Square brackets ([ ]) surround portions of syntax that are optional, and braces ({ }) surround portions of syntax that may appear. A vertical bar (|) separates different options, for example:

```
digit = "0"|"1"|"2"|"3" |"4"|"5"|"6" |"7"|
           "8"|"9"
```

Here is a slightly more complicated example:

```
repeat-statement =
     "repeat"
        [ statement {";" statement} ]
     "until" Boolean-expression
```

This EBNF expression tells you that a repeat statement consists of the reserved word **repeat**, followed by an optional statement list, the reserved word **until**, and a Boolean expression. The statement list itself consists of a single statement followed by zero or more instances of a semicolon preceding another statement.

Note that this expression does not define the terms *statement* or *Boolean-expression* as they are defined elsewhere.

In this manual, we use EBNF to clarify the more confusing parts of Pascal syntax, especially those that involve recursive structures. Since we have not attempted to construct a full axiomatic description of UCSD Pascal, we did not provide EBNF descriptions of those objects that appear in our EBNF descriptions.

THE CHARACTER SET

UCSD Pascal source files can contain the letters A..Z, a..z, the digits 0..9, and the following special characters:

() [] {} + − * / < = > : ; . , '^

The underscore (_) can appear in identifiers.

The following characters can appear (along with all other printable characters) in comments or strings, but they have no particular meaning:

! @ # $ % & ? | \ " ~ `

Blanks (' ') and carriage returns (<return>) can also be present. They serve to delimit identifiers, and format the file in a legible way.

UCSD Pascal uses ASCII (the American Standard Code for Information Interchange) to represent these characters. The ASCII code is shown in Appendix E.

SYMBOLS

A source program can contain symbols that are part of the Pascal language (special symbols and *reserved words*), identifiers that have been declared within the system (*predeclared identifiers*), identifiers that the user has defined, and literal text (contained within comments or strings).

Symbols that are already part of the Pascal language can be divided into special symbols of one or two characters, and reserved words.

Special Symbols

These are the one-character symbols:

```
.  ,  ;  :  '  ()  []  {}  +  -  *  /  =  <>  ^
```

These are the two-character symbols:

```
:=   ..   <=   <>   >=   (*   *)
```

The symbol `***` may appear within the code portion of a **unit**. This is not part of standard Pascal.

Reserved Words

Reserved words, like special symbols, are used in Pascal to represent syntactic constructs, such as particular kinds of declarations, particular forms of statements, and groups of statements.

These are the reserved words in UCSD Pascal (an asterisk indicates reserved words that are not part of standard Pascal):

and	goto	record
array		repeat
	if	
begin	*implementation	*segment
	in	*separate
case	*interface	set
const		
	label	then
div		to
do	mod	type
downto		
	not	*unit
else		until
end	of	*uses
*external	or	
		var
file	packed	
for	procedure	while
forward	*process	with
function	program	

Note the absence of **nil**. In UCSD Pascal, it is predeclared rather than reserved.

IDENTIFIERS

Identifiers represent constants, types, variables, and routines (procedures, functions, and processes). Some identifiers are predeclared, that is, they represent constants, types, or routines (but never variables) that the system has already defined.

Definition

An identifier consists of a letter, followed by an indefinite number of letters or digits. Single-letter identifiers are legal, and either upper- or lowercase, or a mixture of both, can be used.

Identifiers can also contain the underscore character (_), but it is not significant (its purpose is to make an identifier more legible).

An identifier cannot be the same as a reserved word. An identifier can be the same as another predeclared identifier (this can lead to problems; see Chapter 4, Structure and Scope).

These are legal identifiers:

```
i   Parity   try13   c2unit78   c2_unit_78
```

These are not legal identifiers:

4tran	Begins with a number
c2.unit.78	Contains special characters
try 13	Contains a space
_Parity	Begins with an underscore

Uniqueness

Upper- and lowercase are not distinct.

These identifiers are equivalent:

```
moss   MOSS   Moss   MosS
```

Only the first *eight* characters of an identifier determine its uniqueness.

These two identifiers are equivalent:

lostinspace lostinspectionitem

Embedded and trailing underscores (_) are ignored.

These identifiers are equivalent:

find_disk finddisk Find____Disk f_ind_d_isk finddisk_

The Jensen and Wirth definition of identifiers does not include the underscore character, but does specify eight-character uniqueness. Both the underscore and the eight-character stipulation conflict with the ANSI draft standard, which states that all characters of an identifier shall be significant.

Predeclared Identifiers

These are the predeclared identifiers in UCSD Pascal (an asterisk indicates identifiers not in standard Pascal):

ABS
ARCTAN
*ATAN
*ATTACH

*BLOCKREAD
*BLOCKWRITE
BOOLEAN

CHAR
CHR
*CLOSE
*CONCAT
*COPY
COS

*DELETE
DISPOSE

EOF
EOLN
*EXIT
EXP

FALSE
*FILLCHAR

GET
*GOTOXY

*HALT

*IDSEARCH
INPUT
*INSERT
INTEGER
*INTERACTIVE

*IORESULT

*KEYBOARD

*LENGTH
LN
*LOG

*MARK
MAXINT
*MEMAVAIL
*MEMLOCK
*MEMSWAP
*MOVELEFT
*MOVERIGHT

NEW
NIL

ODD
ORD
OUTPUT

PAGE
*PMACHINE
*POS
PRED
*PROCESSID
PUT
*PWROFTEN

READ
READLN
REAL
*RELEASE
RESET
REWRITE
ROUND

*SCAN
*SEEK
*SEMAPHORE
*SEMINIT
*SIGNAL
SIN
*SIZEOF
SQR
SQRT
*START
*STR
*STRING
SUCC

TEXT
*TIME
*TREESEARCH
TRUE
TRUNC

*UNITBUSY
*UNITCLEAR
*UNITREAD
*UNITSTATUS
*UNITWAIT
*UNITWRITE

*VARAVAIL
*VARDISPOSE
*VARNEW

*WAIT
WRITE
WRITELN

Note the absence of the standard predeclared identifiers PACK and UNPACK.

The routines IDSEARCH, PMACHINE, and TREESEARCH are for the system's use, and are not described in this handbook. The *UCSD p-System Internal Architecture Guide* describes PMACHINE.

COMMENTS

A comment is a passage of text that is ignored by the Pascal compiler. The purpose is to allow the programmer to explain the actions of the program in a language other than Pascal.

Comments can appear virtually anywhere in a source program. Like spaces or carriage returns, they delimit tokens, so they must appear *between* them. A comment cannot appear in the middle of an identifier, constant, reserved word, or two-character symbol.

A comment is any text enclosed by the delimiters { }, or (* *).

These are comments:

```
{ A comment with fancy delimiters! }
(*Another sort of comment, with less exuberance.*)
```

The delimiters cannot be mixed.

These are not comments:

```
(* Two kinds of delimiters here }
{ The same problem, in reverse *)
```

Comments with the same kind of delimiter cannot be nested.

This is not a legal comment:

```
{ This is an unsuccessful attempt {to create nested} comments. }
```

The compiler would read this as a comment ending with nested}.

A comment *can* contain a comment that uses the other type of delimiters.

These are legal comments:

```
(* This is a comment {that contains another}*)
{and so (*is this*)}
```

This construct is *not* legal in standard Pascal.

Comments can be longer than one line of source. If the compiler finds a comment that contains a semicolon (;), it issues a warning in the program's listing file. It is a common error to begin a comment, then forget to close it with a matching delimiter. The result is a comment that may swallow many lines of Pascal code (possibly the entire remainder of a program!). Since Pascal statements typically end with a semicolon, flagging semicolons within comments is a good way for the compiler to notify the programmer of this potential error.

Comments that contain a dollar sign ($) *immediately* after the first bracket are treated as instructions to the compiler. These comments must conform to a special format. A comment should not begin with a dollar sign unless the programmer wishes to invoke a specific compiler option. See Chapter 10 on compilation.

These are compiler option comments:

```
(*$I−*)
{$L list.5.text}
```

These are *not* compiler option comments:

```
{ $I−} Space before the $
{$M+} No M option at this time
```

TOKEN BOUNDARIES

Special-character symbols are tokens in themselves, but reserved words, constants, and identifiers must be clearly delimited. Except for special-character tokens, one token must be separated from the next by a special character, space, comment, or by pressing the **RETURN** key.

No reserved word, constant, or identifier can *contain* a special character, space, comment, or carriage return. This means, that tokens cannot cross the end of a line of source code. (A comment is not a token, and can cross the end of a line.)

INDENTATION AND LEGIBILITY

Aside from the restrictions mentioned in the previous section, there are no restrictions on the way a program can be arranged in the source file. However, there are some traditions concerning the visual format of a structured program, especially a Pascal program.

In general, a line of source code contains a single statement, or in the case of large compound statements, a single phrase or reserved word.

The hierarchy of statements that is implicit in a structured program is indicated in the source by indentation. Two or three spaces per level is usually favored. The **begin end, then else, case end, record end,** and **repeat until** pairs are typically indented so that they align vertically.

The last line of a program, or a relatively long routine, is usually a single **end**. This is often augmented with a comment that simply contains the name of the program or routine that is being ended.

This would not be considered a legible routine:

```
procedure decide;
var i: INTEGER; sortof: REAL;
begin
for i:=1 to 81 do
if choose(i) then move:=0 else begin
traverse(move,sortof);
respond(sortof)
end end;
```

This is the same routine, and would look acceptable to most programmers:

```
procedure decide;

  var i: INTEGER;
    sortof: REAL;

  begin
    for i := 1 to 81 do
      if choose(i) then
        move := 0
      else
        begin
          traverse (move, sortof);
          respond (sortof)
        end;
  end {decide};
```

However, we know programmers who would prefer to write it this way:

```
procedure decide;

  var i: INTEGER;
      sortof: REAL;

  begin
    for i := 1 to 81 do
      if choose(i) then
        move:=0
      else begin
            traverse(move,sortof);
            respond(sortof)
      end
  end {decide};
```

The important thing is that the programmer must consider the future readers of the program. Even the person who writes a program can be confused when he reads it some days (or hours) later.

3

Data Types and Operations

UCSD Pascal provides a large number of data types, each appropriate to certain applications. This chapter is therefore the longest in the handbook. Each data type is described in terms of its intended use, representation in the source program, limitations, legal comparisons and operations, and the intrinsic routines that operate on it. The internal representation of data types is discussed at the end of this chapter in the paragraph entitled Space Allocation for Data Types. Input and output of data is described in Chapter 7.

This chapter describes a number of intrinsic routines, but does not group them together. Appendix C contains a complete alphabetical list of the UCSD intrinsics.

INTRODUCTION

A Pascal program specifies a set of data, and a set of statements that operate on that data. The format of data, the variables that contain it, and the algorithms that use and modify it must all be specified explicitly.

Constants, Variables, and Expressions

In Pascal, the format of data is specified by its *type*. Pascal offers a variety of predeclared types. Unlike many languages, it also allows the user to define new types. Identifiers are used as names of types.

A *constant* is an object in a Pascal program that is a specific value of a specific type. As the name implies, it cannot be changed while the program is running. A constant can be a literal representation of the value, or an identifier that is declared as a constant.

A *variable* is an object in a Pascal program that is of a specific data type, and contains a value. A variable can contain only one value at any given time, but that value can be modified by an assignment statement, a procedure call, and so forth. Variables are represented by identifiers.

Both constants and variables can appear in *expressions*. An expression consists of constants, variables, and operators that yield new values. Operators are defined in certain ways for certain types, and the result of an expression is either a value of a specific type, or is undefined.

These are numeric expressions:

```
height+1
(width * height)/2
1+1+1-3+14/7
```

These are Boolean expressions:

```
glass=house
(EdgeTest and Emergency) < > finished
```

The identifiers in all of these examples might have been declared as either constants or variables. All identifiers must be declared prior to the body of the program, unit, or routine that uses them (see Chapter 4).

A function is a kind of routine that returns a value of a specific type. This value can be used in an expression, and hence the function call can be embedded in the expression itself:

```
a + 5*b + sqr(c)
not EOF(LinkFile)
not EOF
findpos(oper)/findpos(OpSum)
```

Functions are described in Chapter 5.

When a program runs, an expression can be evaluated, yielding some value. This value can be inspected on the spot and then forgotten, or it can be saved by assigning it to a variable (or printing it out, and so forth).

Assignment

An assignment statement consists of a variable, followed by the symbol :=, and an expression.

These are assignment statements:

```
x:= 6
graph := picture_file
dog := eat(dog)
i := i+1
muffle:=seeknoise(pipe,5)*2.137
```

The action of an assignment changes the value of the variable on the left of the := to the value of the evaluated expression. If the value of the expression is undefined or if it is an incompatible type, then an error results.

The full semantics of expressions and assignments will become clearer after the discussion of specific data types.

Evaluating Expressions

Two rules govern the order in which operators are evaluated within an expression.

The first rule is operator precedence; certain operations *take precedence* over other operations (they are evaluated first). For example, multiplication and division operations precede addition and subtraction. Thus, the following expression is TRUE:

```
5*3+4*6-2 = 37
```

The multiplication operations are evaluated first, followed by the addition and subtraction.

The second rule is that subexpressions can be grouped together by the use of parentheses. Subexpressions within parentheses are evaluated before the rest of the expression, and they can be nested. Thus, by modifying the previous expression, we can produce the following TRUE expressions:

```
5*(3+4)*6-2 = 208
```

or with nested parentheses:

```
5*((3+4)*6-2) = 200
```

Other than these two rules, the order in which operations are evaluated is undefined. The actual order is determined by the compiler.

If the result of an expression depends on the order in which it is evaluated, parentheses should be used. Sometimes the result of an expression will be known before the entire expression has been evaluated (for example, when a subexpression is multiplied by zero). When this is the case, it is possible that part of the expression will not be evaluated. The programmer should never assume that all operations in an expression will be carried out. This is especially important when a function call appears in an expression; since it will not necessarily be called, the program should not depend on any side-effects of its call.

The precedence rules for the operators on a given data type follow with the description of each type.

The Use of Data Types

There are two ways in which a program can be seen as a model of the real world. In one respect, a program can literally be an implementation of a model used to manipulate that model (store and retrieve data, calculate results from that data, and generate new data for other uses). In another respect, programs are used to control the machines on which they run (monitor input and output, issue instructions to the user, and accept instructions from the user).

In both of these cases, the data structures that the program uses should mirror the real-world situation. The more naturally data structures reflect the problem at hand, the easier it is to code the program, and the easier it is to use the program when it becomes a finished product.

Pascal presents a wide variety of data types for just this purpose. Simple quantities, integers or floating point numbers, can be represented by values of types INTEGER or REAL. Yes/no states can be represented by BOOLEAN values, and characters by values of type CHAR. The user can define new types (*scalars*), and simple types can be combined into more complex structured types that are richer and more useful representations of the real-world data that the program must deal with.

A full description of a set of data, of course, includes not only the data itself, but the possible actions that can be performed upon it (that is, the interrelation of data items). It is often the case that a single set of data-plus-algorithms can be useful to more than one program. The UCSD Pascal construct of the unit allows the programmer to define such a set; any number of programs (or other units) can use a unit, and a unit can be compiled separately from these *clients*.

SIMPLE DATA TYPES

The following paragraphs describe the simple data types that are predeclared in UCSD Pascal. It ends with a section that describes the routines that perform conversions from one data type to another.

A variable or constant that is of a simple data type has only one element. It is not a collection of values, either of the same simple data type (see **array**) or of different data types (see **record**) (both arrays and records are described in the paragraph entitled Structured Types).

Integer

The type INTEGER is used to represent integral values (whole numbers and their negatives).

Integer Format

An integer value is represented by a sequence of digits. It may be preceded by a − or + symbol. If no sign is present, the integer is assumed to be positive.

These are integers:

```
12345
51
+51
-51
-232
```

These are not integers:

1234.0	Contains a decimal point
/51	Contains a special character
89i5	Contains a letter

Integers are defined over the range − MAXINT .. MAXINT. The MAXINT range is a predeclared constant in each Pascal implementation; in UCSD Pascal it is equal to 32,767.

Integer Comparisons

These are the legal comparisons on integers:

=	Equal to
<>	Not equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to

Thus, the following comparisons are all TRUE:

```
17 = 17
32767 > −32767
32767 >= 0
13 <> 43
0 <= 0
```

Integer Operations

These operations yield results of type INTEGER. The operands can be of type INTEGER or a subrange of INTEGER (see the paragraph entitled Scalars and Subranges in this chapter). Integers can also appear in expressions that yield results of type REAL. These operations are described in the subsection on real numbers below.

These are the legal operations on a single integer:

+	Unary plus (identity)
−	Unary minus (change sign)

These are the legal operations on two integers:

+	Plus (addition)
−	Minus (subtraction)
*	Times (multiplication)
div	Integer divide (divide and truncate)
mod	Modulo (remainder of integer division)

Thus, the following expressions are all TRUE:

+2 = 2
−2 = −2
5+6 = 11
5−6 = −1
5*6 = 30
33 **div** 5 = 6
33 **mod** 5 = 3

The second operand of a **div** cannot be a zero for it causes a runtime error.

The operation **div** first performs the division, then truncates the result toward zero.

The operation i **mod** j is defined by:

for i > = 0: i − ((i div j) * j)
for i < 0: i − (((i+1) **div** j) * j) + j

j cannot be less than or equal to zero.

When an expression is evaluated, the multiplicative operators *, **div**, and **mod** take precedence over the additive operators + and −.

The following expressions are all TRUE:

345+10 **div** 5=347	The **div** is performed first
23*24*3+6=1662	The *'s are performed first
6+3*23*24=1662	The *'s are performed first

A unary operator cannot be strung together with a binary operator. The following expression is illegal:

```
5*−4
```

whereas the following expression is legal and TRUE:

```
5*(−4) = −20
```

To override operator precedence, subexpressions may be grouped together with parentheses. If there is any doubt about the order of the evaluation of an expression, parentheses should be used to ensure that the program states what the programmer intended.

The following expressions are all TRUE:

5+4+3+ 2 **div** 7=12	The **div** is performed first
(5+4+3+2) **div** 7=2	The portion in parentheses is evaluated first
5+4 − 3+17 = 23	
(5+4) − (3+17) = −11	
2+3 * 4 * 5+6=68	
2+3 * 4 * (5+6)=134	
2+3 * ((4 * 5)+6)=80	Nesting parentheses can be useful
(2+3) * ((4 * 5)+6)=130	

Integer Routines

The following paragraphs describe the two functions, ABS and SQR, that can take an integer value as an argument and return an integer value.

Some other intrinsic routines also deal with integers. The function ODD takes an integer value and returns a Boolean; the procedure STR converts an integer (or long integer) into a string; these are described in the paragraphs entitled Type Conversion at the end of this section. The function ORD takes a scalar value and returns an integer, as described in the paragraphs entitled Scalars and Subranges. The function SQRT takes either an integer or real value, and returns a real value; it is described in the following paragraphs on real numbers.

For information on the input and output of integer values, see Chapter 7.

ABS

ABS(I)

where:

I is an integer value (either a constant, variable, or expression), that returns the absolute value of I.

The following expressions are TRUE:

```
ABS(15) = 15
ABS(-15) = 15
ABS(12 - 45) = 33
ABS(-MAXINT) = MAXINT {= 32767}
96 = 12 * ABS(-13+5)
```

SQR

SQR(I)

where:

I is an integer value, returns the square of I.

The following expressions are TRUE:

```
SQR(15) = 225
SQR(1) = 1
SQR(-6) = 36
SQR(SQR(3)) = 81
```

Real

The type REAL is used to represent fractional numbers and numbers of very large or very small magnitude.

Real numbers are represented by a numerical portion (the *mantissa*), and an *exponent* that determines the position of the decimal point. This representation is called a *floating point* representation, and is similar to conventional *scientific notation.* More information on the internal representation of real numbers appears in the paragraph entitled Space Allocation for Data Types, at the end of this chapter.

Because the mantissa of a real number (as stored in the computer) contains a limited number of digits, real values must not be considered precise values; they are accurate only to a certain level of precision. Some advice on the use of real values in calculations and comparisons appears in Part 2, Chapter 2.

Real Format

A real value is represented by:

- A sequence of digits that contains a decimal point (.) (the decimal point must be preceded and followed by at least one digit)
- An integer value followed by an exponent (the letter *e* or *E* followed by an integer)
- A real value with a decimal point followed by an exponent

A real value can be preceded by the + or − symbols. In the absence of a sign, the value is assumed to be positive.

The exponent stands for *times ten to the power of* the integer that follows the letter *e* or *E*.

These are real numbers:

```
12345.0
1.2345
+12.4
−12.4
12e4
12.12e4
12.12e−4
12.12E−4
```

The range of real numbers depends on whether they are compiled into a two- or four-word format (two-word reals is the default). See the section, Space Allocation for Data Types, at the end of this chapter.

Real Comparisons

The comparisons on real numbers are the same as the comparisons on integers. The operands can be real, an integer, or a subrange of integer (see the subsection, Scalars and Subranges).

=	Equal to
<>	Not equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to

It is recommended that the = comparison *not* be used since representations of real numbers can be very close in value without being identical. Calculations and comparisons using real numbers are discussed in Part 2, Chapter 2.

Real Operations

These operations yield results of type REAL. The operands can be of type REAL, INTEGER, or a subrange of INTEGER (see the section Scalars and Subranges).

These are the legal operations on a single real value:

+	Unary plus (identity)
−	Unary minus (change sign)

These are the legal operations on two real values:

+	Plus (addition)
−	Minus (subtraction)
*	Times (multiplication)
/	Divide (division)

The second operand of a division (/) cannot be an expression whose result is zero since it causes a runtime error.

When an expression using REAL values is evaluated, the * and / functions take precedence over the + and − functions. As with integers, two real operations cannot appear in a row. For example:

3.0 * − 5.6 is illegal

while:

3.0 * (−5.6) is legal

Real Routines

The functions that return a real value, ABS, SQR, SQRT, SIN, COS, ARCTAN (or ATAN), EXP, LN, and PWROFTEN, are described in the following pages.

The functions TRUNC and ROUND are available to convert a real value to an integer; they are described in the paragraph entitled Type Conversions.

For information on the input and output of real values, see Chapter 7.

In the examples for the following function descriptions, the = should be read as *approximately equals* since values are shown only to three decimal places. Within memory, they would be stored with greater precision.

ABS

ABS(X)

where:

X is a real value (either a constant, variable, or expression), that returns the absolute value of X.

```
ABS(1.5) = 1.5
ABS(−1.5) = 1.5
ABS(−1.2*45) = 54.0
```

SQR

SQR(X)

where:

X is a real value, returns the square of X.

```
SQR(1.5) = 2.25
SQR(1.0) = 1.0
SQR(−6.0) = 36.0
```

SQRT

SQRT(X)

where:

X is a real or integer value, returns the square root of X.

```
SQRT(4.0) = 2.000
SQRT(7.0) = 2.646
```

SIN

SIN

where:

X is a real value or an integer value (in radians), returns the trigonometric sine of X.

SIN(1) = 0.841
SIN(3.14) = 0.002

COS

COS(X)

where:

X is a real value or an integer value (in radians), returns the trigonometric cosine of X.

COS(1) = 0.540
COS(3.14) = −1.000

ARCTAN or ATAN

ARCTAN(X)

where:

X is a real value or an integer value (in radians), returns the trigonometric arctangent of X. This function can also be called by ATAN(X) (this is a UCSD extension).

ARCTAN(0.3) = 0.291 {= ATAN (0.3)}
ARCTAN(0) = 0.000

EXP

<u>EXP</u>(X)

where:

X is a real value or an integer value that returns the constant e to the power of X.

<u>EXP</u>(1) = 2.718
<u>EXP</u>(6) = 403.429

LN

<u>LN</u>(X)

where:

X is a real value or an integer value, returns the natural logarithm of X (the logarithm with base e).

<u>LN</u>(3) = 1.099
<u>LN</u>(13) = 2.565

LOG

<u>LOG</u>(X)

where:

X is a real value or an integer value, that returns the logarithm base 10 of X. This function is a UCSD extension.

<u>LOG</u>(3) = 0.477
<u>LOG</u>(13) = 1.114

PWROFTEN

PWROFTEN(I)

where:

I is an integer value that returns a real value equal to 10 to the power of I. This function is a UCSD extension.

```
PWROFTEN(0) = 1.000
PWROFTEN(5) = 100000.0
```

Long Integer

Long integers are a UCSD extension to the type INTEGER. They are used to represent integers with a magnitude that can be greater than MAXINT or less than −MAXINT (they can represent integers within −MAXINT..MAXINT as well).

Long Integer Format

A long integer constant is declared by simply defining an integer constant with a magnitude outside the range −MAXINT .. MAXINT. For example:

```
const Rydberg = 10973731
```

A long integer variable is declared by INTEGER [n], where n, the *length attribute*, is an unsigned integer < = 36. The n represents the maximum number of decimal digits that the long integer may contain. For example:

```
var BigCount: INTEGER[10]
```

specifies that BigCount contains no more than ten decimal digits.

Long Integer Comparisons

The comparisons on long integers are the same as the comparisons on integers. The operands can be either integer or long integer values.

=	Equal to
<>	Not equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to

Long Integer Operations

The operations defined on long integers are the same as for integers, except that the **mod** operation is undefined:

+	Unary plus (identity)
−	Unary minus (change sign)
+	Plus (addition)
−	Minus (subtraction)
*	Times (multiplication)
div	Integer divide (divide and truncate)

When expressions using long integers are evaluated, intermediate results are allocated the necessary amount of space.

When a long integer is assigned the result of an expression that uses long integers, it must have been declared with enough digits to contain the resulting value, otherwise an overflow error occurs.

The compatibility of assignments using long integers is described in the paragraph entitled Type Conversions.

Long Integer Routines

There are no long integer routines per se. The function TRUNC can be used to convert a long integer to an integer, and the procedure STR can be used to convert a long integer to a string; see the paragraph entitled Type Conversions.

Because Pascal requires that a parameter type be declared by a type identifier, the following declaration would cause a syntax error:

```
procedure Large ( BigSum: INTEGER[10] )
```

To declare a parameter of type long integer, an appropriate type identifier must be declared, as follows:

```
type Digit10 = INTEGER[10];

. . .

procedure Large ( BigSum: Digit10 )
```

Long integers *cannot* be returned as function results. In terms of standard Pascal, this means that they are not a true simple type, although they are used in a manner similar to the types INTEGER and REAL.

For more information about procedures and functions, refer to Chapter 5.

Boolean

The type Boolean is used to represent logical truth values.

Boolean Format

A Boolean value can equal either TRUE or FALSE (these values are predeclared). FALSE is defined to be less than TRUE.

Boolean Comparisons

The following comparisons can be used with Boolean operands:

=	Equals
<>	Not equals (or XOR)
<=	Implies
>=	Is implied by
>	Does not imply
<	Is not implied by

Boolean Operations

The comparison operations that have already been described for INTEGER, REAL and long INTEGER all yield results of type BOOLEAN. The operands can be of any compatible ordered type (see the paragraphs entitled Type Conversions).

The following are operations on Boolean values only, and yield results of type BOOLEAN:

not	Logical negation (a unary operator)
and	Logical conjunction
or	Logical union

In expressions, **not** has the highest precedence of any Boolean operator, followed by **and** (at the same level as the multipliers *, /, **div**, and **mod**), followed by **or** (at the same level as + and −), followed by all of the relational operators (=, <>, >, <, >=, <=, and **in**, which is described in the subsection on Sets in the section on Structured Types).

Since the value of a Boolean expression may be known before the entire expression has been evaluated, the Pascal language does not require full evaluation of Boolean expressions. For example:

```
flag1  and  flag2  and  flag3
```

If flag1 is FALSE, there is no need to check the values of flag2 or flag3.

The order in which Boolean expressions are evaluated is chosen by the compiler. The programmer should be aware of this situation, and not write code that depends on an entire Boolean expression being evaluated. In particular, a function call that must be made (because of its side effects) for the program to work should never be embedded in a Boolean expression. (Conversely, the programmer should never assume that part of an expression will not be evaluated.)

Boolean Routines

The function ODD takes an integer value and returns a Boolean; it is described in the paragraph entitled Type Conversions.

The functions EOF and EOLN each return a Boolean value based on file operations; these functions are described in Chapter 7.

Boolean values cannot be written by any UCSD Pascal intrinsic; this is contrary to standard Pascal.

Characters

The type CHAR is used to represent individual characters. Character values are ordered. The digits 0..9 and the alphabets a..z and A..Z are contiguous within the character set.

Characters are often used as elements of sets and strings; the reader should refer to the paragraph entitled Structured Types.

Character Format

A printable character value is represented by a single character, surrounded by single quotes (apostrophes).

These are characters:

```
'a'  'B'  '/'  '7'  '['
```

An apostrophe is represented by typing it twice: ''''.

UCSD Pascal represents characters by using the ASCII character set. The ASCII characters are shown in Appendix E.

ASCII contains many nonprintable characters. Within the body of a program, a nonprintable character can be represented by using the intrinsic function CHR. See the paragraph entitled Type Conversions.

Character Comparisons

Character values have the same order as their underlying representation (the ASCII character set).

Because the character set is ordered, the numeric comparisons (=, < >, >, > =, <, < =) can be used on values of type CHAR.

Character Operations

Character values can be assigned to variables of type CHAR, and parameters of type CHAR can be passed, but there are no operations on characters.

Character Routines

There are no character routines per se. An integer can be converted to a character with CHR, and a character to an integer with ORD (see the paragraph entitled Type Conversions).

Character values can be read or written using READ and WRITE (see Chapter 7).

The intrinsics PRED and SUCC can be used with character values. These are described in the section on Scalars and Subranges.

For examples of converting characters to numeric values, or lowercase to uppercase and vice versa, see Part 2, Chapter 2.

Special-Purpose Types

UCSD Pascal defines two types that are used in the handling of concurrent processes. They do not appear in the standard language.

A PROCESSID is used by the system to distinguish concurrent processes. Every START process is assigned a unique PROCESSID. The programmer can examine this value, but cannot alter it.

A SEMAPHORE is used to synchronize concurrent processes. The intrinsic procedures SIGNAL and WAIT each depend on a parameter of type SEMAPHORE. A SEMAPHORE is initialized by the procedure SEMINIT, and can be associated with a hardware interrupt vector by the procedure ATTACH.

Concurrent processes are described in Chapter 9.

Type Conversions

The following subsections describe some ways to convert the type of a value. The first subsection describes the compatibility between types across an assignment (:=), and the second section describes intrinsic routines that can be used to explicitly convert a value.

Compatibility

An INTEGER variable can be assigned an integer value, or the result of an expression that contains (legal) operations on integers.

A REAL variable can be assigned a real value, an integer value, or the result of an expression that contains operations on real values or integers.

A long INTEGER variable can be assigned an integer value, a long integer value, or the result of an expression that contains operations on integers or long integers.

A BOOLEAN variable can be assigned the result of a comparison, or the result of an expression that contains operations on Boolean values.

Variables of type CHAR, PROCESSID, and SEMAPHORE, can be assigned a value of the same type, but are never operated on.

Subranges of the type INTEGER can be used wherever it is legal to use integers, but the overflow conditions of an expression depends on the subrange bounds. See the section on Scalars and Subranges.

Conversion Routines

The following pages describe intrinsic functions that can be used to convert a value of one type into a value of a different type.

TRUNC

TRUNC(X)

where:

X is a real value that returns an integer value equal to the whole part of X; the fractional part is discarded.

The following expressions are TRUE:

```
TRUNC(12.3) = 12
TRUNC(−12.3) = −12
TRUNC(67.0) = 67
```

TRUNC(L)

where:

L is a long integer value returns an integer value equal to L. If L is not in the range −MAXINT .. MAXINT, an overflow results.

ROUND

ROUND(X)

where:

X is a real value that returns the integer value nearest X.

The following expressions are TRUE:

```
ROUND(12.3) = 12
ROUND(12.7) = 13
ROUND(4.5) = 5
ROUND(−4.5) = −5
ROUND(67.0) = 67
```

ODD

ODD(I)

where:

I is an integer value, and returns a Boolean value that is TRUE if I is odd, and FALSE if I is even.

ORD

ORD(C)

where:

C is a character value, and returns an integer value equal to the ordinal number of C within the character set.

ORD applies to scalar and subrange types (including BOOLEAN), as well as to CHAR (see the paragraph entitled Scalars and Subranges).

CHR

CHR(I)

where:

I is an integer value, and returns a character value equal to the character with ordinal number I within the character set.

These functions are opposites:

```
ORD(CHR(I)) = I
CHR(ORD(C)) = C
```

The following expressions are true:

```
ORD('A') = 65
ORD(' ') = 32

CHR(32) = ' '
CHR(93) = ']'
CHR(3) {= ETX (which is not printable)}
```

STR

The STR(L,S) procedure sets the string variable S to a representation of the value of the integer or long integer L.

SCALARS AND SUBRANGES

Scalars

Scalar types consist of an enumeration of values. The name of the type is an identifier. You can define new scalar types, whose values are represented by identifiers. A user-defined scalar is usually declared as a **type** — it can also be declared as a **var**, but this is less useful.

Here is a program fragment defining a few scalar types:

```
type
    color = (red, yellow, blue, green, lavender, purple,
             mauve, amber);
    month = (Jan, Feb, Mar, Apr, May, Jun,
             Jul, Aug, Sep, Oct, Nov, Dec);
    sex = (male, female);
    DoorState = (open, closed);
```

Within the scope of the type declaration (see Chapter 4), a scalar value must be unambiguous; the same identifier cannot appear in the definition of two different scalar types.

The following program fragment is illegal:

```
type
    DoorState = (open, closed);
    LockState = (open, locked);
```

The values of a scalar type are ordered. The relational operators (=, <>, >, >=, <, <=) are defined for scalar types, and have their usual meanings, based on the order in which the scalar values are declared.

Using the (legal) types previously defined, the following expressions would be TRUE:

```
Jan <> Feb
blue <= lavender
female > male
```

Subranges

The programmer can define a type or a variable that is a subrange of a previously declared scalar type. The symbol .. denotes intervening values (*red .. green* means *red through green*).

Given our previous examples, these are legal declarations:

```
type
    winter = Jan .. Mar;
    spring = Mar .. Jun;
    primary = red .. blue;

var
    summer: Jun .. Aug;
    palette: primary;
```

The following two declarations describe equivalent subranges:

```
tag: sex;
gender: male .. female;
```

Note that subranges can overlap.

Subranges of predeclared scalar types are also frequently used:

```
type
    f_index = 1..77;           Subrange of INTEGER

var
    Kinsey: 1..7;              Subranges of INTEGER
    state: 0..5;
    grade: 'A' .. 'F'          Subrange of CHAR
```

Any legal operations on a predeclared type are always legal on a subrange of that type, but the overflow conditions may vary:

```
state:= 3;                  This is ok
state:= state+1;            Result is 4; no problem
state:= state-10;           Result < 0; a value range
                            error occurs at runtime
```

Note that the types REAL and long INTEGER are not considered scalar types, and cannot be used to construct subranges.

Predeclared Standard Types

Some of the simple types we have discussed may be thought of as scalar or subrange types, for example:

```
INTEGER = -MAXINT .. MAXINT
```

or

```
INTEGER  =  -32767..32767      For UCSD Pascal,
                               this is an equivalent
                               declaration
BOOLEAN = (FALSE, TRUE)
```

This is not the way that these types are represented internally, but they do behave as if they were declared this way.

Note that the types REAL and long INTEGER are not considered scalar types.

Scalar and Subrange Routines

The following pages describe the three intrinsic functions that are provided for manipulating scalar and subrange values: ORD, PRED, and SUCC.

ORD

ORD(V)

where:

V is a value of a scalar type and returns an integer that is the ordinal value of V in the sequence of values declared for that type.

All scalar types are ordered (as the name implies), and their values are numbered starting from zero.

We have already seen (in the paragraph, Type Conversions) ORD used to convert character values to integer values. This is a special case of the use of the ORD function.

Given the declarations earlier in this section, the following expressions are TRUE:

```
ORD('Z') = 90      (using ASCII characters)
ORD(blue) = 2
ORD(Jan) = 0
ORD(TRUE) = 1
```

Note that ORD has no inverse function, except for the special case of CHR (see the description of CHR).

PRED

PRED(V)

where:

V is a value of a scalar or subrange type, and returns the value that precedes that value. PRED stands for *predecessor*.

Given the declarations earlier in this section, the following expressions are TRUE:

```
PRED('Z') = 'Y'
PRED(blue) = yellow
PRED(Dec) = Nov
PRED(TRUE) = FALSE
```

If the value V is the first value in the scalar type (that is, if ORD(V) = 0), then PRED(V) results in a value range error at runtime (unless range-checking has been turned off (see Chapter 10).

SUCC

SUCC(V)

where:

V is a value of a scalar or subrange type, and returns the value that SUCCEEDS that value. SUCC stands for *successor*.

Given the declarations earlier in this section, the following expressions are TRUE:

```
SUCC('A') = 'B'
SUCC(blue) = green
SUCC(Jan)= Feb
SUCC(FALSE) = TRUE
```

If the value V is the last value in the scalar type (that is, if ORD (V) is the greatest possible value for that type), then SUCC(V) results in a value range error at runtime (unless range-checking has been turned off—see Chapter 10).

STRUCTURED TYPES

A *structured type* in Pascal is a single type built out of simple types in certain ways, and given a single name. The following paragraphs discuss arrays, strings, records, and sets. Files are discussed in the paragraph entitled Files.

An *array* is a table of values all of the same type. It corresponds to the notion of a matrix in mathematics, and can have one or more dimensions.

A *string* is a sequence of characters. Unlike an array, the length of a string can change during the execution of a program. UCSD Pascal provides several intrinsics for the manipulation of strings.

A *record* is a group of values of (possibly) mixed type. Records are useful for maintaining information that is logically grouped together, but best represented by a variety of types.

A *set* is, in the mathematical sense, the powerset of its base type. In other words, a set value is an unordered collection of values from the base type. Sets are useful for truth tables and tests of membership.

Arrays

An array is a table of values. The values in an array must all be of one type, the *base type* of the array.

An array can have one or more dimensions. The number of dimensions and the size of each dimension cannot change during the execution of the program.

The individual elements in an array are also called *components*. Each individual element can be referenced in a program by the name of the array, followed by an *index* (also called a *subscript*) surrounded by square brackets ([ ]).

The following expressions reference array elements:

```
Directory[45]
Year[month]
TokenList[i+4]
CubePoints[0,0,0]
Hexagram [Upper3] [Lower3]
```

Array Format

An array is declared in the following way (using Extended BNF):

```
array-type = "array" "[" ordinal-type {","
                  ordinal-type } "]""of"base-type
```

where:

the *ordinal-type* defines the bounds of the array, and *base-type* is either a type declaration, or the name of a type that has already been declared.

Each *ordinal-type* is a subrange expression, or the identifier of a scalar or subrange type that is called the *index type* of the array.

The *base-type* can be any type except a file type. It may well be another array.

The following would be legal array declarations:

```
var
        Students: array [1..ClassSize] of Grade;
        logout: array [day] of time;
        LastQuarter: array [1..ClassSize] of Boo-
        lean;
                state: array [0..49] of 1..7
                schedule: array [day] of
                      array [9..18] of initials;
        Cube: array [0..2, 0..2, 0..2] of color;
```

Arrays with multiple dimensions can be indexed by the following:

```
schedule [monday] [13]
```

a shorter form:

```
schedule [monday, 13]
Cube[1,1,0]
```

an expression:

```
Cube[1,1,0]:=red
```

An array declaration can be preceded by the keyword **packed**, as follows:

```
surname: packed array [0..19] of CHAR
```

The semantics of using **packed** are described in the paragraph Space Allocation for Data Types, at the end of this chapter.

The size of an array is limited only by the maximum number of words that may be local to a routine or compilation unit (unit or main program). This is currently 16,383 words.

Standard Pascal requires that a **packed array of** CHAR have at least two elements. UCSD Pascal does not have this restriction.

Array Comparisons

Two arrays can be compared using the operators = (equal) and < > (not equal). This is not legal in standard Pascal. The arrays must have the same dimensions and same base type, and should not be packed.

Any other comparisons involving arrays must be done element-by-element, for example:

```
same:= TRUE;
i:= 0;
while (i < = maxelement) and same do
    if a[i] = b[i]
       then i:= i+1
       else same:= FALSE;
```

Array Operations

A single assignment can be used to assign an entire array value to another array, provided both arrays have the same dimensions and the same base type. This is a UCSD extension.

A **packed array of** CHAR can be assigned another **packed array of** CHAR value of the same length.

There are no Pascal operations that apply to arrays. The individual elements of an array can be operated upon, following the rules that apply to the base type of the array.

For example:

```
vector[5]:= vector[5] + table[5, 12];
```
(the base type could be a numeric type or a set)

Array Routines

UCSD Pascal provides four intrinsic routines for the rapid manipulation of arrays. They are most frequently used to handle packed arrays of character, but they do no type checking on their operands, and so are generally applicable. The following pages describe the procedures FILLCHAR, MOVELEFT, and MOVERIGHT, and the function SCAN.

These four intrinsics have certain parameters that can be of any type (much like parameters to READ and WRITE). These are described below (not quite accurately) as typeless parameters. Because of Pascal syntax, these parameters must be declared as having a specific type, but the intrinsics merely operate on main memory at the location of the parameter, and do not check what type it is.

If a typeless parameter is an array, it can have a subscript. If it is a record, it can have a field specification. These specify a location in memory where the intrinsic will operate.

Because of the generality of these routines, and because they do no type checking or range checking whatsoever, they should used with extreme caution, lest valuable information be destroyed.

FILLCHAR

The FILLCHAR (DESTINATION, LENGTH, CHARACTER) procedure fills an area of memory with a single character.

DESTINATION is a typeless parameter. LENGTH is an integer.

CHARACTER is a single CHAR, or an integer (FILLCHAR ignores the eight most-significant bits of the integer, so it should be in the range 0..255).

The FILLCHAR procedure fills memory with LENGTH instances of CHARACTER (two characters per word), starting from DESTINATION.

For example, given the declaration:

```
var buf: packed array [0..19] of CHAR;
```

the statements:

```
for i:= 0 to 19 do buf[i]:= '* ';
FILLCHAR(buf, 10, 'e');
```

set buf to this value:

```
'eeeeeeeeee**********'
```

MOVELEFT

The MOVELEFT (SOURCE, DESTINATION, LENGTH) procedure moves LENGTH bytes from SOURCE to DESTINATION. The bytes are moved from left to right.

SOURCE and DESTINATION are typeless parameters. LENGTH is an integer.

For example, given the array initializations:

```
src:= '1234567890';
dst:= '**********';
```

the call:

```
MOVELEFT(src, dst, 5);
```

sets dst to: '12345*****'

MOVERIGHT

The MOVERIGHT (SOURCE, DESTINATION, LENGTH) procedure moves LENGTH bytes from SOURCE to DESTINATION. The bytes are moved from right to left.

SOURCE and DESTINATION are typeless parameters. LENGTH is an integer.

The MOVELEFT and MOVERIGHT procedures both accomplish the same thing, except when bytes are moved to an overlapping location within the same array.

For example, given the same value of the array src, the following call correctly sets src to '1231234590'

```
MOVERIGHT(src, src[3], 5);
```

On the other hand, the following call sets src to '1231231290' because bytes are modified before they have been moved.

```
MOVELEFT(src, src[3], 5);
```

SCAN

The SCAN (LENGTH, <partial expression>, SOURCE) function returns the location of a character within an array.

LENGTH is an integer.

The <partial expression> is an = or <> symbol followed by a single character expression.

SOURCE is a typeless parameter.

The SCAN function scans SOURCE until the partial expression is satisfied or until LENGTH characters have been scanned; whichever comes first. It returns an integer value that is the offset from the beginning of SOURCE to the point at which it stopped scanning.

If LENGTH is negative, SCAN scans from right to left rather than left to right, and returns a negative offset.

For example:

```
var test: STRING;

    test:= 'For he on honey dew hath fed,';

    index:= SCAN(10, ='h', test[11]);
            (index is set to 0)

    index:= SCAN(10, = 'h', test[12]);
            (index is set to 9)

    index:= SCAN(-9, = 'h', test[9]);
            (index is set to -4)

    index:= SCAN(10, <> 'h', test[11]);
            (index is set to 1)
```

Strings

A string is a sequence of characters that has an associated length. The length of a string can change during the execution of a program.

Several intrinsics are provided to simplify the manipulation of strings.

Strings and string intrinsics do not appear in standard Pascal.

String Format

The STRING sequence is a predeclared type. A string value is a sequence of characters with an associated length. A string constant is typed as a sequence of characters surrounded by single quotes (apostrophes).

The following are string constants:

```
'hello there'
'And whether pigs have wings.'
{embedded quotes are typed twice:}
''''   {a single apostrophe}
'The sixth sick sheik''s sixth sheep''s sick'
''   {the empty string}
```

The empty string is allowed, and has a length of zero.

Each string variable has a maximum length. The default is 80 characters. This can be overridden when the string is declared, by following the identifier STRING with a length attribute in brackets ([ ]). A string cannot be declared longer than 255 characters.

The following are declarations of string variables:

```
heading: STRING;               Default maximum
                               length is 80
graphline: STRING[200];        Has a maximum
                               length of 200
surname: STRING[20];
abbrev: STRING[3]
```

String Comparisons

Strings are ordered, and can be compared with any of the comparison operators. The ordering of strings is lexicographical (dictionary order), shorter strings precede longer strings, and uppercase precedes lowercase.

String Operations

Strings can be assigned. No other operations are defined on strings.

The characters in a string are indexed from one up to the dynamic length of the string. The dynamic length of the string cannot be greater than the string's maximum length (its *static length*).

If a string is indexed outside its bounds, a value range error occurs when the program is executed (unless range-checking is disabled: see Chapter 10). The empty string cannot be indexed at all.

String Routines

The simplest way to manipulate strings is to use the string intrinsics. The following pages describe the functions CONCAT , COPY, LENGTH, and POS, and the procedures INSERT and DELETE.

CONCAT

The CONCAT (SOURCE1, SOURCE2, ..., SOURCEn) function returns a string that is the value of the concatenation of strings SOURCE1 .. SOURCEn.

The SOURCE strings are string expressions, and there can be any number of them, separated by commas.

The length of the new string is the sum of the lengths of the sources.

For example:

```
CONCAT('All in ', 'a garden ' , 'green ...')
```

returns the string:

```
'All in a garden green ...'
```

```
CONCAT('There is a long poisonous ',
            CONCAT ('snake-like ', ' object'))
```

returns the string:

```
'There is a long poisonous snake-like object'
```

COPY

The COPY (SOURCE, INDEX, SIZE) function returns a substring of SOURCE that is SIZE characters long and starts at SOURCE[INDEX]. If SIZE is too long (INDEX + SIZE − 1 > LENGTH(SOURCE)), then COPY does nothing.

SOURCE is a string variable; INDEX and SIZE are integers.

For example, if:

```
Long:= 'Fortune my foe, why dost thou frown on me?'
```

then:

```
COPY(Long, 17, 19)
```

returns the string:

'why dost thou frown'

DELETE

The DELETE (DESTINATION, INDEX, SIZE) procedure removes SIZE characters from DESTINATION, starting from DESTINATION[INDEX]. If SIZE is too long (INDEX + SIZE − 1 > LENGTH(SOURCE)) then DELETE does nothing.

DESTINATION is a string variable; SIZE and INDEX are integers.

For example:

```
DELETE(Long, 17, 19)
```

replaces Long with the string:

'Fortune my foe, on me?'

INSERT

The INSERT (SOURCE, DESTINATION, INDEX) procedure inserts the string SOURCE into the string DESTINATION, starting from DESTINATION[INDEX]. The new length of DESTINATION is its old length plus LENGTH (SOURCE).

SOURCE is a string expression, and DESTINATION is a string variable; INDEX is an integer.

For example, if:

```
Long:= 'There were three ravens,';
Short:= 'old '
```

then:

INSERT(Short, Long, 18)

replaces Long with the string:

'There were three old ravens,'

LENGTH

The LENGTH (SOURCE) function returns the current (dynamic) length of the string SOURCE.

SOURCE is a string expression; LENGTH returns an integer.

For example, if:

```
Long:= 'At noon Dulcina rested'
```

then:

```
LENGTH(Long)
```

returns the integer 22.

POS

The POS (PATTERN, SOURCE) function returns an integer that is the location of the string PATTERN in the string SOURCE.

The integer that POS returns is the index of the first character of the matching substring.

If PATTERN cannot be matched, POS returns 0.

For example, if:

```
Long:= 'He that would an alehouse keep';
Short:= 'use';
```

then:

```
POS(Short, Long)
```

would return the integer 23.

Sets

A set value is a collection of members that are values from some scalar or subrange type. In mathematical terms, a set type is the powerset of its base type.

Set Format

A set is declared in the following way (using extended BNF):

```
set-type = "set" "of" ordinal-type
```

where:

ordinal-type is a scalar type or a subrange.

These are some set variable declarations:

```
ASCII:   set of CHAR;
palette:   set of color;
attribute:   set of 1..5;
LowerCase:   set of 'a' . . 'z';
```

The value of a set can be represented in the following way (again using EBNF):

```
set-value = "[" [ member {"," member } ] "]"

member = expression [ ". ." expression ]
```

In other words, a set value is an enumeration of values or subranges of values, enclosed in square brackets. The empty set is denoted by [ ].

For example:

```
palette:= [red, yellow, blue];
attribute:= [1,3];
newset:= [ ]   {the empty set}
```

The following expression is TRUE since set elements are not ordered:

```
[red, blue, green] = [green, red, blue]
```

A set can have up to 4,080 elements (16 bits/word * 255 words). A set of a subrange of INTEGER must have positive bounds, and the upper bound must be no greater than 4,079, regardless of the value of the lower bound. These restrictions apply only to UCSD Pascal.

Set Comparisons

The following are legal comparisons on sets:

=	Equal to
<>	Not equal to
>=	Includes (is a superset of)
<=	Is included in (is a subset of)

The comparison **in** is also defined for sets.

```
<value> in <set>
```

is a Boolean expression. <value> is a scalar or subrange expression from the base type of <set>. If <value > is indeed a member of this <set>, the value of the expression is TRUE.

For example, if IsLetter is BOOLEAN and ch is a CHAR, then:

```
IsLetter:= ch in ['A'..'Z', 'a'..'z']
```

determines whether ch is in the alphabet.

Comparisons of sets are only legal if the two sets have the same base type (or if the subranges on which they are based have the same base type).

Set Operations

The following operations are defined on sets:

+	Set union
*	Set intersection
−	Set difference

The following expressions are TRUE:

```
['C', 'A', 'R'] + ['H'] = ['C', 'A', 'R', 'H']
['C', 'A', 'R'] * ['H'] = [ ] (the empty set)
['N', 'W'] * ['E', 'W', 'D'] = ['W']
['C', 'A', 'R'] − ['H'] = ['C', 'A', 'R']
['N', 'W'] − ['E', 'W', 'D' = ['N']
[blue, red, green] + [gold] = [blue, red, green,
gold]
```

As with set comparisons, set operations are only legal if the sets or their base subranges have the same base type.

Set Routines

There are no intrinsic routines for the manipulation of sets. Nor is there any provision in Pascal for the standard output of set values. The user must create routines that are appropriate to the purposes of a particular program.

Records

A record value is a collection of values that are possibly of different types.

Record Format

The following expressions in EBNF describe the declaration of a record:

```
record-type = "record" field-list "end"

field-list = fixed-part [";"]
        | variant-part [";"]
        | fixed-part ";" variant part [";"]

fixed-part = id-list ":" type { ";" id-list ":" type }

variant-part = "case" ordinal-type "of" variant
             | "case" tag ":" ordinal-type "of"
             variant

variant =id-list ":" "(" field-list ")"
           { ";" id-list ":" "(" field-list ")" }

id-list =identifier { "," identifier }
```

As indicated, variant-parts in a record must always follow fixed-parts, and there is only one of each at any given *level* of the record.

The id-lists in a variant must be constants of the type designated in the **case** heading of the variant-part.

Tag is an identifier.

Note that the keyword **end** pairs with the keyword **record**.

The field-list cannot be empty although it can in standard Pascal.

The following are some record type declarations:

```
type
   complex =  record
                 RealPart,
                 ImaginaryPart: REAL
              end;

   student =  record
                 surname: STRING[30];
                 score1, score2: integer;
                 grade: 'A'..'F';
                 again: BOOLEAN
              end;

   disk =  record
              artist, composer: STRING[50];
              size: (EightInch, TenInch, other);
              release: integer;
              condition:
                 record
                    chipped,
                    humor,
                    jazz,
                    tradeable: BOOLEAN
                 end
           end;
```

It should be evident that records can be as simple or as complex as you desire.

The elements of a record can be accessed by <record name>.<field name>. Given the declarations:

```
var
  class: array [1..50] of student;
  platter: disk
```

fields could be accessed by the following names:

```
class[1].surname
class[j].grade
platter.release
platter.conditioi. tradeable
```

Record Comparisons and Operations

As with arrays, the comparisons = and <> can be used to compare records of the same type that are *not* packed. No other comparisons are defined for records.

No operations are defined on records. Fields of records can be operated on, according to the type of the field.

Variant Records

Here are some examples of records with variants:

```
entry = record
           case head: BOOLEAN of
              TRUE: (number: INTEGER);
              FALSE: (identifier: STRING[5]);
           end;

choice = (addr, bits);
trix = record
          case choice of
             addr: (locn: INTEGER);
             bits: (bmap: packed array
                             [0..15] of BOOLEAN);
        end;
```

A variant record allows the programmer to treat a single field (a single memory location) as a variable that has a different type in different situations.

If the variant does not have a tag variable, then the names of the fields within the variant list are used as any other record field name. For example:

```
var
    two__way: trix

begin
    . . .
    two__way.locn:= 17760;
    . . .
    two__way.bmap[0]:= FALSE;
```

If the variant DOES have a tag variable, then the tag variable is itself a field. For clarity, it can be set to a particular value before the field corresponding to that value is used. For example:

```
var node: entry;

begin
  . . .
  node.head:= TRUE;
  node.number:= 57;
  . . .
  node.head:= FALSE;
  node.identifier:= 'WOOF0';
```

Because variant records can treat the same area of memory in different ways, their use can become involved; variant records can be a means of doing dirty tricks. See Part 2, Chapter 8.

The With Statement

Specifying an element of a record by enumerating all of the appropriate fields can become tedious, and interfere with the legibility of a program. The with statement provides a shorthand means of dealing with field names.

A **with** statement has the following form:

with < record name list> **do** < statement>

where:

< statement> is usually a compound statement; see Chapter 6. < record name list> consists of one or more identifiers of record variables (separated by commas).

If multiple record names are used, the names of their fields must be unambiguous (that is, a field name cannot be used within the **with** statement if it is common to more than one of the records). If a simple variable has the same name as the field of a record, then within the **with** statement, the *field* name takes precedence (and the simple variable cannot be used).

The following is an example of the **with** statement:

```
with class[n] do
    begin
      surname:= buffer;
      score1:= 0;     (initial values!)
      score2:= 0;
      grade:= 'A';   (benefit of a doubt)
      again:= FALSE;
    end;
```

The following is a slightly more complicated example:

```
with platter.condition do
    begin   {initializations}
       chipped:= FALSE;
       humor:= FALSE;
       jazz:= FALSE
       tradeable:= FALSE
    end {with};
```

We could also have the (unlikely) example:

```
with class[n], platter do
     begin
          grade:= 'C';
          artist:= 'Ellington';
     end;
```

DYNAMIC TYPES

In Pascal, *dynamic types* are data structures whose size and configuration may change during the execution of a program. This section describes files, which store data serially, and pointers, which are a means of indirectly referencing data. Pointers can be used to build flexible data structures such as search trees.

Files

A file is a serial stream of data. The data can be read or written from within a Pascal program.

In standard Pascal, files are restricted to serial access for the sake of simplicity. In the UCSD p-System, programs, text, and data are all stored in random-access files on some form of disk; therefore, UCSD Pascal provides for random access. UCSD Pascal also deviates slightly from standard Pascal in order to better handle interactive terminals.

When we refer to an *external* file, we are talking about a file on a disk, which has a name in the disk's directory. When we refer to an *internal* file, we are talking about a Pascal program's description of some file. The reason for this distinction is that the file name in the program does *not* need to be the same as the external file name; in fact, when a program handles disk files, it is most useful to make it capable of handling many different disk files with different (external) names.

This section deals with internal files (that is, only with a Pascal program's ways of handling a file). The way to connect a program's file name with an external file name is described in Chapter 7. Also see Part 2, Chapter 4.

Internal File Format

A file can be declared in the following manner:

```
file of < base type>
```

The base type of a file can be any type but another file type (this restriction does not apply in standard Pascal).

These are some file declarations:

```
var
    Chapter1: file of CHAR;
    enrollment: file of student;
    seismic: file of INTEGER;
```

A file consists of a serial stream of elements.

A file with no elements at all is allowed, and is called an *empty* file.

A file can be modified only at a single location, which is called the file *window*. The window is associated with a *window variable* which contains the value of the file element at the window's location. If FileName is the name of a file, FileName∧denotes the window variable.

The window variable can be modified just as an ordinary variable (with an assignment, a routine call, and so forth). It is initialized when the file is opened, or by a call to the intrinsic GET. Its value can be written to the file by a call to the intrinsic PUT.

In standard Pascal, the window is always at the end of the sequence of file elements. In UCSD Pascal, files can be accessed randomly by repositioning the window with a call to the intrinsic SEEK, which is described in Chapter 7.

The predeclared type TEXT is defined as:

```
TEXT = file of CHAR;
```

Files of type TEXT are usually saved as p-System text files (a p-System text file must have the suffix .TEXT in its external name).

In UCSD Pascal, a file can be declared without a type, for example:

```
NoStructure: file;
```

Input and output to untyped files can only be done with the UCSD intrinsics BLOCKREAD and BLOCKWRITE; see Chapter 7.

Standard Pascal defines the predeclared files INPUT and OUTPUT. In UCSD Pascal, these files normally refer to the device *console*:, which is an integral part of the p-System environment. (This is a default which can be overridden by the user — see the description of redirection in the *UCSD p-System Operating System Reference Manual.*)

Character-oriented devices (such as CONSOLE:, SYSTERM:, PRINTER:, REMOUT:, and REMIN:) are usually opened as text files when they are used within a Pascal program (see the REWRITE and RESET intrinsics in Chapter 7).

In UCSD Pascal, INPUT, OUTPUT, and the predeclared file KEYBOARD are files of the predeclared file type INTERACTIVE. An INTERACTIVE file has the same structure as a TEXT file, but is treated somewhat differently by the intrinsic routines READ, READLN, and RESET (see Chapter 7).

A programmer can define a new interactive file:

```
ExtraTerminal: INTERACTIVE
```

There are no inherent limits to the size of a file, but the storage device on which a file resides will be limited in size.

In UCSD Pascal, the restriction against a **file of file** (or TEXT or INTERACTIVE) is a special case of a general restriction against files being declared within *any* structured type. A file cannot be an element of an array, or a field within a record. In standard Pascal, this would be allowed. The chief reason for this restriction is so that the compiler can easily generate code to automatically close files when a program (or routine) terminates.

Standard File Handling

The following pages describe the file-handling routines that are part of standard Pascal: EOF, EOLN, PUT, GET, RESET, and REWRITE. The RESET and REWRITE routines can also be used to associate internal files with external files. This use is nonstandard, and is described in Chapter 7.

Files are frequently accessed by using the standard routines READ, READLN, WRITE, and WRITELN. These routines are described in Chapter 7.

EOF

The EOF (FileName) routine returns a Boolean value that is true if the window variable has been moved *beyond* the end of file (EOF stands for end of file). When EOF is TRUE, the value of the window variable is undefined.

If the FileName parameter is absent, the file INPUT is assumed.

If the file is INTERACTIVE, EOF becomes TRUE when the CTRL-C keys have been pressed.

If the file is a p-System textfile, EOF=TRUE implies EOLN=TRUE.

WARNING

> **If EOF (FileName) becomes TRUE during a call to GET (FileName), READ (FileName, ...), or READLN (FileName, ...), the data obtained from the get or read is not valid and must not be used.**

EOLN

The EOLN (FileName) routine is defined only if the file is of type TEXT or INTERACTIVE. It returns a Boolean value that is TRUE if a <return> character (ASCII CR, CHR(13)) has just been read (EOLN stands for end of line). The window variable contains a blank (' ').

If the FileName parameter is absent, the file INPUT is assumed.

If EOF is TRUE, EOLN is TRUE.

PUT

The PUT (FileName) routine writes the value of the window variable to the file FileName.

The FileName parameter *must* be included.

In standard Pascal, PUT is only defined if EOF is TRUE, in other words, one can only write to a file by appending values to its end.

In UCSD Pascal, PUT can be performed anywhere in the file. If EOF is indeed TRUE, the value of the window variable is appended. If EOF is FALSE, the value of the current record in the file is replaced by the value of the window variable. In other words, UCSD Pascal allows random-access updates of file records; see the description of SEEK in Chapter 7.

If PUT is used to append an element to the file, and there is no more room on disk, a runtime error occurs.

GET

The GET (FileName) routine is defined only if EOF = FALSE. It advances the file window by one file record; the value of the window variable is replaced by the value of the next element in the file sequence. If there is no following record, EOF is set to TRUE, and the value of the window variable is undefined.

The FileName parameter *must* be included.

RESET

The RESET (FileName) routine resets the file window to the beginning of the file. If the file is not empty, EOF is set to FALSE, and the window variable is set to the value of the first record in the file.

The FileName parameter *must* be included.

If the file is empty, EOF remains TRUE, and the value of the window variable is undefined.

If the file is INTERACTIVE, EOF is set to FALSE, but the value of the window variable remains undefined. The reason for this is that the INTERACTIVE file might well be an input device or an input/output device, and not have any pending data. Trying to define the value of the window variable before the user has typed in any data would be difficult!

The RESET routine in UCSD Pascal can be called with an optional second parameter that defines an external p-System file to be associated with the internal FileName. This use of RESET is described in Chapter 7.

REWRITE

The REWRITE (FileName) routine clears the file FileName by creating a temporary file that is empty. The EOF routine is set to TRUE.

The FileName parameter *must* be included.

The temporary file can either replace the original file, or be discarded. This disposition depends on the way the file is closed (see Chapter 7).

As with RESET, in UCSD Pascal REWRITE can be called with an optional second parameter that associates an external file with FileName. This use of REWRITE is defined in Chapter 7.

Pointers

Dynamic variables are allocated memory at runtime, rather than compile time. A (nameless) variable of a given type can be created by declaring a pointer to that type, and then allocating space to the variable by calling the intrinsic NEW. Once the variable has been created, it can be referenced by pointers.

Pointers are used to build data structures such as linked lists and trees. For this purpose, they are typically embedded in record structures that contain both information and pointers to other records of similar type.

Pointer Format

A pointer is declared in the following way:

```
<pointer type name> =∧ <type identifier>
```

such as:

```
PoolPtr =∧ STRING;
```

The pointer is said to be *bound* to the type named by < type identifier> .

If we had variables of type PoolPtr:

```
var PoolP, NewPoolP: PoolPtr;
```

we could then reference a string indirectly:

```
PoolP∧ := 'isn''t this a nice test string?' ;
```

In a Pascal program, the <pointer type name> by itself refers to the pointer itself; the <pointer type name> followed by a caret (∧) refers to the object that is pointed to.

For example, if we made the following assignment:

```
NewPoolP:=PoolP;
```

then the following expression would be TRUE:

```
NewPoolP∧= 'isn''t this a nice test string?' ;
  {= PoolP∧}
```

The predeclared identifier NIL is a pointer to nothing. It is usually used to signal the end of a linked list. For example:

```
PoolP:= NIL;
```

At this point, PoolP∧ would be undefined; a program which tried to access PoolP∧would abort with a runtime error.

In standard Pascal, NIL is a reserved word, not a predeclared identifier.

A pointer can be assigned the value of another pointer provided *both* are pointers to the same data type.

Pointers can be compared with = and <>. There are no operations on pointers.

Building Data Structures

As we stated before, pointers are most frequently used in conjunction with record structures that carry other information as well. The following paragraphs illustrate the use of pointers to construct some familiar data structures.

Given the following declarations, we could construct a linked list of names:

```
type
      arrow = ∧ NameRec;
      NameRec = record
                  name:   STRING[30];
                  next:   arrow;
                    end;
var
      head: arrow;
```

The beginning of the list is pointed to by the variable head, and the pointer field (next) in the last record of the list is equal to NIL.

Note that arrow was declared as a ∧ NameRec before NameRec itself was declared. This is legal as long as both the pointer and the type it references are declared in the same **type** declaration part.

Note also that we did not declare any variables of type NameRec. We are able to build the list by allocating space that is pointed at by head and by subsequent next fields. This is done with the intrinsic NEW which is described later in this chapter.

If the next field of the last record is changed to equal head ∧ instead of NIL, we can turn our linear list into a circular queue.

By changing the declaration of NameRec, we can make our list doubly-linked provided the pointers are initialized correctly:

```
NameRec = record
              name: STRING[30];
              fwd, back: arrow;
          end;
```

In the first record of this list (head∧), forward points to a new record, and back=NIL. In the last record of this list, back points to the previous record, and forward equals either NIL or head∧, depending on whether the list is linear or circular.

Here are the declarations for a weighted binary tree:

```
type
    edge = ∧ node;
    node = record
               Weight: INTEGER;
               LLink, RLink: edge;
          end;
var
    Root: edge;
```

In this example, the pointer called Root would be the base of the tree, and in each node, LLink would point to the left-hand subtree, and RLink to the right-hand subtree.

These small examples should give the reader some insight into how similar (and more complex) data structures would be declared. The following section explains how to allocate a new data object to a pointer by using the intrinsic NEW. For a program example that uses pointers, refer to Part 2, Chapter 2.

Standard Memory Management

The following pages explain how to allocate and deallocate memory by using the standard procedures NEW and DISPOSE. UCSD Pascal provides some further memory-management intrinsics that are more powerful, but more error-prone. These are described in Chapter 8.

NEW

The NEW (POINTER) intrinsic allocates space for a variable of the type to which POINTER is bound, and sets the value of POINTER equal to the location of that variable.

If the bound type is a record with variants, NEW allocates enough space for the largest possible record of that type. This can be avoided by the following form of a call to NEW:

```
NEW(POINTER, TAG1, TAG2, ... , TAGn);
```

where:

TAG1 . . TAGn are the tag fields for particular variants.

If there are more variants than tag fields specified, the remaining variants are allocated the maximum necessary space.

WARNING

If a record has been allocated with a particular variant, and the program accesses the record using a *larger* variant, the program will encounter incorrect data. In UCSD Pascal, there is no runtime check to protect against this.

Note that a call to NEW with tag fields does *not* initialize the record, it merely specifies which variants will be used. The record must still be initialized by a READ or an assignment to POINTER∧.

DISPOSE

The DISPOSE (POINTER) intrinsic deallocates the variable POINTER∧, and sets POINTER equal to NIL.

If the program does not use much data space, then DISPOSE is probably unnecessary, but if the program's data occupies a lot of main memory, the use of DISPOSE, along with some other memory management scheme, can be required (see Chapter 8 and Part 2, Chapter 7).

The DISPOSE intrinsic can be called with case variant tags, just as NEW. If fact, If NEW is called with particular variant tags, then DISPOSE must be called with the *same* variant tags; otherwise, the heap can be damaged and a runtime disaster (as opposed to an error) may result (see Chapter 8).

SPACE ALLOCATION FOR DATA TYPES

The following paragraphs outline the space allocated to Pascal data types, and their overall format. All information here applies to UCSD Pascal, and not necessarily to any other implementation of Pascal. Finer details are not dealt with; programmers who need more information on System internals should refer to the *UCSD p-System Internal Architecture Guide.*

Packed Data

The declaration of an array or record can be preceded by the reserved word **packed**, for example:

```
TITLE: packed array [0..89] of CHAR;
```

Declaring an array or record as **packed** does not alter the semantics of a program, it merely alters the way in which the data is stored.

Packed data is stored as tightly as possible. A single data item is never split across a word boundary, and an array or record within a **packed** array or record always begins on a word boundary.

Here are some examples:

```
TITLE: packed array [0..89] of CHAR;
```

The previous example occupies 45 words, rather than 90. Characters are packed 2 per word.

```
SAMPLE: packed array [0..9] of
            packed array [0..9] of 0..3;
```

This example occupies 20 words, rather than 100. Each element of 0..3 requires only 2 bits, so each row of the matrix is 2 words long; 8 elements in the first word and 2 in the next, followed by blank space.

```
OneWord:  packed record
          choice1, choice2, choice3, choice4,
          choice5, choice6, choice7, choice8:
             BOOLEAN;
          initial: CHAR;
       end;
```

This example occupies 1 word, rather than 9. The 8 Boolean bits are packed into one byte, and the character occupies one byte.

If an element in a packed array or record is too large to be effectively packed, the word **packed** is ignored. For example:

```
packed array [1..5] of INTEGER;
```

Since an integer must occupy 16 bits, the example contains 5 words, just as it would if the word **packed** were not present.

If a field in a packed record happens to be allocated a full word (for example, the next field had to be word-aligned), then the field is unpacked and occupies the whole word.

A packed variable cannot be compared to an unpacked variable, even if the underlying type is identical.

An element of a packed array or record can never be passed as a **var** parameter, but can be passed as a value parameter.

If a packed record contains a case variant, enough space is allocated for the largest possible variant.

In an array declaration, the appearance of the reserved word **array** *without* the word **packed** in effect turns packing off. If the entire array must be packed, it is safest to use **packed** in front of every instance of the reserved word **array**.

Standard Pascal defines the intrinsic procedures PACK and UNPACK. Since packing and unpacking are done automatically in the UCSD p-System, there is no need for these procedures, and they are not implemented.

Simple Types

Integer

Integers are stored in 16-bit words (two 8-bit bytes) in two's complement format. The constant −32768 is illegal, and cannot be compiled.

When an operation on integers results in an overflow, a runtime error occurs. This can sometimes be avoided by restructuring the expression, for example:

big1 * big2 **div** big3	Causes an overflow
big2 **div** big3 * big1	Does not overflow

Integers are packed one per word.

Real

Real numbers are stored in either 2-word (32-bit) or 4-word (64-bit) formats. If the 2-word format is used, constants generated by the compiler are also 2 words long. If the 4-word format is used, constants generated by the compiler are from 4 to 6 words long. When a code segment is loaded into memory, real constants are converted to the standard real format for the 8088 processor.

When an integer is compared to a real constant, the integer is converted to a real format without changing its value.

Since integer values can appear in an expression whose result is real, it is possible for an integer overflow to occur while evaluating a real expression. This can be avoided by reordering the expression.

As with integers, real values are packed at their actual size (two or four words).

Long Integer

A long integer is always allocated an integral number of words.

As with real expressions, it is possible for an integer overflow to occur while evaluating an expression whose result is a long integer.

Long integers are packed at their actual size.

Boolean

Boolean values are stored in a 16-bit word; a 0 is FALSE and a 1 is TRUE.

The intrinsic function ODD does not generate code to convert an integer to a Boolean; it merely allows an integer value to be treated as a Boolean. The low-order bit of the integer indicates whether it is odd or not; the other 15 bits of the word are left unchanged. See Part 2, Chapter 8.

A Boolean value is packed into one bit.

Character

A character is stored as an integer in the range 0..255 (using the ASCII character set). An unpacked character occupies the low-order byte of a full word.

ASCII codes are only seven bits long. When ASCII characters are used in I/O to or from a character-oriented device, it is possible that the system will strip the characters' high-order bit.

The intrinsic function CHR does not emit code to change the contents of the integer value it is passed; it merely allows that value to be treated as a character.

A character value is packed into a single byte.

Scalars and Subranges

Scalar values are represented as integers; the first value equals 0, the next 1, and so forth.

Values in a subrange are represented as the values of their (scalar) base type; only the boundary conditions for range-checking are altered.

The intrinsic function ORD does not emit code to change the contents of the scalar value it is passed; it merely allows that value to be treated as an integer.

When scalar and subrange values are packed, each value is allocated the minimum number of bits necessary to store the *maximum* value of the type.

Structured Types

Arrays

The space allocated to an array depends on the space needed to store its base type. All array entries are aligned on word boundaries, unless the array is **packed**. Elements in an array are stored in row-major order; a single row is stored in sequence, followed by the next row, and so forth. This is opposed to storing the array by columns, which is standard for some other languages.

Arrays are packed according to their base type. A **packed array of** CHAR contains two 1-byte characters per word, and so forth.

Strings

A string is allocated enough words to contain its static (maximum) length. The first byte contains the string's dynamic length (0..255). The rest of the string consists of 1-byte characters packed two per word.

Since strings are already packed, packing does nothing.

Sets

Each value that can be present in a set is represented by a single bit. The location of the bit corresponds to its ordinal value; bits are numbered from zero, starting at the left. Zeroed bits are padded on the right of the used bits, so that each set occupies an integral number of words. The length of a set is not normally allocated with the set, but it is sometimes loaded with the set at runtime.

A set can contain at most 4,080 elements (255*16=4,080), and 4,079 is the highest possible ordinal value in a set. The set [4,078 .. 4,079] is a full-sized set, *not* a set of one word.

Sets are already packed, so packing does nothing.

Records

Unless a record is packed, each field begins on a word boundary. The space allocated to each field depends on the type of the field.

Fields of a record are packed according to their type, but never cross word boundaries.

Dynamic Types

Packing does nothing to dynamic types.

Files

Files are created as p-System disk files. If the program does not explicitly associate a Pascal file with an existing (or newly created) disk file, and does not use CLOSE to save that file, a temporary disk file is created that is removed from the directory when the scope of the file's declaration is exited.

For a typed file, the system allocates a 1-block (512-byte) area of memory as a buffer. No buffer is allocated for untyped files.

More details about disk files and disk directories can be found in the *UCSD p-System Operating System Reference Manual* and *UCSD p-System Internal Architecture Guide,* respectively.

Pointers

A pointer is an address of a physical location. Since data items always begin on word boundaries, a pointer is always a word pointer.

The SIZEOF Intrinsic

The SIZEOF (name) intrinsic, where name is the identifier of *any* variable or data type, returns an integer that is the number of bytes that have been allocated to that variable or data type.

The SIZEOF intrinsic does not appear in standard Pascal.

WARNING

The SIZEOF (ptr∧) intrinsic does not give the size of the object pointed to, but the size of ptr (always one word). To find the size of the object pointed to, call SIZEOF with the name of the object's type.

4

Overall Program Syntax

This chapter describes several topics that apply to the overall syntax of a program. First it covers a program's outline and declarations, then the scope of identifiers, and various restrictions. It ends with a discussion of **units** and separate compilation.

THE OUTLINE OF A PROGRAM

A Pascal program can be outlined in the following form (using EBNF):

```
"program" program-name
              [ "  (" identifier { "," identifier } ")" ] ";"

[ label-declarations ]
[ constant-declarations ]
[ type-declarations ]
[ variable-declarations ]
[ routine-declarations ]

"begin"
   [ statement { ";" statement } ]
"end" "."
```

Every identifier that is used in a Pascal program must be declared before it is used (predeclared identifiers are declared global to the entire program). Declarations must appear in the order they are shown. A particular kind of declaration does not need to appear if it is not used (for example, a program can use variables, but no constants; only the variable declarations need be present).

UCSD Pascal allows one exception to the order of declarations. One or more include files that contain only declarations (in their proper order) can appear *after* any **var** or **forward** routine declarations, and *before* any bodies of code. For more details, see Chapter 10.

Program Heading

The first line in a Pascal program consists of the reserved word **program**, followed by an identifier and a semicolon. The following is an EBNF description:

```
"program" program-name
              [ "(" identifier { "," identifier } ")" ] ";"
```

For example:

```
program sample;
```

A list of identifiers can follow the program name, for example:

```
program sample(INPUT, OUTPUT);
```

This feature is provided for compatibility with standard Pascal. In UCSD Pascal, these identifiers can be present, but they are ignored.

Label Declarations

In Pascal, a label is an integer in the range 0..9999. Any labels that a program uses must be declared in a label declaration with the following scheme (using EBNF):

```
"label" [ label { "," label } ] ";"
```

such as:

```
label 1, 13;
```

A label is a way of tagging a statement so that the statement can be jumped to by a **goto** statement elsewhere in the program. The use of labels and **gotos** is described in Chapter 6. Every label that is declared must be referenced by at least one **goto** statement.

Constant Declarations

A constant declaration associates an identifier with a value; this value does not change throughout the program. Constants are declared according to the following scheme (using EBNF):

```
"const" { identifier "=" value ";" }
```

For example:

```
const
  file_number = 77;
  ListKey = 0;
  inv_fnum = -file_number;
  KEYLETTER = 'Y';
```

The value of a constant must be a numeric constant, a Boolean, a character, or a string.

The length of a string constant is fixed; it can be assigned to any string variable, or to a **packed array of** CHAR that has the same length.

As shown, a numeric constant can be declared as the opposite (−) of a numeric constant that has already been declared. Other than this, no expressions or previously defined constants are allowed as values in constant declarations.

Type Declarations

Type declarations are similar to constant declarations. They associate an identifier with a description of a type. Here is a description in EBNF:

```
"type" { identifier "=" type-description ";" }
```

For example:

```
type
  directions = (north, south, east, west);
  link = ^tree;
  tree = record
              height, girth: INTEGER;
              locn: directions;
              next: link
      end;
  bulk = file of INTEGER;
```

Variable Declarations

A variable declaration associates an identifier with a type. In EBNF, this is the scheme of a variable declaration:

```
"var" { identifier-list ":" type ";" }
```

The type can either be the name of a type, or a type description (which need not have appeared in the type declarations).

For example:

```
var
    compass: directions;
    i, j, k, l: INTEGER;
    judgement: (bad, so_so, good);
    in_file: bulk
```

Routine Declarations

The details of routine declarations (that is, declarations of procedures, functions, or processes) appear in Chapter 5. The scope of identifiers is affected by routine declarations, and is described in the following paragraphs. Some restrictions on routine declarations are also described in this chapter.

In general, the format of a routine declaration is similar to the format of a program. It contains a heading, followed by declarations, and a set of statements enclosed by the reserved words **begin** and **end**.

A routine heading consists of **procedure**, **function**, or **process** followed by an identifier, followed by an optional list of parameters and a semicolon. The heading of a **function** also contains the type of the value that is returned by the function.

The following are some routine headings:

- **procedure** bean;
- **function** maximum(i,j: INTEGER): INTEGER;
- **process** cold;
- **procedure** greater (a,b: REAL; **var** c: REAL);

The Main Body

The main body of a program consists of the reserved word **begin**, followed by a list of statements separated by semicolons, the reserved word **end**, and a period (.).

Every Pascal program must have a main body.

STRUCTURE AND SCOPE

Every routine in a Pascal program, and the Pascal program itself, is said to be a *block*. A block consists of a heading (such as a program heading or procedure declaration), a list of declarations (if used), and a list of statements surrounded by the reserved words **begin** and **end**.

This structure allows blocks to be nested. A program can contain blocks in the form of routines, each routine can contain blocks in the form of nested routines, and so forth.

The nesting of blocks governs the scope of variables and routines.

Scope

An item declared within a block is said to be *local* to that block. An item that has been declared in the block that contains a local block is said to be *global* to that local block.

If two procedures are declared in a Pascal program in the following way:

```
procedure initialize;
    . . .;

procedure terminate;
    . . .;
```

then each of them is an independent block, and neither is local to the other (both of them are local to the program in which they are declared).

On the other hand, if two procedures are nested, then the nested procedure belongs to its global procedure:

```
procedure scan;
    . . .
    procedure get_char;
        . . .
        begin {get_char}. . .
        end; {get_char}
    begin {scan}. . .
    end; {scan}
```

In this example, get_char is local to scan, and scan is global to get_char.

Any item (variable, type, routine, and so forth) that is local to a block is invisible to all of a program except that block itself and any blocks it may contain.

Thus, a procedure (or function, or process) can call any procedures that are global to it, or any procedures that are declared within it, but it cannot call procedures that are local to other blocks in the program.

These rules apply to all identifiers. Identifiers declared at the program level can be used anywhere in the program; identifiers delcared local to a routine can be used only within that routine and its nested routines, and so forth. Predeclared identifiers are considered global to the entire program.

A global identifier can be redefined within a subordinate block; the local meaning supplants the global one. For example:

```
program credits;
    . . .
    var COUNT: INTEGER;
    . . .
    procedure DEBITS;
        . . .
        var COUNT: REAL;
        . . .;
    begin ... end.
```

The variable COUNT that is local to DEBITS is a different variable than the variable COUNT that is global to the program. Note that it does not need to be of the same type; in this example, it is not. Within the procedure DEBITS, COUNT would always refer to the local variable; the global COUNT could not be used.

Predeclared identifiers can be redefined just as user-declared identifiers. The programmer should be cautious about doing this, since it impairs a program's readability.

A routine cannot call a routine whose declaration follows it in the source program. This restriction can be circumvented by using the reserved word **forward**, as described in the following paragraph.

RESTRICTIONS

Forward Declarations

It is sometimes necessary for two routines to call each other. It is also sometimes desirable, in order to make a program more readable, to list routine declarations in one place, and the body of the routine code elsewhere. These situations can be handled by use of **forward** declarations.

A routine can be declared with the reserved word **forward** in place of its block (its declarations and statements). The routine(s) that must call this routine (and be called by it) can then be declared, followed by a second declaration of the routine along with its full body.

If the routine has parameters, they must be specified in the **forward** declaration, and not in the second declaration.

Suppose the procedures grab1 and grab2 must call each other. The following declarations would allow this:

```
procedure grab1 (x: INTEGER); forward;

procedure grab2 (x: INTEGER);
    {declarations and body of grab2};

procedure grab1;
    {declarations and body of grab1};
```

Note that the parameters to grab1 are declared in the *first* declaration of grab1. They must not be included in the second declaration of grab1.

Size Limits

These limits apply only to UCSD Pascal.

A routine can contain no more than 16,383 words of local variable space.

A given segment (see the following subsection, also Chapter 8) can contain up to 255 routines.

Segment Routines

A code segment is normally the object code (P-code) produced by a single compilation. A routine can be made to occupy a code segment of its own by preceding its heading with the reserved word **segment**. Code segments and segment routines are discussed in more detail in Chapter 8.

Within a given code segment, the code bodies of all segment routines must precede the code bodies of all non-segment routines.

If a segment routine must call a non-segment routine, the non-segment routine must be declared by a **forward** declaration that precedes the segment routine's declaration.

UNITS AND SEPARATE COMPILATION

Portions of a program in UCSD Pascal can be separately compiled, and packaged into a **unit** that can be used by programs or by other **units**.

The UCSD Pascal Compiler will compile:

- A single Pascal program
- A single unit
- A number of units (separated by semicolons)
- A Pascal program with in-line units

A **unit** is not a complete program. Instead, it consists of an **interface** part, whose declarations can be used by code that **uses** the unit, and an **implementation** part, whose declarations and code are private to the unit. It can also contain initialization and termination code.

A program or other unit can use the objects declared in a unit's **interface** part by naming the unit in a **uses** declaration.

When a unit is declared within a program, it must appear after the program heading and before any declarations. A number of units can be declared in this way, each of them followed by a semicolon. This is a simple way to compile units, but it defeats the advantages of separate compilation.

The Uses Declaration

All units that a program uses must be declared in **uses** declarations. All of these declarations must follow the program heading, and precede any other declarations (such as **label**, **const**, and so forth).

For example:

```
program NotInsane;
    uses smallplot, Record_Ops;
    . . .
```

It is possible for a unit to use other units. Within a unit, uses declarations immediately follow either the reserved word **interface** or the reserved word **implementation**.

Within a uses declaration, the name of a unit can be followed by a list of identifiers (enclosed in parentheses). The identifiers are names of objects declared in the unit's **interface** part; that is, the identifiers can be constants, types, variables, or routines.

For example:

```
program theory;
    uses Turtlegraphics(Move,Turn,PenMode,PenColor),
        Record_Ops;
```

This construct is called a selective **uses**.

Note that the parameters of each routine do not need to be listed; the routine's name is enough. This also applies to scalar (or subrange) types and records. Listing the name of the type (or variable) in the selective uses declaration will give the program that uses the unit access to the names of the constants within a scalar, the field names within a record, and so forth.

CAUTION

If a routine that is named in a selective uses declaration depends on a type or variable that is also declared in the unit's interface part, that type or variable must also be named in the uses declaration.

The advantage of a selective uses declaration is that it can save memory space during compile time; only those identifiers that are selected need appear in the program's symbol table. The selective uses declaration is especially useful if the unit has a very large **interface** part. It is also a good form of documentation, since it explicitly identifies those portions of a unit that the host program requires.

The Format of a Unit

A **unit** has the following scheme (using EBNF):

```
"unit" identifier ";"

   "interface"
      [ uses-declarations ]
      [ constant-declarations ]
      [ type-declarations ]
      [ variable-declarations ]
      [ procedure-and-function-headings ]

   "implementation"
      [ uses-declarations ]
      [ label-declarations ]
      [ constant-declarations ]
      [ type-declarations ]
      [ variable-declarations ]
      [ procedure-and-function-declarations ]

   [ "begin"
      [ initialization-statements ]
      [ "***;"
        termination-statements ]
   ] "end"
```

The data structures and routines declared in a unit's **interface** part are global to the program or unit that **uses** them; the data structures and routines declared in the **implementation** part are strictly private to the unit, and may not be used by any other code. This allows the programmer to separate the unit's interface (whose form is determined by the problem to which the unit is applied) from the implementation (which can vary from system to system or be improved over time).

When a unit is changed, it must be recompiled. If its interface section is *not* changed, the programs that use it do not need to be recompiled. If the interface section *is* changed, then all programs that use the unit must be recompiled as well.

Restrictions

When a procedure or function is to be part of a unit's interface, its heading must appear in the interface part. This heading must include all of the procedure's or function's parameters. The actual code for the procedure must appear in the unit's implementation part; the procedure or function heading must reappear *without* any parameters. This scheme is analogous to the declaration of forward routines, although the word **forward** is not used.

Segment procedures and functions must be declared as such in the implementation part; the reserved word **segment** must not appear in the interface part. The same restriction applies to procedures and functions that are declared **external**.

An interface part cannot contain an include file (see the string option $I, described in Chapter 10).

An include file can contain *all* of an interface part, followed by the reserved word **implementation**, and, possibly, the rest of the implementation part as well. An interface part is treated by the compiler as if it were a single include file.

If **uses** declarations appear within the **interface** part of a unit, they must be ordered correctly; if unit first__level **uses** unit second__level, the interface part must specify:

```
uses second__level, first__level;
```

to avoid a compilation error.

Initialization and Termination Code

The *main body* of a unit can consist of a single **end**, or a compound statement (with a **begin end** pair).

The compound statement can contain the construct:

```
***;
```

This pseudo-statement is used to separate initialization code from termination code. All statements that precede ***; are initialization statements, and all statements that follow ***; are termination statements. If no ***; is present, the entire compound statement is understood as initialization code; if the entire compound statement is intended to be termination code, ***; must be the first statement to follow the **begin**.

Initialization code is executed *before* any code of the host that uses the unit (whether that host is a program or other unit).

Termination code is executed *after* the host's code has terminated.

The compound statement should not contain branches around ***;: branching past it from initialization to termination code has the effect of executing the termination code and skipping the host's code entirely, while branching from termination to initialization code has the effect of re-executing the initialization code and locking the host's execution in an endless loop!

If an EXIT (**program**) call appears in initialization code, the remainder of initialization code is skipped, and so is the host's code. Execution proceeds with the unit's termination code. If an EXIT(**program**) appears in termination code, the remainder of the termination code is skipped (this applies only to the unit in which the EXIT appears; if the termination code of other units is pending, it will still be executed). The call EXIT (< unit name>) is not legal.

The syntax ***; was adapted from the Pascal dialect called Pascal Plus, by Welsh and Bustard.

The Implementation of Units

When a program uses a unit, the code for that unit must be either within the program itself, the file *SYSTEM.LIBRARY, a user library file, or the operating system. If the system can find the unit's code, it is automatically linked to the host's code.

If the unit is in a user library file, this file must be named with the $U compiler option (see Chapter 10). Library files are described in the *UCSD p-System Operating System Reference Manual.*

5

Procedures and Functions

Routines are distinguished by the manner in which they are called. A **procedure** is called by a statement that simply consists of the procedure's name (and possibly a parameter list). A **function** is called from within an expression. A **process** is initiated by a call to the intrinsic procedure START.

This chapter describes procedures and functions. Processes are described in Chapter 9, which deals with concurrency.

PROCEDURE AND FUNCTION DECLARATIONS

In the previous chapter, we saw some examples of procedure and function headings. They can be described by the following EBNF expressions:

```
"procedure" identifier [
                    "(" ["var"] id-list ":" type-id
                         { ";" ["var"] id-list ":" type-id }
                    ")" ] ";"

"function" identifier [
                    "(" ["var"] id-list ":" type-id
                         { ";" ["var"] id-list ":" type-id }
                    ")" ] ":" type-id ";"
```

The list of parameters to a procedure or function is optional. Parameters are a means by which a procedure or function can communicate with other portions of a program.

The parameters declared in the routine heading are known as *formal* parameters. When a routine is called, the program specifies the values and variables that the routine should actually use; these are known as *actual* parameters.

The following is an example:

```
procedure max (a,b: INTEGER; var Result: INTEGER);
    {determines the maximum of two integers}
    begin
        if a > = b then
            Result:= a
        else
                Result:= b;
    end {max};
```

The preceding example could be called by the following statements:

```
max(bound1, bound2, compare);
. . .
max(7,346,maxval);
```

Note that the declaration of the formal parameter Result is preceded by the reserved word **var**. This allows a value to be assigned to Result, which changes the value of the actual parameters used (in this example, compare and maxval). The **var** parameters are discussed in more detail later in this chapter.

We could use a function to do the same calculation:

```
function max (a,b: INTEGER): INTEGER;
  {finds the maximum of two integers and returns it}
  begin
    if a > = b then
      max:= a
    else
      max:= b;
  end {max};
```

Note in this example that the code is essentially the same, but the result is assigned to the function name, rather than a parameter. This is how a function's result is returned. The type of the value assigned to the function name must be assignment-compatible with the function's type, which is declared in the function heading (for compatibility rules, see Chapter 3).

The function could be called in the following ways:

```
higher:= max(threshold, 100);
. . .
result:= 134 + max(term1,term2)*2)/reduce;
. . .
if year <> max(date1, date2) then . . .
```

In the routine heading, a parameter's type must be specified by a type identifier (type-id in the EBNF), and not a type description.

Value Parameters

A *value parameter* is declared in a routine heading by including the name of the parameter and its type.

Value parameters are used as local values for the purpose of local calculations. If a value parameter is changed within a routine, only the local copy of the parameter changes.

An actual parameter that corresponds to a value parameter may be either a constant or a variable, provided it is of the correct type. If the actual parameter is a variable, its value is not affected by the called routine.

Variable Parameters

A *variable parameter* or **var** parameter is declared in the same way as a value parameter, but the declaration is preceded by the reserved word **var**.

Variable parameters are used to pass the results of calculations from the routine (usually a procedure) to its caller. If, within the routine, a value is assigned to the formal name of a variable parameter, the value of the actual parameter also changes. (This form of parameter passing is also called *call-by-reference*.)

An actual parameter that corresponds to a variable parameter must be a variable of the proper type. The programmer should assume that its value will by changed by the call to the routine.

The declarations of value and variable parameters can be lumped together, for example:

```
procedure many__parms (a, b, c, d: REAL;
                       var e, f, g, h, i: REAL;
                       j, k, l, m, n: REAL);
```

The first and last groups of parameters declare value parameters, and the middle group declares variable parameters. The **var** declaration applies to the declarations that immediately follow it; its influence extends to the next semicolon (;).

Constants, elements of a **packed array**, fields of a **packed record**, and **for** loop counters cannot be passed as **var** parameters.

A file can *only* be passed as a **var** parameter.

Range-checking is never performed on a STRING passed as a **var** parameter.

Variable parameters, unlike value parameters, are not copied into a routine's local area. When very large variables (such as arrays) must be passed to a routine, memory space can be saved by passing them as **var** parameters, even when the routine is not meant to change their contents.

Procedures and Functions as Parameters

In UCSD Pascal, a procedure or function *cannot* be passed as a parameter to another routine. This is allowed in standard Pascal.

CALLING PROCEDURES AND FUNCTIONS

As already illustrated, a procedure is called by a single statement, and a function is called by using it within an expression.

Procedures and functions can call themselves recursively. This is illustrated in Part 2, Chapter 3. It is important that recursive routines be written so that they will eventually terminate; the calls must eventually reach a case where the procedure or function does not call itself. There is no way for the system to check this, so the programmer must use caution. A *runaway* recursive routine causes the program to halt with a stack overflow runtime error.

When the code of a procedure or function has finished executing, the procedure or function returns to the caller. Execution proceeds from the point at which the procedure or function was called.

The user can cause a routine to return immediately, from anywhere in its code body, by calling the UCSD intrinsic procedure EXIT, which is described in Chapter 6.

A function must contain at least one statement that assigns a value to the name of the function, and at least one of these statements must be executed. There is no runtime checking to see whether this is done; it is the programmer's responsibility.

In a procedure or function call, the order of actual parameters, their number, and their type must correspond exactly to the order, number, and type of formal parameters.

Certain Pascal intrinsics have optional parameters that may be omitted when the intrinsic is called, but this is a peculiarity of intrinsic routines, and there is no way for a user-written routine to specify optional parameters.

6

Control Statements

This chapter discusses the statements that can be used to direct a program's sequence of execution (its *flow of control*).

COMPOUND STATEMENTS

A group of statements can be made to appear as a single statement by surrounding it with the reserved words **begin** and **end**. Such a group is called a *compound statement*. Here is an EBNF description:

```
compound-statement =
  "begin" statement { ";" statement } "end"
```

As the expression shows, a semicolon serves to separate statements rather than terminate them. Nonetheless, it is common practice to put a semicolon after the last statement in the statement list (it is considered to be followed by a null statement). This simply makes it easier to add statements to the statement list while debugging or maintaining the program. Most of the code samples in this book show this optional semicolon.

These are compound statements:

```
begin
    initialize;
    calculate;
    print_report;
    terminate;
end {outer_proc}

begin
    sum:= odd_int(sum,pile[i]);
    all:= all+1; choice[all]:=i;
    if pile[i] = 1 then ones:= ones+1
    else
        if pile[i] > 1 then p:= i;
end
```

A compound statement can appear anywhere a single statement can.

CONDITIONAL STATEMENTS

Pascal provides two forms of decision-making statements; the **if** statement, and the **case** statement.

The If Statement

An **if** statement has two forms. The following is an EBNF description:

```
if-statement =
    "if" Boolean-expression "then" statement
    |
    "if" Boolean-expression "then"
      statement
    "else"
      statement
```

If the Boolean expression is TRUE, then the statement that follows the reserved word **then** is executed. If the **if** statement has an **else** part and the Boolean expression is FALSE, then the statement that follows the reserved word **else** is executed.

In the **if then else** form of the **if** statement, a semicolon *must not* precede the reserved word **else**, since the semicolon indicates the end of the entire **if** statement.

The **if** statements are often nested, and this can appear to be ambiguous:

```
if < Boolean expr1> then
    if < Boolean expr2> then < statement1>
    else < statement2> ;
```

Does the **else** pair with the first **if** or the second **if**? In Pascal, the rule is that an **else** pairs with the *closest* matching **if**. Thus, this example is indented correctly. An equivalent statement would be:

```
if < Boolean expr1>  then
     begin
          if < Boolean expr2>  then < statement1>
          else < statement2>
     end;
```

If the outer **if** statement has an **else** part but the nested **if** does not, then **begin** and **end** should be used:

```
if < Boolean expr1>  then
     begin
          if < Boolean expr2>  then < statement1>
     end
else < statement2> ;
```

or the nested **if** should use a null **else** part:

```
if < Boolean expr1>  then
     if < Boolean expr2>  then < statement1>
     else < do nothing>
else < statement2> ;
```

The following examples contain **if** statements:

```
if found
     then Classify:= sym_tab[m].kind
     else Classify:= symbol;

if sym_name = sym_tab[m].name
     then found:= TRUE
else if sym_name <  sym_tab[m].name
     then r:= m-1
else if sym_name >  sym_tab[m].name
     then l:= m+1;
```

It is common practice to indent long sequences of **else ifs** so that they line up vertically to make them easier to read, and emphasize the fact that they represent a series of cases of which only one will be executed.

The Case Statement

A **case** statement has this form (using EBNF):

```
case-statement =
  "case" expression "of"
    constant-list ":" statement
    { ";" constant-list ":" statement } [";"]
  "end"
```

Note that the reserved word **end** pairs with the reserved word **case** (just as an **end** pairs with **record**); there is no **begin**.

The expression must evaluate to a scalar (or subrange) value. Each constant-list consists of one or more constants of the same type as the expression (if there is more than one constant, they are separated by commas). No constant can appear in more than one constant list.

If the value of the expression matches a constant in one of the constant lists, the statement that follows that constant list is executed.

The following is an example of a **case** statement:

```
READ(letter);
case letter of
    'A': Assemble;
    'C': Compile;
    'D': Debug;
    'E': Edit;
    'F': File;
    'H': HALT;
    'I': Re_Init;
    'L': Link;
    'M': monitor:= TRUE;
    'R': begin
            codefile:= '* system.wrk.text';
            Execute;
        end;
    'U': Execute;
    'X': begin get_codefile; Execute end;
    'end' {case}
```

We hasten to point out that this is merely an illustration, and not a portion of the actual operating system! Similar code is used to handle single-character commands at the outer command level.

In UCSD Pascal, if the case expression does not evaluate to one of the given options (for instance, in the example, not all 26 letters are shown), then the **case** statement is bypassed, and the next statement in sequence is executed. The **case** is said to fall through. This is contrary to standard Pascal, in which the result of a **case** is undefined if a selected option is not present.

There must be at least one statement preceded by a constant list. This is a restriction of UCSD Pascal, and is not required in standard Pascal.

Here are examples of **case** statements with more than one constant in a constant list:

```
case digit of
    0: resume;
    1: prompt;
    2: restart;
    3: quit;
    4, 5, 6,
      7, 8, 9: {do nothing};
    end;

  case color of
    orange, brown: repaint;
    red: touch_up;
    blue, yellow, green: mix;
    end;
```

REPETITION

In Pascal, there are three forms of loops; **while**, **repeat**, and **for**. The **while** and **repeat** loop statements execute indefinitely, until some terminating condition is met. In a **while** statement, the condition is tested before the loop is executed, while in a **repeat** statement, the condition is tested after the loop. These two statements complement each other. A **for** statement uses a variable as a counter, and loops for a given number of times.

The While Statement

A **while** statement has the form (using EBNF):

```
while-statement =
    "while" Boolean-expression "do"
      statement
```

Before each iteration of the loop, the Boolean expression is tested; if it is TRUE, the statement is executed, otherwise the loop ends.

Clearly, unless the statement itself contains some code that will eventually cause the Boolean expression to be FALSE, the **while** loop will continue to execute indefinitely.

Since the **while** statement tests the Boolean expression *before* the statement is executed, it is often necessary to initialize variables that appear in the Boolean expression.

Occasionally, it is desirable for a **while** to execute indefinitely (for example, some kinds of concurrent processes require this). In this case, the programmer can specify:

```
while TRUE do < statement> ;
```

Here are two examples of **while** statements:

```
go_on:= TRUE; {this initialization is necessary}
while go_on do
    begin
        GENERATE(half, whole);
        if half < 1 then go_on:= FALSE;
    end;

num_piles:= 0; {initialization}
while not EOLN and (num_piles < pile_max) do
    begin
        num_piles:= num_piles+1;
        READ(pile[num_piles]);
    end;
```

Note that the second example is guaranteed to terminate. Presumably, the first example will terminate as well, provided the procedure GENERATE does what it is supposed to do.

The Repeat Statement

A **repeat** statement has the form (using EBNF):

```
repeat-statement =
    "repeat"
        statement { ";" statement }
    "until" Boolean-expression
```

The program executes the statement list, then tests the Boolean expression. If the Boolean expression is TRUE, the statement list is executed again, and so forth. If the Boolean expression is FALSE, the loop ends.

Since the Boolean expression is tested *after* the statement list is executed, there is usually no need to initialize the variables used in the Boolean expression.

When a **repeat** must execute indefinitely, the programmer can use the form:

```
repeat < statement list> until FALSE;
```

Here are two examples:

```
repeat READ(remainder) until EOF;

repeat
    num_piles:= num_piles+1;
    READ(pile[num_piles]);
until EOLN or (num_piles > = pile_max)
```

Note that the second **repeat** example accomplishes the same thing as the second **while** example in the preceding section, except that the assignment and the READ are always executed at least once. Notice that the conditions for terminating the loop have been inverted.

The For Statement

A **for** statement has two forms. Here is an EBNF description:

```
for-statement =
    "for" var-id ":=" start-value "to"
       stop-value "do"
           statement
    |
      "for" var-id ":=" start-value "downto"
         stop-value "do"
              statement
```

The *var-id* is a variable of some scalar or subrange type. It is called the index of the **for** loop. The *start-value* and *stop-value* are values of the same type as the index variable (typically INTEGER).

When a **for to** statement is executed, the start-value and stop-value are evaluated once. The index is assigned the value of the start-value. The statement is executed, and then the index is replaced by its successor (if it is an integer, it is incremented by 1). This continues until the value of the index is greater than the stop-value.

A **for downto** statement is executed in the same way as a **for to** statement, except that before each iteration of the loop, the index is replaced by its predecessor (if it is an integer, it is decremented by 1). The loop ends when the value of the index is less than the stop-value.

If the start-value is greater than the stop-value (or less than, for the **downto** form), then the statement is not executed at all. If the start-value equals the stop-value, the statement is executed once.

The index must be a local variable. It cannot be a **var** parameter.

After the **for** statement has executed, the value of its index is undefined, and the programmer must reinitialize the index variable if it is to be used again.

Here are some examples:

```
for weekday:= Mon to Sun do
    print_schedule(weekday);

for j:= i-1 downto 2 do
    p[j]:= p[j] + p[j-1];

for j:= 1 to i do
    begin
      WRITE(j,': ');
      WRITELN(p[j]);
    end;
```

BRANCHING

By *branching* we mean changing a program's flow of control by some means other than the statements already described in this chapter. This practice is discouraged in structured programming, but there are times (usually error conditions) when it is the least painful way to accomplish something. The **goto** statement allows unconditional branching. UCSD Pascal also provides the intrinsic procedures EXIT and HALT to allow emergency terminations.

The Goto Statement

A **goto** statement has the form (using EBNF):

```
goto-statement =
    "goto" label
```

The label must be declared, and must be local to the block (the routine or main program body) in which the **goto** statement appears (see Chapter 3). A label has the form of an integer in the range 0..9999.

UCSD Pascal requires a **goto** statement to be in the same block as the label it names; standard Pascal does not.

Execution proceeds from the labeled statement. This may cause some statements to be skipped, or some statements to be repeated.

Using EBNF, the form of a labeled statement is:

```
labeled-statement =
    label ":" statement
```

For example:

```
. . . {program code}
goto 1558;
. . . {more program code ---- }
        {all of this is skipped}
1558: {execution continues here}
        report_disaster;
. . .
```

The Procedure EXIT

A routine or a program can be terminated immediately by calling the UCSD intrinsic EXIT in three ways:

- EXIT(< routine name>)
- EXIT(< program name>)
- EXIT(**program**)

When EXIT is used in a procedure or function, execution returns to the caller of that routine. Any local files still open are closed and purged (see Chapter 7). If EXIT is used within a function, and called before the value of that function has been defined, the function returns an undefined value. If EXIT is used in a procedure or function that has been called recursively, control returns to the previous call (the call stack is *popped*).

When an EXIT process occurs, its execution ends.

Within the initialization and termination code of a **unit**, EXIT behaves in particular ways. These are described in Chapter 4.

When EXIT is called with the name of the program or the reserved word **program**, the program is terminated immediately and control returns to the operating system. Any open files are closed as if by a call to CLOSE(FileName, normal) (see Chapter 7).

The Procedure HALT

A call to the UCSD intrinsic HALT causes the program to abort with a runtime error. The effect is similar to that of pressing the **BRK/PAUS** key while a program is running.

Control is transferred to the operating system unless the debugger is running, in which case control is transferred to the debugger. The debugger is described in the *UCSD p-System Operating System Reference Manual.*

It should be evident that a call to HALT is either an emergency measure or a means of deliberately invoking the p-System debugger.

7

Input and Output

This chapter first describes the standard Pascal I/O intrinsics. It then covers I/O for external files, and finally device I/O. Standard I/O for internal files is described in Chapter 3.

Standard input and output in Pascal is done with the intrinsics READ, READLN, WRITE, WRITELN, EOF, EOLN, and PAGE (refer to the paragraph entitled Standard Pascal I/O).

External files under the UCSD p-System can be opened with extended calls to the intrinsics RESET and REWRITE. They can be closed with the intrinsic CLOSE. Disk files can be accessed randomly with the intrinsic SEEK. Untyped files are handled with the intrinsics BLOCKREAD and BLOCKWRITE (refer to the paragraph entitled Handling External Files).

In UCSD Pascal, device I/O can also be done with a group of lower-level intrinsics: UNITCLEAR, UNITREAD, and UNITWRITE (refer to the paragraph entitled USCD p-System Environment).

STANDARD PASCAL I/O

For examples of the use of these intrinsics, the reader should refer to Part 2, Chapter 4.

READ and READLN

The READ(FILENAME, ITEM1, ITEM2, ..., ITEMn) intrinsic reads characters from FILENAME, converts them into values, and sequentially assigns them to the variables ITEM1, ITEM2, through ITEMn.

The FILENAME parameter need not be present. If it is absent, the standard file INPUT is used. In the p-System, INPUT defaults to the device CONSOLE:, but this default can be overridden by the user; see the description of redirection in the *UCSD p-System Operating System Reference Manual.*

If FILENAME has been associated with an external file, it must be a text file (that is, a file of type TEXT, INTERACTIVE, or **file of** CHAR).

There can be one or more ITEMs. Each is a variable. If the type of the data read from the file does not match the type of the corresponding variable, a runtime error results.

An ITEM must be of type INTEGER (or subrange of INTEGER), long INTEGER, REAL, CHAR (or subrange of CHAR), STRING, or **packed array of** CHAR. Boolean values and structured types other than STRING or **packed array of** CHAR cannot be read.

If the input file is INTERACTIVE, READ behaves a bit differently than it would with a standard textfile. A standard READ(FILENAME,ch) is defined as the sequence:

```
ch:= FILENAME ^; GET(FILENAME);
```

Since an INTERACTIVE file normally contains no values until the user has started typing them in, this standard definition would result in a runtime error. UCSD Pascal therefore defines a READ on an INTERACTIVE file to be the opposite sequence from the standard:

```
GET(FILENAME); ch:=FILENAME ^
```

This altered order also affects the setting of EOLN.

When INTEGER or REAL values are read, leading blanks and carriage returns (<return>, ASCII CR) are ignored until a non-blank character is read.

The entry of a value is terminated by a space (' '), pressing the **RETURN** key, or a character that is not a digit (reals may be entered with an exponent such as e+24). When entering a value from an interactive file, the user can correct it before it has been terminated by using the **BACKSPACE** key and retyping; once a value has been read, there is no way to correct it.

When a STRING value is READ, it must be terminated by pressing the **RETURN** key. For this reason, strings should be read by READLN rather than READ.

The READLN (FILENAME, ITEM1, ITEM2, ..., ITEMn) intrinsic is identical to READ except that the ITEM parameters are optional, and READLN expects to read a <return> after all the ITEMs have been read; READLN will wait until this <return> is read.

Since STRING values must be terminated by pressing the **RETURN** key, READLN is the proper way to read strings; only one STRING value per call to READLN.

A call to READLN without parameters can be used to ignore the rest of a line. This is especially useful for stepping to the next line of a file after a series of READs:

```
n:= 0;
while not EOLN do
      begin
         READ(a[n]);
         n:= n+1;
      end;
   READLN;
```

WRITE and WRITELN

The WRITE (FILENAME, VALUE1, VALUE2, ..., VALUEn) intrinsic sequentially writes VALUE1, VALUE2, through VALUEn to the file FILENAME.

The FILENAME parameter need not be present. If it is absent, the standard file OUTPUT is used. In the p-System, OUTPUT defaults to the device CONSOLE:, but this default can be overridden by the user; see the discussion of redirection in the *UCSD p-System Operating System Reference Manual.*

If FILENAME is associated with an external file, it must be a textfile (that is, a file of type TEXT, INTERACTIVE, or **file of** CHAR).

There can be one or more values.

In UCSD Pascal, a value to be written must be of type INTEGER, long INTEGER, REAL, CHAR, STRING, or **packed array of** CHAR. Boolean values and structured types other than STRING or **packed array of** CHAR cannot be written. In standard Pascal, Boolean values can be written.

The WRITELN(FILENAME, VALUE1, VALUE2, ..., VALUEn) intrinsic is identical to WRITE, except that the VALUE parameters are optional, and it writes a carriage return (the **RETURN** key) after all other VALUEs have been written.

It is common practice to write several VALUEs to a single line with repeated calls to WRITE, and then finish the line by a call to WRITELN that has no parameters:

```
for i:= 1 to 10 do
    WRITE(a[i]:5);
WRITELN;
```

Field Specifications

When a value is written, it is written within an *output field* that is some number of characters wide. The default width of an output field is automatically wide enough to contain the value being printed. The user can override the default field width by explicitly specifying a field width within the call to WRITE or WRITELN.

Values can be formatted by entering them in this manner:

```
value1:m
realvalue:m:n
```

where:

m is a positive integer that specifies the output field width.

If the value is real, n can be included since it is a positive integer that specifies the number of decimal places to be written.

If the printed format of the value is shorter than the field width m, the value is right-justified, and blanks are written on the left. If the value is numeric and m does not specify enough spaces, the full value is written, and m is ignored.

Real values must have at least room for the number plus a sign on the left. If the real value is too large for the specified format, it is written in exponential form.

Values of type STRING are always written with a field length equal to their dynamic length, unless a field length m is specified. If m is greater than the string's dynamic length, the string is right-justified, and spaces are written on the left. If m is less than the string's dynamic length, the string is truncated on the right. The length of a **packed array of** CHAR is not dynamic, but otherwise it is written exactly as a STRING .

EOF and EOLN

The intrinsic functions EOF and EOLN have already been described in Chapter 3. They are both Boolean functions; EOF(FILENAME) returns TRUE when the end of FILENAME has been reached, and EOLN(FILENAME) returns TRUE when a < return> character has just been read.

The FILENAME parameter can be omitted, in which case EOF and EOLN refer to the predeclared file INPUT.

When reading from an INTERACTIVE file, EOLN is initially FALSE, and is set to TRUE only when a < return> character is read. The value received by the READ is a space (' ') rather than a < return> character.

PAGE

The PAGE(FILENAME) intrinsic writes a form feed character (ASCII FF, CHR(12)) to the text file FILENAME.

HANDLING EXTERNAL FILES

Files in Pascal programs can be associated with external files in the p-System. To the UCSD p-System, a file is any source of data or sink for data. Thus, a file can be a file on a block-structured device such as a floppy disk, an interactive device such as a terminal, a write-only device such as a printer, and so forth.

More information on p-System files and devices can be found in the *UCSD p-System Operating System Reference Manual.*

Opening and Closing Files

The following paragraphs explain how to open an external file (whether a device or disk file) with the procedures RESET and REWRITE. It also describes how to save or remove a file with the CLOSE intrinsic. The standard use of RESET and REWRITE is described in Chapter 3.

Files opened with RESET or REWRITE can be closed by the intrinsic CLOSE, which is described in the paragraph entitled CLOSE.

RESET

The RESET (FILENAME, EXT_FILE) intrinsic does a standard RESET on FILENAME, and also associates the Pascal (internal) file FILENAME with the p-System (external) file EXT_FILE.

The parameter EXT_FILE is a string expression that specifies the name of a p-System file.

If EXT_FILE is nonexistent, or the device is offline, or the file is already open, a runtime error results (unless I/O-checking has been disabled — see Chapter 10).

If EXT_FILE is a write-only device (such as PRINTER: or REMOUT:), a runtime error results, because RESET attempts to initialize FILENAME∧for all files that are not INTERACTIVE.

Here are some examples:

```
{opening disk files:}
    RESET(bookfile, 'CHAPTER2.TEXT');
    RESET (seismic, '5:MARCH.12.DATA');
{opening an INTERACTIVE device:}
    RESET(talk, 'CONSOLE:');
{opening a file with a variable name:}
    userfile:= '*PROJECT.TEXT';
    RESET(bookfile, userfile);
```

REWRITE

The REWRITE (FILENAME, EXT_FILE) intrinsic does a standard REWRITE on FILENAME, and also associates the Pascal (internal) file FILENAME with the p-System (external) file EXT_FILE.

The parameter EXT_FILE is a string expression that specifies the name of a p-System file.

If EXT_FILE is not the name of an existing file, a new file is created with that name.

If EXT_FILE is a file that already exists, REWRITE creates a temporary file to operate on while the program executes. This temporary file can supplant the original file (as with a REWRITE in standard Pascal), be discarded, or be saved under a new name. This depends on how the CLOSE intrinsic is used.

If EXT__FILE is a device that is offline, or a file that is already open, a runtime error results (unless I/O-checking has been turned off — see Chapter 10).

If there is no EXT__FILE parameter, REWRITE (FILENAME) is equivalent to REWRITE (FILENAME, 'FILENAME'), for the first eight characters of the FILENAME identifier.

Here are some examples:

```
{clearing a disk file:}
    REWRITE(bookfile, 'CHAPTER2.TEXT');
{creating a new disk file:}
    REWRITE(seismic, '5:MARCH.31.DATA');
'resetting an I/O device:'
    REWRITE(talk, 'REMOUT:');
{clearing a file with a variable name:}
    userfile:= '*PROJECT.TEXT';
    REWRITE(bookfile, userfile);
{creating a temporary file:}
    REWRITE(scratch) {= REWRITE(scratch,
'scratch')}
```

CLOSE

The CLOSE(FILENAME [, OPTION]) intrinsic closes the file FILENAME. After the close, the value of FILENAME∧ is no longer defined.

If present, OPTION must be one of the following words:

- Normal
- Lock
- Purge
- Crunch

The OPTION parameter can be omitted. If it is not present, CLOSE(FILENAME) is equivalent to CLOSE(FILENAME, normal).

The CLOSE (FILENAME, normal) intrinsic closes FILENAME. If FILENAME was associated with a disk file by a previous call to REWRITE, then the call to CLOSE deletes the temporary version from the directory, and the original file is unaffected.

The CLOSE(FILENAME, lock) intrinsic is equivalent to a normal CLOSE unless FILENAME was associated with a disk file by a previous call to REWRITE. In that case, the temporary file is saved, and the original version deleted.

The CLOSE(FILENAME, purge) intrinsic is equivalent to a normal CLOSE unless FILENAME was associated with a disk file or a device. If FILENAME is associated with a disk file, that disk file is removed from the directory. If FILENAME is associated with a device, the device goes offline.

The CLOSE(FILENAME, crunch) intrinsic is equivalent to a lock CLOSE, except that the file is truncated where it was last accessed. The position of the last GET, PUT, READ, or WRITE to the file becomes the last record in the file.

When a Pascal program finishes its execution normally (that is, no runtime error occurred), a CLOSE (FILENAME, normal) is done on all files. This does not affect files that have already been closed by the programmer.

Random Access to Files

Unlike standard Pascal, where files are strictly sequential, UCSD Pascal allows file records to be accessed in random order by use of the intrinsic SEEK, which follows.

SEEK

The SEEK (FILENAME, INDEX) intrinsic sets the file window to the INDEX'th record in FILENAME; EOF is set to FALSE .

FILENAME cannot be a textfile or an untyped file. An internal file is a textfile if it is of type TEXT, INTERACTIVE, or file of CHAR. Trying to SEEK on a textfile or untyped file causes a runtime error.

INDEX is an integer. File records are numbered starting from zero. If INDEX is less than zero or greater than the largest record index in the file, SEEK accepts this; the next GET or PUT to FILENAME will cause EOF to be set to TRUE.

A GET(FILENAME) or PUT(FILENAME) call must be made between two consecutive calls to SEEK. If this is not done, the contents of the window will be undefined and unpredictable.

Note that SEEK does not set the value of the window variable (FILENAME ^). That must be done by a call to GET or PUT.

Untyped Files

In UCSD Pascal, a file can be declared in the following way:

FileName: file;

This is called an *untyped* file. An untyped file does not have a window location or a window value. It must be associated with an external file by RESET or REWRITE, and the only way to manipulate it is by the UCSD intrinsic functions BLOCKREAD and BLOCKWRITE, which are described in this section.

The BLOCKREAD and BLOCKWRITE functions do no type checking or range checking at all, so their use can lead to errors. The programmer must take care that they do not destroy any valuable data.

BLOCKREAD

The BLOCKREAD (FILENAME, BUFFER, COUNT, RELBLOCK) intrinsic attempts to read COUNT blocks from FILENAME into BUFFER, starting from RELBLOCK. It returns the number of blocks that were actually transferred.

FILENAME is an untyped file that has been associated with an external file by a call to RESET or REWRITE.

BUFFER is a typeless parameter (typeless parameters are described in Chapter 3).

COUNT is an integer.

RELBLOCK need not be present. If it is present, it is a (zero-based) block index into FILENAME, and indicates the location from which the read should begin.

If the value returned by BLOCKREAD does not equal COUNT, then either the end of the file was encountered, or a read error occurred. The programmer should check these conditions. If the end of file IS encountered, BLOCKREAD sets EOF to TRUE.

Successive BLOCKREADs on the same file will continue to read blocks in sequence, unless RELBLOCK is used to explicitly name a location, or the file is reinitialized with RESET or REWRITE.

BLOCKWRITE

The BLOCKWRITE (FILENAME, BUFFER, COUNT, RELBLOCK) function attempts to write COUNT blocks from BUFFER into FILENAME, starting at RELBLOCK. It returns the number of blocks that were actually transferred.

FILENAME is an untyped file that has been associated with an external file by a call to RESET or REWRITE.

BUFFER is a typeless parameter.

COUNT is an integer.

RELBLOCK need not be present. If it is present, it is a (zero-based) block index into FILENAME, and indicates where the writing should begin.

If the value returned by BLOCKWRITE does not equal COUNT, then either the end of the file was encountered, or a write error occurred. The programmer should check these conditions. If the end of file IS encountered, BLOCKWRITE sets EOF to TRUE.

Successive BLOCKWRITEs on the same file will continue to write blocks in sequence, unless RELBLOCK is used to explicitly name a location, or the file is reinitialized with RESET or REWRITE.

THE UCSD p-SYSTEM ENVIRONMENT

Keyboard

Files of type INTERACTIVE have already been described (in the section, STANDARD PASCAL I/O, and in Chapter 3). The standard predeclared files INPUT and OUTPUT are interactive in UCSD Pascal; so is the predeclared file KEYBOARD.

When a character is read from the file INPUT, that character is also echoed on the CONSOLE:. The KEYBOARD file, on the other hand, does not echo the character that is typed. The most common use of KEYBOARD is in reading a user's response to a promptline, since there is no need to redisplay that character on the screen.

A read from INPUT (for example, READ(INPUT, character);) is equivalent to:

```
READ(KEYBOARD, character);
WRITE(OUTPUT, character);
```

Device I/O

This section describes the UCSD intrinsics that can be used to control peripheral devices directly. The use of these intrinsics is somewhat error-prone, and the programmer should be cautious with them. On the other hand, they can be much faster than the standard Pascal I/O intrinsics, and are therefore indispensable for some applications.

For a programming example that uses these intrinsics, the reader should refer to Part 2, Chapter 4.

All of these intrinsics require a parameter, DEVICE_NUMBER, which specifies the peripheral device that the routine call affects. DEVICE_NUMBER must be in the range 1..8.

These are the current standard device numbers in the p-System (they are also shown in the *UCSD p-System Operating System Reference Manual*):

1	CONSOLE:	Screen and keyboard (echoes)
2	SYSTERM:	Screen and keyboard (no echo)
4	< disk name> :	A disk drive
5	< disk name> :	An alternate drive disk
6	PRINTER:	Line printer
7	REMIN:	Serial line input
8	REMOUT:	Serial line output

Because the device I/O intrinsics reference devices directly, their I/O cannot be redirected (redirection is described in the *UCSD p-System Operating System Reference Manual*).

UNITCLEAR

The UNITCLEAR(DEVICE_NUMBER) intrinsic resets the specified device to its power-up state.

UNITREAD

The UNITREAD(DEVICE_NUMBER, BUFFER, LENGTH, BLOCKNUMBER, FLAG) intrinsic reads LENGTH bytes from the specified device into BUFFER.

BUFFER is a typeless parameter. LENGTH is an integer.

BLOCKNUMBER and FLAG are optional parameters. If they are not present, they default to zero.

If BLOCKNUMBER is present, it indicates the block of the specified device from which the read will start (the offset is zero-based). If the device is not block-structured, this parameter is ignored.

If FLAG is present and equal to 1, the transfer is done asynchronously rather than synchronously. If the hardware does not support asynchronous I/O, FLAG is ignored.

If FLAG equals 2 and the device is block-structured, UNITREAD transfers one physical sector. BUFFER should be large enough to contain this sector. LENGTH must be set to 0. BLOCKNUMBER is the relative sector number (zero-based).

For other uses of FLAG, refer to the BIOS documentation in the *UCSD p-System Internal Architecture Guide.*

To specify FLAG but not BLOCKNUMBER, simply precede FLAG with two commas:

```
UNITREAD(5, name_array, count,, 1);
```

UNITWRITE

The UNITWRITE(DEVICE_NUMBER, BUFFER, LENGTH, BLOCKNUMBER, FLAG) function writes LENGTH bytes from BUFFER to the specified device.

BUFFER is a typeless parameter. LENGTH is an integer.

BLOCKNUMBER and FLAG are optional parameters. If they are not present, they default to zero.

If BLOCKNUMBER is present, it indicates the block of the specified device where the write will start (the offset is zero-based). If the device is not block-structured, this parameter is ignored.

If FLAG is present and equal to 1, the transfer is done asynchronously rather than synchronously. If the hardware does not support asynchronous I/O, FLAG is ignored.

If FLAG equals 2 and the device is block-structured, UNITWRITE transfers one physical sector. BUFFER should be large enough to contain this sector. LENGTH must be set to 0. BLOCKNUMBER is the relative sector number (zero-based).

For other uses of FLAG, refer to the BIOS documentation in the *UCSD p-System Internal Architecture Guide.*

To specify FLAG but not BLOCKNUMBER, simply precede FLAG with two commas:

```
UNITWRITE(5, name_array, count,, 1);
```

The TIME Procedure

The TIME (HIWORD, LOWORD) procedure sets HIWORD_LOWORD to a 32-bit unsigned integer that contains the current value of the system's clock in sixtieths of a second.

The programmer should be careful not to treat LOWORD as a negative two's complement integer.

The value returned is *not* correlated to the time of day. The TIME procedure is usually used for incremental time measurements, such as measuring a program's performance by calculating benchmarks.

IORESULT

The IORESULT intrinsic returns an integer that indicates the status of the last I/O operation performed.

The compiler generates I/O checks for *all* I/O operations except device I/O intrinsics. An I/O error at runtime causes a program to abort. If the programmer would rather that the program itself checked IORESULT and took steps to correct an I/O error, I/O checking must be turned off. This is done with the { $I− } compiler directive (see Chapter 10).

The following list contains the possible values of IORESULT:

```
 0 - No error
 1 - Parity error (CRC)
 2 - Illegal device number
 3 - Illegal I/O request
 4 - Data-com timeout
 5 - Volume went offline
 6 - File lost in directory
 7 - Bad file name
 8 - No room on volume
 9 - Volume not found
10 - File not found
11 - Duplicate directory entry
12 - File already open
13 - File not open
14 - Bad input information
15 - Ring buffer overflow
16 - Write protect
17 - Illegal block
18 - Illegal buffer
```

Each process is saved with its own IORESULT, so the IORESULT of one concurrent process does not interfere with the IORESULT of another.

I/O done by the p-System itself does not affect a user program's IORESULT behind the user's back.

Note that the following:

```
WRITELN(ioresult is:, IORESULT);
```

is wrong, since writing the string constant will alter IORESULT . The safest way to write an IORESULT is:

```
iocheck:= IORESULT;
WRITELN(ioresult is: , iocheck);
```

GOTOXY

The following paragraphs describe GOTOXY, which allows the user to position the cursor anywhere on the CONSOLE: display unit. More sophisticated screen control can be accomplished by using the operating system's display control unit, which is described in the *UCSD p-System Internal Architecture Guide*).

The GOTOXY (x,y) intrinsic positions the CONSOLE:'s cursor at (x,y)

where:

x indicates a column and y indicates a row. The x and y are integers. The upper left corner of the screen is assumed to be (0,0). If x or y is too large, the cursor is placed at the edge of the screen. Your terminal displays 80 * 24 characters.

Memory Management

Please keep in mind that the information in this chapter applies only to UCSD Pascal.

For further discussion of memory management critical to the success of a program, see Part 2, Chapter 7.

These topics are also discussed in the *UCSD p-System Operating System Reference Manual* and the *UCSD p-System Internal Architecture Guide.*

THE p-SYSTEM RUNTIME ENVIRONMENT

There are three dynamic data structures that are used to manage main memory while a program is running. These are the stack, the heap, and the codepool.

The stack is a last-in-first-out stack that is used for procedure and function calls, the storage of static variables, and temporary values during expression evaluation.

The heap is essentially a last-in-first-out structure as well, but its management is more complex. It is used to store dynamic variables, the stacks of subordinate processes, and non-relocatable code segments.

The codepool is a collection of active code segments, including the code segment that is currently running. A code segment is a portion of a codefile. Code is swapped in and out of memory one segment at a time. Code segments are described in further detail in the following section.

When extended memory is used, the stack and heap occupy one area (one 64K page) of memory, and the codepool occupies a different area. This tends to improve performance for both user programs and the p-System itself.

SEGMENTATION

Code Segments

A code segment is typically the product of a single compilation; a program or unit is compiled into a single code segment. The code segment that contains a program or a unit is called the *principal segment*; the segments that contain segment routines are called *subsidiary segments*.

A programmer can also create a subsidiary segment that contains a single routine (however complex) by use of the reserved word **segment**. This word must be the first word in a routine heading, for example:

```
segment procedure initialize;

segment function HandleStore
                              (priority: INTEGER): INTEGER;

segment process switching(light: SEMAPHORE);
```

As previously mentioned, program code is swapped in and out of memory one segment at a time. Thus, a programmer can improve the memory utilization of a program by designating certain routines as segment routines which need not be in memory at all times. In particular, routines which are only called once or twice during a program (such as initialization and termination code) are very good candidates for segment routines.

Within a program, the declaration of the code *bodies* of all segment routines must precede the declaration of any non-segment routine code. If a segment routine must call a non-segment routine, the non-segment routine must be declared by a **forward** declaration that precedes the segment routine's declaration.

The name of a code segment is the first eight characters of either the program name, the unit name, or the segment routine name (depending on how the code segment was created).

Controlling Segment Residence

This section describes the UCSD intrinsic procedures MEMLOCK and MEMSWAP, which may be used to control a segment's residence in main memory.

MEMLOCK

The MEMLOCK(SEGMENT_LIST) intrinsic locks all of the segments named in SEGMENT_LIST into main memory.

SEGMENT_LIST is a string that contains the names of segments, separated by commas. Names of nonexistent segments are ignored.

All segments named in SEGMENT_LIST must remain in the codepool until they are released by a call to MEMSWAP. Until that time, they cannot be swapped out of memory.

If a segment that is named is not in the codepool, it is locked in memory the next time it is loaded.

MEMSWAP

The MEMSWAP(SEGMENT_LIST) intrinsic unlocks all of the segments named in SEGMENT_LIST.

SEGMENT_LIST is a string that contains the names of segments, separated by commas.

All segments that are named in SEGMENT_LIST and have been locked in the codepool are released, and can now be swapped out to disk (under the control of the operating system).

If a segment that is named is not in the codepool, or is in the codepool and has not been locked, the name is ignored.

FREE SPACE

This section describes the UCSD intrinsic functions MEMAVAIL and VARAVAIL, which allow a program to discover how much free space remains in main memory.

MEMAVAIL

The MEMAVAIL intrinsic returns an integer which is the number of unallocated words in main memory.

This number is the sum of the number of words between the codepool and the stack, plus the number of words between the codepool and the heap.

Note that the number returned by MEMAVAIL can be less than the number of total available words of memory space, since there may be segments in the codepool that can be swapped out.

VARAVAIL

The VARAVAIL(SEGMENT_LIST) intrinsic returns the number of available words of particular configuration of main memory. It is typically used in conjunction with VARNEW.

SEGMENT_LIST is a string that contains the names of segments, separated by commas. These are segments for which the user wants space to be available.

The VARAVAIL intrinsic returns the number of words that would be available if all of the segments named in SEGMENT_LIST were loaded. It assumes that all memory-locked segments remain in main memory, along with all of the segments named in SEGMENT_LIST.

If a name in SEGMENT_LIST is not known to the operating system, it is ignored.

FREE SPACE ON THE HEAP

The following paragraphs describe a number of intrinsics that allow a program to allocate and deallocate free space on the heap (remember that the heap is used to store dynamic variables).

The NEW and DISPOSE intrinsics are the standard means of allocating and deallocating dynamic variables on the heap. VARNEW and VARDISPOSE can be used to allocate/deallocate variable-sized areas. These are system-level intrinsics that do no checking; they can cause problems if not used carefully. Finally, MARK and RELEASE are a means of controlling still larger portions of heap use: they are also system-level tools.

It is very important that the pairs VARNEW/VARDISPOSE and MARK/RELEASE match up correctly while the program is running. If they do not, the integrity of the heap may be lost: this usually causes the system to crash.

The NEW and DISPOSE intrinsics are described in Chapter 3.

VARNEW

The VARNEW(POINTER, COUNT) intrinsic does a NEW on COUNT words, and returns an integer that is the number of words actually allocated.

POINTER is a pointer to any type (it should point to the type of data for which space is being allocated). COUNT is an integer.

If it is possible to allocate COUNT words, then the number returned by VARNEW should equal COUNT. If COUNT words are not available, then no space is allocated, and VARNEW returns zero.

VARDISPOSE

The VARDISPOSE(POINTER, COUNT) intrinsic does a DISPOSE on COUNT words.

POINTER is a pointer to any type; it should be the same as the pointer in the VARNEW that corresponds to this VARDISPOSE. COUNT is an integer, and should be equal to the COUNT in the corresponding VARNEW.

If either POINTER or COUNT is incorrect, the heap's integrity is destroyed.

MARK and RELEASE

The MARK(POINTER) intrinsic marks a location on the heap by creating and allocating a Heap Mark Record (HMR). POINTER is set to point to that record.

It is customary to declare POINTER as a ∧ INTEGER.

Note that the only heap space allocated by MARK is the space occupied by the HMR itself. Space for dynamic variables beyond the HMR must be allocated by other means (such as NEW and VARNEW); such space can be freed by a subsequent call to RELEASE.

The RELEASE(POINTER) intrinsic cuts the heap back to the HMR that POINTER points to.

Calls to MARK and RELEASE should come in pairs. POINTER should equal the value of POINTER that was set by the previous call to MARK.

POINTER is customarily declared as a ∧ INTEGER.

All space on the heap that was allocated (by NEW or VARNEW) since the previous MARK is deallocated.

If MARK and RELEASE are not paired correctly, or RELEASE is called with an incorrect pointer, the integrity of the heap is destroyed.

9

Concurrency

This chapter describes the management of concurrent routines (processes).

Concurrency, like memory management, pertains more to the p-System than to the UCSD Pascal language, and more information about concurrency may be found in the *UCSD p-System Operating System Reference Manual* and the *UCSD p-System Internal Architecture Guide.*

For programming examples that use concurrency, refer to Part 2, Chapter 5.

CONCURRENT EXECUTION

Your TI Professional Computer has only one processor. Processes do not truly run at the same time, but share the processor, which is switched between them. Thus, using processes can slow a program down somewhat; there is usually a compensating gain in logical consistency, or efficiency in handling certain problems such as interrupts.

The interpreter controls concurrent processes, and initiates them depending on their priority. Priority is discussed later. Once a process is started, it continues to run until it is finished, until it is interrupted, or until it must wait for another process. At this point, the processor is given another process, and it runs in the same manner. The p-System does not do any sort of time-slicing or task-switching among ready processes that have equal priority.

PROCESSES

A process is a form of routine, and is declared in the same general way. Here is an EBNF description of a process heading:

```
"process" identifier [
               "(" ["var"] id-list ":" type-id
                    { ";" ["var"] id-list ":" type-id }
               ")" ] ";"
```

For example:

```
process example (var result: INTEGER;
                 var pause: SEMAPHORE);
```

The remainder of a process declaration is identical to a procedure or function declaration.

Processes must be declared global to a program; they cannot be declared within a procedure, function, or other process.

A process is not called as a procedure or function would be called. Instead, it is initiated by a call to the UCSD intrinsic procedure START.

The same process can be started any number of times; each of these instances is a concurrent process that runs independently of any other process. It may by synchronized with other processes (including other instances of the same process) by the use of semaphores.

The p-System assigns each instance of a process its own PROCESSID. PROCESSID is a predeclared type whose values are represented by integers. A program can examine a process PROCESSID (this can be useful for debugging purposes). There are no operations for PROCESSID.

Each instance of a process is also assigned a priority and a stack-size value.

A process priority determines its position in the queue of processes that are waiting to be run. A priority is a value in the range 0..255; the priority of a user program is 128. The default priority is 128 for all user processes; this value can be changed by the user in the call to START. A priority of 255 should not be used, as this conflicts with the operating system's Faulthandler process.

Processes can have the same priority. If so, they are placed in the queue in the order in which they were entered.

The stack for a process is allocated on the heap, rather than the stack itself. The default size for this area is 200 words. This value can also be changed by the user in the call to START.

INITIATING A PROCESS

As mentioned before, a single instance of a process is initiated by a call to the UCSD intrinsic procedure START; the same process can be initiated any number of times. The START procedure is described in the following paragraphs.

START

The START(<process call>, PROC_ID, STACKSIZE, PRIORITY) procedure initiates the process named in <process call>.

The term <process call> is the name of a process followed by a parameter list, if necessary. It has the same appearance as a procedure call.

The remaining three parameters are optional.

PROC_ID is of type PROCESSID. If present, it is set to the processid value that is assigned to this instance of the process.

STACKSIZE specifies the number of words that will be allocated (on the heap) to the process stack. If not present, a default of 200 is used.

A process stack must have enough room for:

- Five words
- The number of words occupied by the process's local (static) variables
- Room for the activation records of all procedures and functions that the process calls
- Room to evaluate expressions on most machines

Activation records are described in the *UCSD p-System Internal Architecture Guide.*

An overflow on a process stack causes a runtime error.

The programmer should be cautious about using RELEASE when the heap contains the stack of a START process.

PRIORITY is in the range 0..255. If not present, this value defaults to the priority of its parent process (which is a user program whose priority is 128). A priority of 255 should not be used.

PROCESS SYNCHRONIZATION

A process must often wait for some event to occur, or some other process to complete an action. A process can also contain a sequence of statements that must not be interrupted; such a sequence is known as a *critical section.* The programmer can deal with these situations by the use of semaphores and the intrinsic routines that handle them.

Semaphores

A semaphore is a variable of type SEMAPHORE. A semaphore consists of a queue of processes that are waiting on that semaphore, and a count in the subrange 0..MAXINT.

Such semaphores are called 'counting semaphores', as opposed to *Boolean semaphores*. A Boolean semaphore has only the values TRUE and FALSE, and can be simulated by restricting a counting semaphore to the values 0 and 1.

When a process must wait for an event to occur, it calls the UCSD intrinsic WAIT(SEM). This suspends the process until the semaphore SEM allows it to continue. For this to happen, another process must call the intrinsic SIGNAL(SEM).

When the value of a counting semaphore is equal to zero, it is unavailable, and a WAIT will continue to wait on that semaphore until it becomes greater than zero.

Every semaphore that is used in a program must be initialized by a call to the UCSD intrinsic SEMINIT, which is described below. If a program attempts to use a semaphore that has not been initialized, the results are unpredictable.

SEMINIT

The SEMINIT(SEM, INITIAL) intrinsic initializes the semaphore SEM, and sets its count equal to the value INITIAL. INITIAL must be an integer in the range 0..MAXINT.

SIGNAL and WAIT

The following paragraphs describe the UCSD intrinsic procedures SIGNAL and WAIT.

The SIGNAL(SEM) procedure signals the semaphore SEM.

If no processes are waiting for SEM, the value of SEM is incremented.

If one or more processes are waiting for SEM, then the process at the head of SEM's queue is activated by adding it to the queue of ready processes. (A task switch can take place if the process at the head of SEM's queue has a higher priority than the process that was running.)

The WAIT(SEM) procedure waits on the semaphore SEM.

If the count of SEM is greater than zero, then it is decremented and the process that called WAIT continues.

If the count of SEM equals zero, then the process calling WAIT is suspended until SEM is made available again by a call of SIGNAL(SEM) in another process.

EVENT HANDLING

A program can associate a semaphore with an event (such as a hardware interrupt) by means of the UCSD intrinsic procedure ATTACH.

The ATTACH(SEM, I_VEC) intrinsic associates the semaphore SEM with the external interrupt vector I_VEC.

Whenever the hardware raises the interrupt in question, the system automatically calls SIGNAL(SEM).

The possible values of I__VEC, and the hardware states they represent, vary widely from processor to processor. The user should refer to machine-specific documentation for further information.

To detach a semaphore from an interrupt, and free it for other use, the programmer may call attach in the following way:

```
ATTACH(NIL, vector);
```

where:

the value of vector is the value that was previously attached to SEM.

A vector must be attached to only one semaphore at a time, and the semaphore must remain in memory for the entire time it is attached.

10

Compilation

When a UCSD Pascal program is compiled, it can contain directions to the compiler that control the compiler's output. Separate text files can be included in a single compilation, and portions of a program can be conditionally compiled. Under the p-System, it is also possible to call assembled routines (native code) from a Pascal program with the **external** construct.

COMPILER OPTIONS

A compiler option appears in a *pseudo-comment.* A pseudo-comment is a comment whose left delimiter ({ or (*) is immediately followed by a dollar sign ($). For example:

```
{$I+}
(*$Q-*)
```

A single pseudo-comment can contain several compiler options:

```
{$I+,Q-}
```

If a pseudo-comment contains more than one option, only the *last* option can be a string option.

The default options for a compilation are:

```
{$R+, I-, L-, U+, P+}
```

Unless these are explicitly overridden, they remain in effect whenever a program is compiled. The meaning of each of these letters is described in the following paragraphs.

Stack Options

The I (I/O-check), R (Range-check), and conditional compilation flags are known as *stack options*. The on/off state of these options can be nested up to 15 levels deep.

Each of these options (and each individual flag) has its own stack. Whenever a + or − is specified in the pseudo-comment for one of these options, that value is pushed onto the stack. The stack can be popped by using the character ∧ in place of + or −.

If more than 15 values are pushed onto the stack, the bottom of the stack is lost. If the stack is popped when it is empty, the value is always − (off).

Switch Options

Switch options are compiler options that have either an on or off state, as indicated by a + or − when the option appears in the text.

I — I/O check

This is a stack option. The default is I+. When I− is specified, the compiler stops generating the test code that normally follows every I/O statement (except for UNITREAD and UNITWRITE).

If the programmer wishes to test IORESULT explicitly after an I/O operation, in order to correct any errors that may have occurred, then I− must be specified before the operation is done.

L — Listing

The default is L−. L+ causes the compiler to write a source program listing to the file *SYSTEM.LST.TEXT. L can also be used as a string option, in which case the user can specify a different name for the list file.

N — Native Code

The default is N−. If you want native code to be generated for a routine, N+ must appear in a pseudo-comment *before* the first **begin** in that routine.

This option causes the compiler to generate special information that is used by the CODEGEN utility. After a program that contains one or more routines to be translated into native code has been compiled, CODEGEN must be used. See the *UCSD p-System Reference Manual* for more information.

P — Page

The default is P+. This causes the listing to be paginated so that it can be legibly printed on paper that is 8-1/2 inches long.

A pseudo-comment that contains only P (for example, {$P}) forces the listing to start a new page.

Q — Quiet Compile

Q defaults to −.

Q+ suppresses the compiler's output to CONSOLE:, except for error messages.

Q− allows the compiler to send information on its progress to CONSOLE:.

R — Range Check

This is a stack option. The default is R+. If R− is specified, the compiler stops emitting code to check the range of array indices and subrange expressions, the type of assignments, and so forth.

Programs compiled with R− run slightly faster, but invalid assignments do *not* cause a runtime error, leading to unpredictable and sometimes disastrous results. Unless a program is extremely time-critical, and has been thoroughly debugged, R− should not be used.

The letter R is also used for the Realsize option.

R — Realsize

Real numbers are either 32 bits (2 words) or 64 bits (4 words) long. The p-System defaults to 2-word reals. With the R option, a programmer can override this default.

R4 causes the compiler to generate four-word real numbers.

The pseudo-comment that contains this option must appear before the reserved word **program** or **unit**.

U — User Program

The default is U+. U−specifies that the compilation may use unit names that belong to the operating system. If U−is specified, the pseudo-comment it appears in must appear before the reserved word **program** or **unit**.

String Options

String options require the programmer to specify a string that is used by the compiler. This is either the name of a file, or text to be included in a file. The string can be preceded by zero or more spaces. Preceding it with a single space is customary.

If the pseudo-comment uses (* *) delimiters, the enclosed string cannot contain a *.

If the filename begins with a + or − (which is unlikely), then there must be a space after the letter that indicates the option; otherwise, the compiler will treat the option as a switch option and not a string option.

C — Copyright

The string is placed in the copyright field of the codefile (this resides in the segment dictionary (see the *UCSD p-System Internal Architecture Guide*).

For example:

```
{$C Copyright 1931 by Wholly Imaginary Systems}
```

I — Include File

The string is the name of a text file. The text of the specified file is compiled into object code at the position of the pseudo-comment.

If the compiler cannot open the file under the name that is given, it appends '.TEXT' to the name and tries again. If the second attempt fails or an I/O error occurs while reading the include file, the compiler generates a fatal syntax error and aborts.

Include files can be nested up to three levels deep.

Include files can contain **const**, **type**, **var**, and routine declarations (within the include file, they must be in their proper order). If this is the case, the pseudo-comment must precede any blocks of code that appear in the main program file. There can be more than one such include file; if so, only the *last* such include file can contain procedure code.

Here is an example of a program with an include file that contains declarations:

```
program FunnyDeclarations;

  const  a=1;
  type   guess = INTEGER;
  var   gosh: guess;
```

```
procedure fancy (p, q: INTEGER);
   forward;

{$I MOREDECS}

procedure fancy;
   begin ... end;

begin {main program} ... end.
```

These are the contents of MOREDECS:

```
const b=2;
type nonsense = (l, m, n);
var stuff: nonsense;

procedure plain;
   begin
      WRITELN('plain was called');
   end;
```

L — Listing

This corresponds to the preceding switch option. The string is the name of a text file to which the listing is written. Note that this name can be any valid file name, but if the user wishes to edit it later, it must be created with a suffix of .TEXT.

The following is an example of a program listing:

```
393   10   10:D     1 PROCEDURE IOCheck;
{ commented out ';' }{;
{ commented out ';'}This procedure will check the I/O
{ commented out ';'}  operations of the index as it is
                      rebuilding
397   10   10:D     1 }
398   10   10:0     0 BEGIN
399   10   10:1     0      IF ioresult <> 0 THEN
400   10   10:2     6         BEGIN
401   10   10:3     6              p1^:= 'index I/O failure.';
402   10   10:3     32           prompt(errorline);
403   10   10:2     38         END;
404   10   10:0     38 END: {IOCheck}
405   10   10:0     50
406   10   11:D      1 PROCEDURE DropIndex(position:
                       ISAM Cover);
```

The numbers that precede the actual lines of the Pascal program show you:

- The line number
- The code segment number
- The routine number : the lexical level
- The number of words of data or code that have been allocated to the routine so far

Lines that are declarations show a *D* instead of the lex level number, and lines from a unit's interface part show a *U*.

These numbers can be replaced by the warning, { commented out ';' }. This warns you that a semicolon appears within a comment, which often happens when a comment has accidentally swallowed some lines of Pascal code.

If there are errors during compilation, a message with the error number appears below the offending line. If the file *SYSTEM.SYNTAX is on line, the text of the error message is printed as well. These syntax errors are also shown in Appendix D of this manual.

T — Title

The string becomes the title of each new page in the listing file.

U — Uses Library

The string is the name of a codefile. The compiler searches that file for the code of any units used in SUBSEQUENT uses declarations. See Part 2, Chapter 6, for an example of the use of this option. Note that U is also the name of a switch option.

Conditional Compilation

Code to be conditionally compiled is bracketed by the options B and E (which are described later). Whether it is compiled or not depends on the value of a flag, which is set by the D option (also described later). The state of each flag is saved on a stack, as described previously.

B — Begin

This is a string option. The string is the name of a flag that has been defined by a previous D option. If only the string appears, the following code is compiled if the flag is true. If the string is followed by a −, the code is compiled if the flag is false. If the string is followed by a + or ∧ , these characters are ignored.

The section of code to be conditionally compiled must be delimited by a B option and an E option that names the same flag. Sections to be conditionally compiled cannot be nested.

D — Declare

This is a string option. The string is the name of a flag. Its initial value is true unless its name is followed by a −.

All flags must be declared before the reserved word **program** or **unit**. The D option can be used again in a subsequent portion of the program to redefine the value of a flag: + pushes a true value, − pushes a false value, and ∧ pops the flag's stack.

The names of flags follow the rules for Pascal identifiers.

E — End

This is a string option. The string must be the name of a flag used in a previous B option. Any characters that follow the flag are ignored.

Here is a code fragment that illustrates conditional compilation:

```
{$D DEBUG+}
program ToBeTested;

    . . .

    {$B DEBUG}
    procedure test_routine;
       . . .
       begin ... end;
    {$E DEBUG}

    . . .

       begin
         . . .
         {$B DEBUG}
         test_routine;
         {$E DEBUG}
         . . .
       end.
```

EXTERNAL ROUTINES

Under the p-System, a Pascal program can call an assembled routine. This routine must be declared as **external** in the Pascal source, for example:

```
procedure native__code (n: INTEGER); external;

function speed (rush: REAL, direct: BOOLEAN): status;
  external;
```

This construction is analogous to the declaration **forward**.

The assembly-language routine itself must correspond to standard P-code protocols; refer to the *UCSD p-System 8086/88 Assembler Reference Manual* for further information.

Part 2

A Guide for UCSD Pascal Programmers

Introduction

The purpose of this guide is to introduce the reader to the techniques of using UCSD Pascal, and present a number of programming examples.

The programs in Chapters 1 through 4 illustrate basic topics that are essentially common to all Pascal programmers, whether they use UCSD Pascal or some other dialect, although it is always UCSD Pascal that is described. The programs in Chapters 5 through 9 are more advanced, and present programming problems that are essentially unique to UCSD Pascal, the UCSD p-System, and the microcomputer environment.

Chapter 1, Part 1 is a brief survey of the features of Pascal. You may wish to read it before you read Chapter 1 of Part 2.

If you are not a programmer, you will probably find this guide difficult and we recommend that you start with a simpler text. We assume you know the basic concepts of programming, and have written some programs of your own, probably in some other language.

If you have programmed in a language which is not structured (BASIC or FORTRAN, for instance), you will find that Pascal requires you to pay more attention to the definition of data and the form (the structure) of algorithms. When reading this guide, pay special attention to the first three chapters. The payoff for this extra effort results in programs that are more reliable and easier to maintain.

We have tried to cover as much ground as possible, but this guide does not describe all of UCSD Pascal. The details we have left out can be found in Part 1.

Our approach is to present working programs, and a working set of programming concepts at each stage. Many samples introduce a variety of concepts that in the definition would be listed under separate headings. The intention is to provide you with the tools to begin writing useful programs as soon as possible.

We do not guarantee that the sample programs are perfect, but they have been formatted and printed from source code that was actually tested. To the best of our knowledge, they run correctly. The sample outputs were also printed directly from program output.

1

Bootstrapping the Programmer

This chapter introduces most of the topics involved in writing a UCSD Pascal program, but does not discuss any of them in detail. The intent is to illustrate the style of Pascal and the basic form of Pascal programs. By the end of the chapter, you should be able to write simple programs in UCSD Pascal, but you will have to read further to learn about more powerful (and more subtle) aspects of the language.

The first program is one that prints a table of factorials on the console:

```
program Fact;
   var
      i: INTEGER;
      prod: REAL;
   begin
      WRITELN('n factorial of n');
      prod:= 1.0;
      for i:= 1 to 20 do
         begin
            prod:= prod*i;
            WRITELN(i,' ',prod);
         end;
   end.
```

The factorial of a positive non-zero integer n is defined to be the product of all the integers between 1 and n. Fact prints the following table:

```
 n  factorial of n
 1  1.00000
 2  2.00000
 3  6.00000
 4  2.40000E1
 5  1.20000E2
 6  7.20000E2
 7  5.04000E3
 8  4.03200E4
 9  3.62880E5
10  3.62880E6
11  3.99168E7
12  4.79002E8
13  6.22702E9
14  8.71783E10
15  1.30767E12
16  2.09228E13
17  3.55687E14
18  6.40237E15
19  1.21645E17
20  2.43290E18
```

Program Fact is composed of three parts; a program heading, a variable declaration part, and a program body.

Fact begins with the heading:

```
program Fact;
```

which notifies the compiler that the following text will define a program, and that the program will have the name Fact.

The program heading ends with a semicolon. In Pascal, semicolons serve to end declarations and separate statements.

Following the program heading is the variable declaration part:

```
var
    i: INTEGER;
    prod: REAL;
```

All variables in a Pascal program must be declared before they can be used. When a variable is declared it is given a name and a type. In Program 1, two variables are declared; i is of type INTEGER, and prod is of type REAL.

Various objects in a Pascal program (such as variables) are named by user-defined identifiers. Identifiers begin with a letter that is followed by any combination of letters, digits and the underscore character (_). They can be any length (up to the length of a source line), and can contain a mixture of upper- and lowercase. However, the case of a letter is ignored, and so is the underscore character. Also, identifiers are distinguished only by their first eight characters (not counting underscores); identifiers that contain the same first eight characters and differ after that are considered the same identifier.

Certain identifiers are reserved for the Pascal language and cannot be defined by the programmer; these are called *reserved words* and appear in **boldface** throughout this book. Many more identifiers are *predeclared*; these have the same status as user-defined identifiers, but are standard to Pascal (or UCSD Pascal). They are available to the programmer, and there is no need to declare them explicitly. Throughout this book, predeclared variables appear in all capital letters and are UNDERLINED.

REAL, INTEGER, and long INTEGER (described in Chapter 2) are the numeric types available in UCSD Pascal. A variable of type INTEGER can take on any integral value between – MAXINT and MAXINT. The MAXINT constant is a predeclared constant in Pascal that can be different in each implementation. In UCSD Pascal, MAXINT equals 32,767.

Real variables can take on a much wider range of values at a cost of precision. A real variable consists of a mantissa part and a scale factor (also called the exponent). The range of values of real variables in UCSD Pascal varies with each implementation. Two-word reals can have a maximum absolute magnitude of 1.0E+38, with 7 digits of precision; for 4-word reals, the maximum absolute magnitude is 1.0E+308, with 16 digits of precision.

The body of Program 1 is the section of text between the first **begin** and the last **end**. The final **end** is followed by a period to indicate the end of the program.

The body of a program consists of a list of statements that are *executed* essentially in order from top to bottom. Some statements cause this straightforward order to change, and are called *flow of control* statements.

Program 1 contains examples of three statement types: the assignment statement, the WRITELN statement, and the **for** statement. Each statement is separated from the following statement by a semicolon (;).

The assignment statement is used to give a value to a variable. For example:

```
prod:= 1.0
```

assigns prod the value 1.0. Further down in the program there is another assignment statement:

```
prod:= prod*i
```

This statement assigns prod a new value that is the product of i and the current value of prod. (The asterisk is used in Pascal to indicate multiplication.)

The assignment symbol in Pascal (:=) differs from the comparison operator (=). Assignment and comparison are two fundamentally different operations.

Another type of statement in Program 1 is the WRITELN statement. The WRITELN statement is used to print data on the user's console. The statement:

```
WRITELN('n factorial of n')
```

causes the text between the apostrophes to be printed. The WRITELN statement can take a list of items to be printed. Each item must be separated from the next by a comma, as in:

```
WRITELN(i,' ',prod)
```

This prints the value of i, followed by a space (for legibility), followed by the value of prod.

(The WRITELN statement is actually a call to a procedure; in this case, a predeclared one. Procedure calls are discussed in Chapter 3.)

Finally, Program 1 contains an example of a **for** statement. A **for** statement causes the statement that follows the **do** to be executed repeatedly for a given number of times. A **for** statement has the form:

```
for< counter> := < first> to < last> do < statement>
```

The index variable < counter> is assigned the value < first> . On each execution of < statement> , the index variable < counter> is replaced by its successor (if it is an integer, it is incremented by 1). When the value of < counter> is greater than the value of < last> , the loop ends.

For example:

```
for i:= 1 to 10 do

WRITELN(i)
```

prints ten lines on the console, numbered one through ten.

It is often the case that a whole group of statements must be executed repeatedly. Pascal allows a group of statements to take the place of < statement> through use of the *compound statement.* A compound statement is a group of statements separated by semicolons and surrounded by a **begin end** pair.

The **for** statement in Program 1 contains a compound statement that is made up of two statements:

```
for i:= 1 to 20 do
  begin
    prod:= prod*i;
    WRITELN(i,' ',prod);
  end
```

The semicolon following the WRITELN statement is optional. However, programs are seldom static entities; they are constantly changed to incorporate new features or to fix bugs, and a programmer frequently needs to insert statements between the last statement of a compound statement and its closing **end.** Typing a semicolon after the last statement eliminates the need to add one later (better yet, it forestalls the bug that would result from forgetting to put one there later!). Thus, typing a semicolon in the first place is a good habit to acquire. All of the sample programs contain this optional semicolon.

It should now be clear how program Fact works. First, a heading is printed, and prod is assigned the initial value of 1.0. Then i takes on the values 1 through 20, and for each value of i, two statements are executed, prod is updated to have the previous value of prod times the current value of i, and then i and prod are printed. As the output listing shows, a table of twenty items is printed. On the left are 1 to 20 (the values of i), and on the right are the factorials of these numbers (the values of prod).

In Program 1, numeric expressions appeared in all three statement types. Pascal expressions appear very much like standard algebraic expressions. For integers, the following operations are available.

+	Addition
—	Subtraction
*	Multiplication
div	Integer division (truncated)
mod	Remainder after division

For reals also, +, —, and * are available. Real division is represented by:

/	Division

Arbitrary expressions can be formed with these operators and numeric constants and variables. As in algebra, the multiplication operators (*, **div**, **mod**, and /) have precedence over the addition operators (+ and —). Parentheses can be used to set apart subexpressions, which are evaluated first.

In general, integers can be used anywhere in real expressions. Type conversion from INTEGER to REAL is performed automatically. However, reals cannot be used with so much freedom. In order to use a REAL in an INTEGER expression, it is necessary to convert the value to an integer by rounding or truncation. Two functions are available for this purpose:

TRUNC(x)	Discards the fractional portion of x
ROUND(x)	Rounds x to the nearest integer

Here are a few assignment statements using the variables prod and i as declared in our program.

```
prod:= (i+6) * prod
i:= i + TRUNC(prod/4.5)
i:= ROUND(prod)
```

As in the preceding example, and as in Program 1, numeric constants (such as 6) can appear in expressions. Integer constants are made up of one or more digits. Real constants must contain a decimal point, a scaling factor, or both. The following assignment contains a real constant with a scaling factor:

```
prod:= 10e-1
```

A scaling factor is similar to *scientific notation*, and should be read as *times ten to the power of* the integer that follows the letter *e* or *E*.

The WRITELN statement can contain a list of items to print. For our present purposes, these items must be expressions that yield a value of type INTEGER, REAL, CHAR (character), or STRING.

The type STRING will be described in more detail later, but for right now, string constants are a very useful way to print text on the console. A string constant is formed by surrounding text by a pair of apostrophes (single quotation marks). String constants must not cross line boundaries. A single quotation character can appear in a string constant by entering two quotation marks at the desired position. For example:

```
'you can''t mean that'
```

is a string constant that contains the word can't.

The next example introduces several new constructs, including input (READLN) and two new flow-of-control statements; the **if**, which is a means of making simple decisions, and the **while**, which is a loop like the **for** statement, but does not have a control variable, and so repeats for some indefinite number of times.

```
program Factors;
  { Computes prime factors of an integer read from
the keyboard}
    var
      n,factor: INTEGER;
  begin
      WRITE('enter number to factor: ');
      READLN(n);
      factor:= 2;
      while n > 1 do
        if n mod factor = 0
          then
             begin
               WRITE(factor,' ');
               n:= n div factor;
             end
          else factor:= factor+1;
      WRITELN;
end.
```

Program 2

This program finds the prime factors of a number. It does this in a rather forceful way. A possible factor is chosen (starting with 2), and if that divides the number then the factor is printed. If the possible factor does not divide the number, it is incremented (by 1), and the search continues. This ensures that factors are printed in increasing order. The loop repeats until the last factor has been divided out (n = 1).

The number to be factored is entered by the user (it is read into the program). In this way, the program achieves some generality; it will factor any integer, not merely integers that were chosen when the program was compiled. Of course, the larger the number, the more time the program will take in figuring its prime factors.

Both input and output are done on the console. Some runs of the program might produce the following output:

```
enter number to factor: 36
2  2  3  3

enter number to factor: 1719
3  3  191

enter number to factor: 210
2  3  5  7

enter number to factor: 16384
2  2  2  2  2  2  2  2  2  2  2  2  2  2
```

Immediately following the program heading is a comment that briefly describes what the program does. Comments are just that; they are meant to be read by the programmer (either the original programmer or someone else), and they are skipped over by the compiler. A comment is any text enclosed in the delimiters (* *) or { }. Comments can cross line boundaries (as the one in the example does), and can appear anywhere in the program, except in the middle of a token such as an identifier or constant.

A comment at the beginning of a program or routine that describes what that program or routine does is always a good idea. A Pascal program is usually more readable if there are not too many comments; it should make use of intelligible identifiers and clear algorithms. Comments can then be judiciously used to explain what a piece of code does, or better yet, why it does what it does.

It is also a good idea, especially in long programs, to accompany variable (and constant) declarations with a comment that indicates the intended use of each variable. This helps prevent abuses of variables when the program is maintained later.

If several variables are of the same type, they can be declared together, as illustrated in the example by:

```
n, factor: INTEGER;
```

The variable names must be separated by commas.

Within the example program's **while** loop, values are printed by a call to WRITE rather than WRITELN. WRITE prints a new value, but does not start a new output line. When a line of output is complete, a new line can be started by the simple call:

```
WRITELN;
```

The effect of this can be seen in the sample output for the program.

The call to READLN allows a value to be read from the console. The READLN function can read REAL and INTEGER values (it can read some other values as well; we will discuss these later). Unlike some languages (such as BASIC), Pascal does not automatically print a prompt when a value is to be read. The programmer should create a prompt appropriate to the situation. Thus, in the example program, the call to READLN is immediately preceded by:

```
WRITE('enter number to factor: ');
```

The main loop of this program is a **while** statement. While a **for** statement continues to execute a statement for a specific number of times, a **while** statement continues indefinitely as long as some condition is met. The **while** statement is thus a simpler, more general construct than the **for** statement.

The general scheme of a **while** statement is:

```
while < Boolean expression>  do < statement>
```

(Remember that the < statement> can be a compound statement).

Each time through the loop, the < Boolean expression> is tested. If it is TRUE, the < statement> is executed, and if it is FALSE, the loop ends.

A < Boolean expression> is often (though not necessarily) a comparison of some sort:

```
n > 1   {as in the sample program}
(n> 1) and not EOLN
(date < = 0) or (date > = 4000)
```

A BOOLEAN value is equal to either TRUE or FALSE. These are operators that can be used in Boolean expressions:

and	Logical and
or	Logical or
not	Logical negation (a unary operator)

These are numeric comparisons as they appear in Pascal:

>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
=	Equals
<>	Not equals

The operands for these comparisons can be any numeric values.

The **while** statement in Program 2 contains a single **if** statement. The **if** statement is a means of making a simple decision. The form of an **if** statement is:

```
if < Boolean expression> then < statement>
```

or:

```
if < Boolean expression>
  then
      < statement 1>
  else
      < statement 2>
```

In the first form, < statement> is executed only if < Boolean expression> is TRUE. In the second form, < statement 1> is executed if < Boolean expression> is TRUE, and < statement 2> is executed if < Boolean condition> is FALSE.

Once again, < statement> can be a compound statement.

The indentation of the sample program indicates our preferred indentation for both **while** and **if** statements. In general, indentation should be chosen to make the program legible, and reflect the lexical nesting of the program. For example, the **if** appears within the **while**, and therefore it is indented more deeply.

We have one further point to make about this program. If you should run it and make a mistake while typing the value of n (such as typing letters instead of numbers), you can be confronted with a message such as this:

```
enter number to factor: v

User I/O error
Segment PASCAL      Proc# 1      Offset# 15
Type < space> to continue
```

A number of things can cause a runtime error; reading an illegal value is one of them. When a runtime error occurs, you must press the space bar before using the system again, and the program must be restarted, or corrected and recompiled. The command U(ser restart can (usually) be used to restart a program that has just been run. Other situations that can result in runtime errors will be mentioned throughout the manual.

The next sample program prints a Pascal's triangle. It introduces the notion of constants, and the form of a structured variable called an array.

```
program Pascal;
    { Pascal prints a Pascal's Triangle of size n
      on the terminal. }
    const
        n = 10;             {size of triangle}
    var
        i,j: INTEGER;
        p: array[1..n] of INTEGER;
    begin
        for i:= 1 to n do
            begin
                p[i]:= 1;
                for j:= i-1 downto 2 do
                    p[j]:= p[j]+p[j-1];
                for j:= 1 to i do
                    WRITE(p[j]:4);
                WRITELN;
            end;
    end.
```

Program 3

A Pascal's triangle is a triangular array, where the elements along the outer edges are equal to 1, and all other elements are equal to the sum of the two elements above them. Readers familiar with probability will recognize these values as the binomial coefficients. The triangle was a discovery of Blaise Pascal. The program's output appears as follows:

```
1
1  1
1  2  1
1  3  3  1
1  4  6  4  1
1  5  10  10  5  1
1  6  15  20  15  6  1
1  7  21  35  35  21  7  1
1  8  28  56  70  56  28  8  1
1  9  36  84  126  126  84  36  9  1
```

The program accomplishes this by an outer loop that is repeated once for each line of the triangle. First the right edge of the line is initialized to 1, then a backward loop generates the elements of the line by summing elements from the previous line. Finally, a forward loop writes the elements of the line (capped off by a single WRITELN).

The size (the depth) of the triangle is declared as a constant:

```
const
  n = 10;   {size of triangle}
```

To create a Pascal's triangle of another size, the constant can be changed. The program would have to be recompiled. Constants can be constant values of type INTEGER, REAL, BOOLEAN, CHAR, or STRING. Note the similarity to **var** declarations, except that an equals sign (=) is used instead of a colon. All **const** declarations *must* precede all **var** declarations.

Also note that the declaration of n is accompanied by a comment that explains its use.

The program does not store the entire Pascal's triangle; only one line at a time. This is done with an **array**, which is a concept you are probably familiar with from other languages. An **array** in Pascal, like a matrix in mathematics, is a table of values grouped together under a single name. Arrays can have one or more dimensions. The array in our program is quite simple:

```
p: array [1..n] of INTEGER;
```

This declares p to be an array with a base type of INTEGER. The bounds of the array are 1 and n. Since n is a constant equal to 10, p contains 10 elements; each of them is an integer.

Note that the upper bound of the array is specified as a constant n (n also appears in the outer **for** loop below). One advantage of using an identifier to name the constant is that when the value of n is changed, no other part of the program need be changed. In Pascal, array bounds must be constants, not variables.

We will encounter multidimensional arrays in Chapter 2.

Within the main body of the program, p refers to the entire array (the program does not use this construct), and p followed by a value in square brackets specifies an individual element of the array. The value in brackets is called a *subscript.* Thus, we could write:

```
p[1]:= 1;
```

or, as in the program itself:

```
p[i]:= 1
```

When a program specifies an array subscript that is out of range, such as:

```
p[0] {... or ...}
p[23]
```

a runtime error occurs, much like the one we illustrated for Program 2. (This does *not* happen if the programmer has turned off range-checking—see Part 1, Chapter 10).

Using variables as subscripts is a very important practice. Our program does this in three statements:

```
p[i]:= 1;
p[j]:= p[j]+p[j-1];
WRITE(p[j]:4);
```

The second **for** statement that appears in the program has the form:

```
for j:= i-1 downto 2 do
```

Note the reserved word **downto** instead of **to**. This is why this loop was described as backward. The loop counter j is DECREMENTED (by 1) rather than incremented. This is the other form of the **for** statement, and is also frequently used. We cannot use a forward loop here, because that would destroy values in p before they were used.

The outer loop of the program contains two **for** loops nested within it. The outer loop uses the index *i*, and the inner loops each use the index *j*. This is a very common way of handling arrays. The important thing to remember is that once a **for** loop has completed its execution, the value of the loop counter is UNDEFINED; before the loop counter variable can be used again, it must be reinitialized. In Program 3, this is accomplished by the second nested **for** statement itself.

In the call to write:

```
WRITE(p[j]:4);
```

the number that follows the array element expression (:4) is called a *field specification,* and indicates that the value should be printed within four spaces. If this specification were not there, the number would be printed without any surrounding spaces, and the Pascal's triangle would be illegible. Numbers are always right-justified within the output field. If the number is too large for the output field (that is, if it were longer than four digits, which would not occur in so small a Pascal's triangle) the field specification is ignored.

It is very important to remember that declarations do not initialize variables. A declaration of an array does *not* initialize the array; all elements of an array are undefined until they have been initialized by a statement within the program.

An attempt to use an undefined (uninitialized) variable is not necessarily detected by the system, but it frequently results in a runtime error, since the space that is allocated to that variable contains *garbage* values left over from previous programs. These variables do not always correspond to what the programmer intends to find there!

The situation can even be a bit more subtle than this. If you look at Program 3, you may notice that the array p contains undefined values all the way up to the last iteration of the outer loop. For each pass through the loop, p[1] through p[i] are defined, while p[i+1] through p[n] are undefined. There is no problem with this, since we never attempt to use the undefined values in an expression or a call to WRITE.

The other important point about Program 3 concerns the very first iteration through the loop. In this case (as is evident from the output), the output line (and hence the defined portion of p) is only one element long. Such a simple case of our loop (and it is necessary for the initialization of the Pascal's triangle) is called a *degenerate case.*

The following steps occur on the first time through the loop:

1. i is set to 1.

2. p[1] is set to 1.

3. The next **for** loop specifies:

   ```
   for j:= i-1 downto 2 do ...
   ```

   since i−1=0, which is already less than 2, the loop is not executed.

4. The next **for** loop specifies:

   ```
   for j:= 1 to i do ...
   ```

   since i=1, this loop is executed only once.

5. WRITELN is called, which ends the output line that contains only the single value p[1]=1.

Something similar occurs in the second iteration, where only two values are printed; again, the first nested **for** loop is skipped. You may wish to work through the steps for this case.

It is always important to pay attention to degenerate cases while programming. In some algorithms, such as this one, they are a necessary part of the problem's solution. In some algorithms, they are not crucial, but can be elegantly handled with no extra code. In some algorithms, degenerate cases are a nuisance and require that the programmer insert additional code to detect them and correct for their effects. This latter sort of algorithm should be avoided if at all possible. In addition to being inefficient (because of the extra test and correction code), an algorithm that must explicitly account for degenerate cases is frequently an algorithm that is not a good solution to the problem at hand, and often contains additional inefficiencies or bugs.

When an algorithm seems to be too complicated, it may be the case that the data structures you are using are not quite appropriate. Remember that data structure and program structure are two sides of the same coin.

The next sample tests a string to see whether it is a palindrome; that is, a string that is the same both forwards and backwards, such as the popular *able was I ere I saw elba.* It introduces the UCSD type STRING, and some new uses of the type BOOLEAN.

```
program Palin_1;
   { Palin_1 tests to see if string s is a
     palindrome (reads the same forwards and
     backwards). }
   var
      s: STRING;
      i: INTEGER;
      is_palindrome: BOOLEAN;
   begin
      WRITE('enter string to test: ');
      READLN(s);
      is_palindrome:= TRUE;
      for i:= 1 to LENGTH(s) div 2 do
         if s[i] <> s[LENGTH(s)+1-i]
            then is_palindrome:= FALSE;
      if is_palindrome
         then WRITELN ('   a palindrome')
         else WRITELN ('   not a palindrome');

   end.
```

Program 4

The program prompts a user to enter a string, and then tests the string to see whether it is a palindrome by comparing the characters in the string, starting at either end and working toward the center. A Boolean variable is used to represent whether the string is a palindrome or not; if any characters are not equal, the variable is set to FALSE. Finally, a message is printed that reports what the program found.

Some runs of the program might look like this:

- Enter string to test: aba
  a palindrome

- Enter string to test: curious
  not a palindrome

- Enter string to test: able was I ere I saw elba
  a palindrome

The program input is contained in a variable of type STRING. A STRING is a sequence of characters. Single characters are represented by the type CHAR, so a STRING is similar to an **array of** CHAR. The difference is that an array must have a fixed length, while a string has a dynamic length that can change during a program's execution.

The individual characters in a string can be indexed by a subscript, just as the elements of an array. The index can range from one up to the dynamic length of the string. This dynamic length can be determined in the program by a call to the intrinsic function LENGTH. In the preceding program, the expression:

LENGTH(s) **div** 2

results in half the length (truncated) of the strings. The program *must* use a construct like this, since the string itself is typed in by the program's user, and we cannot know beforehand how long the string might be.

All strings have a maximum length (also called the *static length*). Unless the program specifies otherwise (Program 4 does not), it is equal to 80 characters. If we wanted to limit the string S to 20 characters, we could have declared it in the following way:

```
s: STRING[20];
```

In a string declaration, if the predeclared word STRING is followed by a number in brackets, that number is the string's maximum length. No string can have a maximum length greater than 255.

The use of the function LENGTH is the first use of an intrinsic function; there are many more of them. A programmer can also write new functions as illustrated in Program 5.

When a string is read, as in the statement:

```
READLN(s);
```

the characters that the user types are placed in s, up to the <return> character. While typing the string, the user can press the **BACKSPACE** key to make corrections. Since the input of a string must be terminated by pressing the **RETURN** key, strings should always be read by a call to READLN, never READ.

We encountered Boolean expressions in Program 3. A Boolean value is equal to either TRUE or FALSE. Program 4 introduces a Boolean variable called is_palindrome.

In the program, is_palindrome is initialized to TRUE, and only set to FALSE if a non-matching pair of characters is encountered. We assume the string to be a palindrome unless proven otherwise. The important comparison is:

```
s[i] <> s[length(s)+1-i]
```

which appears in the first **if** statement.

Boolean values cannot be printed in UCSD Pascal (in standard Pascal they can). Thus, the last statement in our program tests the value of is_palindrome, and prints a message accordingly.

Note that the **for** loop only loops for half of the length of the string. The **div** operation is an integer divide, and truncates its result. If the string has an odd number of characters, the center character is ignored, which is as it should be, since the center character will be the same either forward or backward, and has no effect on whether the string is a palindrome or not.

Also note that we have decided that the null string '' is a palindrome; this keeps our algorithm simple. There is no question that the null string is the same both forwards and backwards.

Here is the palindrome problem again, but this time we have placed the palindrome test in a function. This is our first example of an important technique; breaking a program down into independent modules. This sample also introduces the **repeat** statement, and more about Boolean expressions.

```
program Palin_2;
   var
      sample: STRING;

   function Palindrome(s: STRING): BOOLEAN;
     { Palindrome returns true if s is a palindrome,
       and false if not. }
     var
        is_pal: BOOLEAN;
        i,j: INTEGER;
     begin
        is_pal:= TRUE;          {assume it is a palindrome}
        i:= 1; j:= LENGTH(s);
        while (i < j) and is_pal do
             if s[i] <> s[j]
               then is_pal:= FALSE
               else
                 begin
                       i:= i+1; j:= j-1;
                 end;
        Palindrome:= is_pal;
     end; {Palindrome}
```

```
begin
    repeat
        WRITE('enter string to test: ');
        READLN(sample);
        if Palindrome(sample)
          then WRITELN('   a palindrome')
          else WRITELN('   not a palindrome');
    until sample = '';
end.
```

Program 5

It should be evident that this program is an elaboration on the last program. The code that tests whether a string is a palindrome has been placed in the function palindrome, which returns the result as a value (the test itself has been improved and we will discuss it later).

The main program now has a loop, so instead of testing just one string, we can test any number of strings. The loop ends when the user enters the empty string. This is accomplished by pressing the RETURN key when the prompt *enter string to test:* appears.

Here is the output from a sample run of the program:

```
Enter string to test: 12321
    a palindrome

Enter string to test: drome
    not a palindrome

Enter string to test: YREKA B AKERY
    a palindrome

Enter string to test:
    a palindrome
```

The loop in the main program is our first example of a **repeat** statement. A **repeat** is similar to a **while** statement, but the test for the loop's termination appears at the end of the loop rather than the beginning. This is an important difference, because it means that the statements within the loop are executed at least once, no matter what. The scheme of a **repeat** statement is:

```
repeat
    [< statement> {; < statement> } ]
until < Boolean condition>
```

Notice that there is no need to bracket the statements in a **repeat** loop with **begin** and **end**, since the reserved words **repeat** and **until** accomplish this.

The **repeat** statement is used when (as mentioned), we want the statements to be executed at least once, and when (as in Program 5) the test for termination depends on a value that is assigned within the loop.

It is customary to indent the statements within a **repeat** loop, as illustrated in Program 5.

The main thing to describe about this program is the function itself.

Creating a function (or a procedure) is basically a means of packaging a code so that it can be accessed from several locations in the program, or at several different times during execution (as in the example). Packaging code in this way is an example of *modularity.* Modularity is an important way of structuring programs, by making code reusable in various contexts, and thus keeping code independent and easier to debug.

Two means of packaging code are procedures and functions. The distinctive feature of a function is that it returns a value; thus, a function call must appear within an expression.

In Program 5, the function Palindrome returns a Boolean value, and when it is called, it appears as a Boolean expression within an **if** statement.

Within a sequence of declarations, procedure and function declarations must appear after all variable declarations and before the code body of the program or routine. The form of a procedure or function declaration is similar to the form of a program itself; there is a heading, a list of declarations, and then a body of statements enclosed by a **begin end** pair.

The declarations within a procedure or function appear just like declarations within the main program; they can even include nested functions and nested procedures. All objects declared within a procedure or function are considered *local* to the procedure or function. No code in the program that is outside the procedure or function can use any identifiers that are declared within it.

On the other hand, variables and so forth that were declared in the main program are considered *global* to the procedure or function, and can be used within it. The scope of identifiers is described in more detail in Part 1, Chapter 4.

In Program 5, the function palindrome contains the variables is_pal, i, and j. These are local variables, and cannot be used by the main program. The main program itself contains the variable sample. This is global to palindrome, and could be used within the function (though it is not).

Local variables allow us to hide the workings of a function or procedure from the rest of the program. This too is an important aspect of modularity. Should we decide to change a function (for example, to replace it with a better algorithm), we can do so without affecting any of the program that calls it. It should be apparent that this is useful.

This is the function heading in our sample program:

```
function palindrome (s: STRING): BOOLEAN;
```

Palindrome is the name of the function. BOOLEAN indicates the type of value returned by the function, and s, which is a STRING, is called a *formal parameter.*

A formal parameter is an identifier that takes on the value of the actual parameter that is used when the function is called. In the main program we have:

```
if Palindrome(sample) ...
```

Sample is the *actual parameter* that corresponds to *s*. When palindrome is called, *s* takes on the value of *sample*.

Within a function, formal parameters are treated as local variables. The difference is that each takes its initial value from the actual parameter that corresponds to it. These actual parameters *must* be present when the function is called, and their type *must* correspond to the type of the formal parameters as declared in the function heading.

For the function to return a value, it must contain an assignment statement that assigns a value to the function name itself. Palindrome contains the assignment:

```
Palindrome:= is_pal;
```

Note that both is_pal and the function itself are of type BOOLEAN.

If this assignment to the function's name is absent (or the code skips it for some reason), then the value of the function is undefined, and the results (when the function is called) are unpredictable.

The algorithm used in the function itself is a bit different than the one in Program 4. Instead of one index i, there are two indices, i and j. Variable i is initialized to 1, and j is initialized to LENGTH(s). Instead of the expression:

```
s[i]<> s[LENGTH(s)+1-i]
```

that appeared in the last program, we have:

```
s[i]<> s[j]
```

which is much more readable. On each iteration through the loop, i is incremented and j is decremented.

We are able to do this in Program 5 by changing the condition of the **while** loop to:

```
while (i< j) and is_pal do
```

This is a more complicated Boolean expression. The simple Boolean expressions i< j and is_pal are combined with the operator **and**. The **and** operator is one of the Boolean operators that were introduced with Program 2. These operators have strict precedence when an expression is evaluated, namely:

not	Highest precedence
and	High precedence (equivalent to *, /, **mod**, or **div**)
or	Middle precedence (equivalent to + or −)
>, <, >=, <=, =, <>, and **in**	Low precedence

(**In** is a relational operator that we will encounter when we discuss sets.)

Because of this precedence of operators, we must put parentheses in the expression:

```
(i< j) and is_pal
```

because if we were to write:

```
i< j and is_pal
```

this would be equivalent to:

```
i< (j and is_pal)
```

which is definitely *not* what we intended (it is also illegal, since j is an INTEGER and is_pal is BOOLEAN).

When in doubt, include parentheses in a Boolean expression to make sure it says what you mean, and to make it more readable.

As the programs in this guide (and in your own work) grow more complex, so will the Boolean expressions, but they should never be so long or so obscure as to be unintelligible to someone reading the program.

Another advantage of the algorithm in Program 5 is that the loop ends as soon as a mismatch has been found. In Program 4, the **for** loop tests the entire string, even if is_palindrome has already been set to FALSE.

The last sample program in this chapter takes a year entered in decimal form (such as 1956), and converts it to Roman numerals (such as MCMLVI). This program introduces procedures, subranges, and the type CHAR.

```
program Roman;
    type
        digit = 0..9; {decimal digit}
    var
        n: INTEGER;
    procedure Write_Digit(d: digit; units, fives, tens: CHAR);
        { Write digit d in Roman numerals, using the
         characters units, fives, and tens. }
         var
              i: INTEGER;
         begin
              if d = 9
                 then WRITE(units,tens)
              else if d = 4
                 then WRITE(units,fives)
              else
                 begin
                     if d >= 5
                         then WRITE(fives);
                   for i:= 1 to d mod 5 do
                         WRITE(units);
                 end;
           end; {Write_Digit}

    procedure Write_Date(date: INTEGER);
          { Write date in Roman numerals. Dates not in the
            range 1..3999 are printed as ***. }
         begin
            if (date <= 0) or (date >= 4000)
              then WRITE('***')
            else
              begin
                   Write_Digit(date div 1000,'M','*','*');
                   Write_Digit((date div 100) mod 10,'C','D','M');
                   Write_Digit((date div 10) mod 10, 'X','L','C');
                   Write_Digit(date mod 10,'I', 'V','X');
              end;
         end; {Write_Date}

begin
    WRITE('enter date: ');
    READLN(n);
    WRITE('in Roman numerals: ');
    Write_Date(n);
    WRITELN;
end.
```

Program 6

The main body of the program is quite simple. It reads a single date and prints it in Roman form. You should now be able to convert it so that it works on a series of dates, just as our last sample program worked on a series of strings. In fact, some of our future programming samples will show only the procedures and relevant declarations, and dispense with the program heading and main body, since we are concerned with the algorithms embodied in the procedures and functions, not the program's superstructure, which should be easy for the reader to supply.

The program accepts a year in the range 1 to 3,999, and converts it to Roman numerals. As the comment in the procedure Write_Date points out, dates outside this range are simply printed as '***'. Restricting the range of allowable dates in this way is necessary, otherwise the algorithm we use would become too complicated (longer dates require some non-standard characters).

It is acceptable to restrict the range of allowable inputs to a program, provided the program always tests the input to see that it is valid. There is no way of knowing what the user of a program will enter, so it is always a good idea to test input before trying to operate on it. Bad input should not cause a program to generate bad results.

Here is output from some sample runs of the program:

```
Enter date: 1492
in Roman numerals: MCDXCII

Enter date: 1981
in Roman numerals: MCMLXXXI

Enter date: 1957
in Roman numerals: MCMLVII
```

The procedure Write_Date is the first that is called. It checks the input, and if this is within the accepted range, it then calls Write_Digit four times: one each for the thousands, hundreds, tens, and ones place of the date in decimal form.

Write__Digit is a bit complicated, because a single digit in a decimal numeral can be as long as four letters in a Roman numeral. Every digit in a Roman numeral is either:

- A string of 1..3 'units'
- A 'five' followed by 0..3 'units'
- A 'five' preceded by a single 'unit'
- A 'ten' preceded by a single 'unit'

The actual characters for 'units', 'fives', and 'tens' are passed from Write__Date to Write__Digit when Write__Digit is called. For each decimal place, the characters vary.

It should now be evident how the program works. If you are still uncertain, try working through an example.

The two procedure headings in our program are:

procedure Write__Digit (d: digit; units, fives, tens: CHAR);

procedure Write__Date (date: INTEGER);

As mentioned before, a **procedure** is like a **function**, except that it does not return a value. No type can be specified in a procedure heading, and it is illegal to assign a value to the name of a procedure.

Since procedures do not return a value, they are not called from expressions, but instead from a single statement that consists only of the procedure's name, followed by the list of actual parameters, if there are any. For example, the call to Write_Date is:

```
Write_Date(n);
```

and the calls to Write_Digit are:

```
Write_Digit(date div 1000, 'M', '*', '*');
Write_Digit((date div 100) mod 10, 'C', 'D', 'M');
Write_Digit((date div 10) mod 10, 'X', 'L', 'C');
Write_Digit(date mod 10, 'I', 'V', 'X');
```

We have already seen calls to intrinsic procedures; WRITE, WRITELN, READ, and READLN are all examples of these. Intrinsic routines can often be called with different numbers or types of actual parameters. A user-written routine must always be called with actual parameters that exactly match the formal parameters.

One thing to notice about the calls to Write_Digit is that the actual parameters passed to a procedure can be expressions as well as variables or constants. Of course, the type of the actual parameter must always be the same as the type of the value parameter. (There is another type of parameter called a *variable* parameter; an actual parameter that corresponds to a variable parameter must be a variable. Variable parameters are described in Chapter 2.)

Another thing to notice about Write_Digit is that in the first call, the fives and tens parameters are specified as '*'. The thousands place of the Roman numeral should not contain any fives or tens. The reason we pass these characters is that we must pass something, since actual parameters must match formal parameters. We pass '*', because if by some error the procedure should print out fives or tens in this place, we want the output to look incorrect, so that we know there is a bug somewhere.

The formal parameters units, fives, and tens in Write_Digit are declared to be of type CHAR. CHAR, short for CHARacter, is another simple type like INTEGER, REAL, or BOOLEAN. The calls to Write_Digit have some examples of character constants: 'M', 'C', 'L', 'D', 'X', 'V', 'I', and '*'. Character constants can in fact be any printable character; letters (either upper or lowercase), numerals, and special symbols. A character constant must be enclosed in apostrophes (') to distinguish it from a single-letter identifier or special symbol. A single apostrophe is represented by ''''. It is also possible to declare variables of type CHAR, and assign values to them or read them. There are no operations on characters.

CHAR is the base type of the type STRING, so that a single character in a string (such as s[i]) is of type CHAR.

UCSD Pascal uses the ASCII character set; both printing and non-printing characters are shown in Appendix E. The next chapter discusses characters in greater detail.

In this program, we have declared a **type**:

```
type
    digit = 0..9; {decimal digit}
```

This declaration indicates that the program now has a new type called digit, which can only take the integer values 0 through 9.

The expression 0..9 is the first example of a type that is a *subrange,* that is, a restricted range of some other type; in this case, INTEGER. The type INTEGER is itself an example of a *scalar* type. A scalar type is a simple type whose values are both finite and ordered. In Pascal, it is possible to declare subranges of any scalar type.

So far, the scalar types we have seen are INTEGER, CHAR, and BOOLEAN (BOOLEAN is a scalar type because by definition, TRUE > FALSE). The types REAL and long INTEGER are not scalar. Strings and arrays are structured types, not simple types, so they are not considered scalar either.

For now, the important point is that in Pascal, new types can be declared. The advantage is that the programmer can tailor the structure of data to better match the problem at hand. The following chapter discusses this topic in much more detail.

New types can be associated with an identifier by defining them in a **type** declaration. Type declarations must come after constant declarations and before variable declarations.

In Program 6, the new type digit is used as the type of d, the first parameter to Write_Digit. This has two effects. First, declaring d as a digit makes the program more readable. Second, the compiler normally generates code to check for range errors (as we have seen when using array indices), and should Write_Digit be called with a parameter that is not in the range 0..9, a runtime error will be generated. Thus, declaring a new type buys us some extra protection against programming mistakes.

We have some final observations about Program 6. Since Write_Digit is only called from Write_Date, we could have declared it within Write_Date, making it a local procedure. But it is sometimes clearer not to nest things too deeply, and we feel that the program, as written, is more readable.

Write_Digit is a good example of a procedure that can be used in multiple ways, since it can write a Roman digit for a decimal thousand's place, hundred's place, ten's place, or one's place, all depending on the parameters it is passed.

This is the end of Chapter 1. If you have programmed before, you should now be capable of writing some programs in Pascal. We have touched on the basic tools, but have not gone into great detail about the great variety and flexibility of data types, and the aspects of UCSD Pascal that make it a separate dialect that fits into the p-System. Chapter 2 discusses various data types and how to use them.

2

Data and Expressions

This chapter discusses various Pascal (and UCSD Pascal) data types and ways to use them. We have already seen some types and uses in Chapter 1, and since those are straightforward, we will tend to skim over them and move on to newer topics.

An understanding of the material in this chapter is essential to becoming a versatile Pascal programmer. Unfortunately, we cannot cover all topics with equal depth, so we will refer you back to Part 1 for full details.

EXPRESSIONS AND ASSIGNMENT

We have already seen a large number of expressions in the previous chapter, and can reiterate the following facts about them:

- An expression is code that returns a value of a particular type, including INTEGER, long INTEGER, REAL, CHAR, STRING, BOOLEAN, or a set.
- An expression can include constants, variables, and calls to functions.
- These elements can be combined with operators. Different operations are defined for different data types.
- Numeric expressions are similar to algebraic formulas, and have much the same meaning. In the same way, Boolean expressions are similar to logical formulas.
- The order in which an expression is evaluated depends on the precedence of the operators used; certain operators have higher precedence than others.
- Subexpressions can be grouped together with parentheses in order to alter the order of evaluation.

We have also seen that a variable can be set to a value by an assignment statement, which consists of the name of the variable, followed by :=, and an expression of the appropriate type. This can be used to initialize a variable or to change its value. The new value can be based on the previous value, as in:

```
count:= count + 1;
```

DECLARATIONS

We have also seen that all identifiers used in a program must be declared by the programmer (except for predeclared identifiers such as <u>WRITE</u>). Declarations precede the main body of a program, and must follow a particular order.

The first kind of declaration we have seen is the **const** declaration. Constants are values that cannot be changed. They can be numeric values, Boolean values, characters, or strings. The advantages of using constants are:

- A constant name is easier to read than the value itself.

- If it is necessary to change a constant value, only the declaration need be changed, rather than every occurrence of the value within the program text.

Constant declarations cannot contain expressions, but a numeric constant can be declared as the opposite of a numeric constant that has already been defined, for example:

```
const
        hi_bound = 128;
        lo_bound = -hi_bound;
```

There are also **type** declarations. Type declarations are essential if more than one variable is to be of the new type, or the new type is to be used for routine parameters.

This chapter will go into more detail on the subject of declaring new types.

Next come **var** declarations. The name of each variable is associated with a particular type. Variable declarations cause space to be allocated for each variable, but do not assign it a value. Variables must be initialized before they are used.

Finally, routines can be declared. A routine is a procedure, function, or process. Processes are described in Chapter 5 on concurrency. Each routine has the same general format as the program itself: a heading, a set of declarations, and a main body (enclosed by **begin** and **end**). The declarations within a routine must be in the same order as declarations for the main program, and can include other routines.

The organization and nesting of routines determines the scope of identifiers. Scope is discussed in Chapter 3. First we shall discuss data types themselves, starting with the simple types that we have already seen.

SIMPLE TYPES

In the previous chapter, we have already seen some use of the types **integer**, <u>REAL</u>, <u>BOOLEAN</u>, and <u>CHAR</u>. This section does not go into details about simple types, but presents examples of the use of the types long <u>INTEGER</u>, <u>REAL</u>, and <u>CHAR</u>.

Simple types and the intrinsic routines that handle them are described fully in Part 1, Chapter 3.

Long integers can be used to represent integer values, including values outside the range −<u>MAXINT</u>..<u>MAXINT</u>. They are not considered a scalar type, so they cannot be used where scalar values are required, such as subrange expressions or array indices.

A long integer is declared as an INTEGER with a length attribute in brackets, for example:

```
var lengthy = INTEGER[20];
```

The length attribute represents the maximum number of decimal digits that the value of the long integer variable will contain. Length attributes can be in the range 1..35. This example declares a long integer that can contain up to 20 digits.

Long integers are used much as integers are, but the operation **mod** is not available. The intrinsic function TRUNC can be used to convert a long integer value into an integer value, provided the long integer is in the range −MAXINT .. MAXINT.

As an example of programming with long integers, here is an adaptation of the factorial program from Chapter 1:

```
program Fact_Test;

    type
      long_int = INTEGER[35];

    var
      n: INTEGER;
      fact: long_int;

    procedure Factorial(n: INTEGER; var result: long_int);
      { computes the factorial of n and returns in result }
      var
        i: INTEGER;
      begin
        result:= 1;
        for i:= 2 to n do
              result:= result*i;
      end; {Factorial}
```

```
begin
  WRITELN(' n n!');
  for n:= 1 to 20 do
    begin
      Factorial(n,fact);
      WRITELN(n:2,' ',fact);
    end;
end.
```

Program 7

As in Program 6, this program uses a **type** definition. To pass a parameter of type long INTEGER, the type must be given a user-defined name. A routine heading can contain type identifiers, but not type descriptions.

In addition, a function cannot return a long integer value, so instead we have a procedure with a **var** parameter. If a formal parameter of a procedure is a *variable* parameter, and its value is changed within the procedure, the value of the actual parameter changes as well. For this reason, the actual parameter that corresponds to the formal **var** parameter must be a variable.

Here is the program's output:

```
n  n!
1  1
2  2
3  6
4  24
5  120
6  720
7  5040
8  40320
9  362880
10  3628800
11  39916800
12  479001600
13  6227020800
14  87178291200
15  1307674368000
16  20922789888000
17  355687428096000
18  6402373705728000
19  121645100408832000
20  2432902008176640000
```

To represent fractional values, or numbers of very small or very large magnitude, the type REAL can be used. Because real values are imprecise by nature, they should never be compared with an equals comparison (=). Instead, they should be compared within some tolerance, for example:

```
const epsilon = 1e−8;
...

if ABS(x−y) < epsilon then
```

is more correct than (x=y), if x and y are real values.

Furthermore, the order in which real expressions are evaluated can greatly effect the accuracy of the result. This is illustrated by our next sample program:

```
program Quadratic;

    { This program finds the roots (real or complex) of the poly-
      nomial A*x*x + B*x + C = 0. If A=0, then there are not
      two roots, and an error message is displayed. }

    var
      A,B,C: REAL; {coefficients}
      d: REAL;   {discriminant}
      large_root: REAL;   {real root with largest ABS. value}
      small_root: REAL;   {root with smallest ABS. value}
      real_part: REAL;   {real part of imaginary roots}
      imag_part: REAL;   {imaginary part of imaginary roots}

    begin
        WRITE('enter coefficients a b c: ');
        READLN(A,B,C);
        if A = 0
          then
               WRITELN('does not have two roots')
          else
           begin
               d:= SQR(B) - 4*A*C;
               if d >= 0
                 then       {roots are real}
                    begin
                           WRITELN('roots are real');
                           if B >= 0
                           then large_root:= -(B+SQRT(d))
                           /(2*A)
                           else large_root:= (SQRT(d)-B)/(2*A);
                           small_root:= C/(large_root*A);
                           WRITELN(large_root, ', ',small_root);
                    end
```

```
                  else        {roots are complex}
                   begin
                      WRITELN( 'roots are imaginary');
                      real_part:= -B/(2*A);
                      imag_part:= SQRT(-d)/(2*A);
                      WRITE(real_part, ' + ',imag_part,'i, ',
                      real_part,' - ', imag_part,'i');
                   end;
       end;
end.
```

Program 8

Here is some output from sample runs of the program:

```
enter coefficients a b c: 1 -1 -12
roots are real
4.00000, -3.00000

enter coefficients a b c: 5 2 1
roots are imaginary
-2.00000E-1 + 4.00000E-1i, -2.00000E-1 - 4.00000E-1i

enter coefficients a b c: 0 1 3
does not have two roots
```

The structure of this program is straightforward and by now should be familiar; we will not discuss it. The important thing to observe is that it does *not* use the conventional formulas for finding quadratic roots. The traditional formulas for finding the real roots of the equation A*x*x + B*x + C = 0 are:

```
x1:= (-b+SQRT(d))/(2*a);
x2:= (-b-SQRT(d))/(2*a);
```

The problem with this lies in the nature of real representations. Regardless of the number of digits of accuracy, these are always finite. When two values of similar magnitude are subtracted, there is a danger that precision will be lost (the same happens when a number is added to a value of similar magnitude but opposite sign) because the significant digits cancel each other, and only the digits of low significance (which usually contain roundoff error) remain. This cancellation error can propagate through future calculations.

Thus, if b and SQRT(d) have similar values, the formula for x1 can give drastically poor results.

The program deals with this problem by finding only the *larger* of the two roots in this way:

```
large_root:= -(B+SQRT(d))/(2*A)
```

or:

```
large_root:= (SQRT(d)-B)/(2*A)
```

The smaller root is found by:

```
small_root:= C/(large_root*A)
```

which uses only multiplication and division. These operations cannot cause cancellation errors of the kind we are worried about.

The point of Program 8 is that one should use caution in dealing with real quantities. We will encounter another example of real calculation in Program 18.

The type CHAR is used to represent single characters. In UCSD Pascal, the character set is represented by the ASCII code, which is shown in Appendix E.

In the ASCII code, digits, uppercase letters, and lowercase letters are all in contiguous groups, and in their usual order. This is required by standard Pascal, although the standard does not actually stipulate a particular character code.

The next sample program is actually just a procedure that translates lowercase characters into uppercase:

```
procedure U_Case(var s: STRING);
  {  converts all lowercase letters in s
     to uppercase }
  var
      i: INTEGER;
  begin
      for i:= 1 to LENGTH(s) do
           if (s[i] >= 'a') and (s[i] <= 'z')
                then s[i]:= CHR(ORD(s[i])
                             -ORD('a') + ORD('A'));
  end; {U_Case}
```

Program 9

U_Case accepts a string and translates all of its lowercase characters into their uppercase equivalent.

The algorithm does not assume that we are using ASCII. If it were rewritten to avoid using the type STRING, it would work on any standard Pascal implementation. This generality comes from the (baroque) assignment:

```
s[i]:= CHR(ORD(s[i]) - ORD('a') + ORD('A'));
```

The following information is true:

```
ORD(s[i])
```

is the numerical equivalent of a character in the string (we know it is lowercase because of the if statement).

```
ORD(s[i]) - ORD('a')
```

results in an "offset" into the lowercase alphabet:

```
if s[i]='a', then this offset = 0,
if s[i]='b', then this offset = 1,
... and so forth up to 25.
```

```
ORD('A')
```

is the numerical equivalent of uppercase A.
This is the base of the uppercase alphabet.

```
ORD(s[i]) - ORD('a') + ORD('A')
```

converts to uppercase by adding the offset
into the alphabet (which has nothing to do
with case) to the base of the uppercase alphabet.

```
CHR( ... )
```

converts the whole mess back into a character.

Note that throughout this exercise, we never depended on the exact value returned by the calls to ORD, we simply assumed that the uppercase alphabet and the lowercase alphabet were each contiguous, and in normal alphabetical order.

SCALARS AND SUBRANGES

This section is an introduction to user-defined scalar and subrange types. Scalars and subranges are covered in Part 1, Chapter 3.

The predeclared scalar types (INTEGER, BOOLEAN, and CHAR) have already been introduced. The programmer can also create new scaler types, with declarations similar to these examples.

```
Phase = (new, quarter1, half, quarter2, full)
Grade = (A, B, C, D, E, F)
Knot = (square, granny, hitch, half_hitch, cloverleaf)
```

A declaration of a scalar type is an enumeration of values, enclosed in parentheses. The order of the values is the order in which they are declared.

The advantage of declaring a scalar type is that both the variable and the name of its possible values describe the data on which the program is working.

Because scalar types are ordered, the comparisons =, <>,>, >=, <, and <= apply with their usual meanings.

User-defined scalar types cannot be read or written with the standard Pascal procedures.

The standard functions on scalar types are:

- PRED(v) returns the *predecessor* of v, that is, the value that precedes it

- SUCC(v) returns the *successor* of v, that is, the value that follows it

If v is the first value in the scalar type, then PRED(v) causes a runtime error, and if v is the last value, SUCC(v) causes a runtime error.

The ORD(v) returns an integer that is the ordinal value of v in the list of scalar values

Scalar values are numbered starting at 0. For example, if we use the type phase as declared above, ORD(new) = 0 and ORD(half) = 2.

For user-defined scalars, there is no inverse of the ORD function. For type CHAR, the function CHR is the inverse of ORD, as we have seen.

For an example of a user-defined scalar type, refer to Programs 15 and 16.

We can define a subrange of any scalar type. For example:

```
Lcase = 'a' .. 'z'; {subrange of CHAR}
Mark = 10 .. 20; {of INTEGER}
Not_new = quarter1 .. full; {of phase}
Passing = A .. C; {of grade}
```

The scalar type from which a subrange is derived is called the *base type*. Any input or output, operations, comparisons, procedures, or functions that are defined for the base type are also legal for the subrange, although the overflow conditions for calculations can vary.

Subranges are frequently used as the index type of an array.

Subranges are also useful for the automatic range checking they provide. A subrange was used for this purpose in Program 6.

More examples of subranges will appear in future programs.

ARRAYS

Arrays were introduced in Chapter 1. Our next sample program (actually a program fragment) introduces the notion of **packed**, and an array of multiple dimensions:

```
const
    max= 4;      {adjust for larger matrix size}
```

```
type
    matrix= packed array[1..max,1..max] of BOOLEAN;

procedure Sqr_Matrix(var m: matrix);
    {   squares Boolean matrix m }
    var
      n: matrix;
      i,j,k: INTEGER;
      s: BOOLEAN;
    begin
      n:= m;
      for i:= 1 to max do
        for j:= 1 to max do
          begin
            s:= FALSE;
            for k:= 1 to max do
              s:= s or (n[i,k] and n[k,j]);
            m[i,j]:= s;
          end;
    end; {Sqr_Matrix}
```

Program 10

Program 10 squares a Boolean matrix. This is the same as squaring a numerical matrix, that is, multiplying it times itself, except that + is replaced by the Boolean operator **or**, and * is replaced by **and**.

There are applications for this routine. Each entry of the array, if TRUE, can represent a one-way route from, say, one city to another. Squaring the matrix produces a table of cities that can be reached by a road that passes through exactly one other city.

When a matrix M is squared, each element of the resulting matrix M′ is given by the formula:

```
M'[i,j] = SUM (M[i,k] * M[k,j])
```

or using Boolean elements:

```
M'[i,j] = OR (M[i,k] and M[k,j])
```

SUM and OR are defined over all k, where 1 < = k < = max.

For example, if the matrix to be squared was:

```
1 0 0
0 0 1
1 1 0
```

the result would be:

```
1 0 0
1 1 0
1 0 1
```

Here is a matrix actually passed to Sqr__Matrix:

```
1 0 0 0
0 0 1 1
1 1 0 1
0 1 0 0
```

and the result is:

```
1 0 0 0
1 1 0 1
1 1 1 1
0 0 1 1
```

You should note that operations must be done element-by-element. The two indices into the two-dimensional array are controlled by nested **for** loops. This is a very common way of handling multidimensional arrays.

The word **packed** only appears in the global declaration of the array type *matrix*: it does not appear in the body of the program, and has *no* effect on the algorithm. Packing is a means of reducing the storage space of an array or record. In a Boolean array that is **packed**, each element consists of only one bit, so the array occupies very little space.

STRINGS

The type STRING is another type that we have already encountered. A string is a sequence of characters whose length can change dynamically. This is a UCSD Pascal extension. A number of intrinsic routines are provided to handle strings. See Part 1, Chapter 3.

All of the comparisons can be used on string values; the result returned is based on lexicographic (dictionary) order.

Program 11 consists of two routines that demonstrate some common conversions using strings:

```
function Str_to_Int(s: STRING): INTEGER;
    {  converts s to an integer }
    var
        i, result: INTEGER;
    begin
        result:= 0;
        for i:= 1 to LENGTH(s) do
            result:= result*10 + ORD(s[i]) - ORD('0');
        Str_to_Int:= result;
    end; {Str_to_int}

procedure Int_to_Str(n: INTEGER; var s: STRING);
    {  converts integer n to a string returned as s }
    begin
        s:= '';
        while n > 0 do
        begin
            s:= CONCAT(' ',s);
            s[1]:= CHR(n mod 10 + ORD('0'));
            n:= n div 10;
        end;
    end; {Int__to__Str}
```

Program 11

The way that these two routines operate should be familiar from the uppercase conversion program (Program 9), and the Roman numeral problem (Program 6). Int__to__Str must be a procedure, because a string cannot be returned as a function result (only simple variables can be returned from user-written functions).

The intrinsic LENGTH(S) was introduced in Program 4. It returns an integer value that is the dynamic length of the string S.

The intrinsic CONCAT is a function. It returns a string value that is the concatenation of all the string values passed to it.

Since the parameters to CONCAT must be strings, it was not possible to append the next digit by a call such as:

```
s:= CONCAT(CHR(n mod 10 + ORD('0')),s);
```

Instead, it was necessary to append a *dummy character:*

```
s:= CONCAT(' ',s);
```

and then replace it with the desired character. (In this context, ' ' is treated as a string constant of length 1.)

The intrinsic POS(SOURCE,PATTERN) attempts to match PATTERN to a substring of SOURCE. If it succeeds, it returns an integer that is the index in SOURCE of the first character of the matched PATTERN. If it fails, it returns zero. Program 12 illustrates a way in which POS might be implemented:

```
function Position(s,p: STRING): INTEGER;
  { Finds first occurrence of p in s and returns the
    start character position. Returns 0 if no
    occurrence. }
  var
    a: INTEGER;   {position in s of current search}
    n: INTEGER;   {offset in p of current search}
    same_so_far: BOOLEAN; {TRUE until mismatch}
  begin
    a:= 1; same_so_far:= FALSE;
    while not same_so_far
      and (a <= LENGTH(s)-LENGTH(p)+1) do
      begin
        n:= 0; same_so_far:= TRUE;
        while same_so_far and (n < LENGTH(p)) do
          if s[a+n] = p[1+n]
            then n:= n+1
            else same_so_far:= FALSE;
        a:= a+1;
      end;
    if same_so_far and (LENGTH(p) > 0)
      then Position:= a-1
      else Position:= 0;
  end; {Position}
```

Program 12

The techniques used here are ones we have seen before. Note how the two nested search loops are controlled by a single Boolean variable.

In the last **if**, Position is assigned a−1 rather than a. This is because the outer **while** loop increments a at the end of every pass through the loop. It is simpler to do this than to check the value of same_so_far every time the loop is repeated. If same_so_far ever becomes true, then the loop ends, and a−1 is the solution.

SETS AND RECORDS

This section contains three sample programs that briefly introduce sets and records that are data-structuring constructs that we have not seen before.

A set value is similar to a mathematical set. It is a collection of membership assertions about values from a base type. The base type of a set is a scalar or subrange type. A value of the base type is either **in** the set or not in the set.

Set constants are a list of elements or subranges enclosed in brackets, such as:

```
[0..9]   {the set of digits}
['A'..'Z']   {the uppercase alphabet}
[new, half, full]
     {some values from the type phases}
[9, 3, 5..7]   {some miscellaneous digits}
```

Set values of the same base type can be manipulated with these operators:

+	Set union
−	Set difference
*	Set intersection

and compared with these operators:

=	Equals
<>	Not equals
in	Membership

The **in** operator tests whether a value is in a set. It is illustrated in Program 13.

Sets are described in Part 1, Chapter 3.

The following rewrite of Program 9 illustrates the use of sets for range-checking, and the comparison **in**:

```
procedure Ucase(var s: STRING);
    {converts s to upper-case}
    var
        i: INTEGER;
    begin
        for i:= 1 to LENGTH(s) do
            if s[i] in ['a'..'z']
              then s[i]:= CHR(ORD(s[i])
                           -ORD('a') + ORD('A'));
    end; {Ucase}
```

Program 13

The only difference from the previous Ucase is the line:

```
if s[i] in ['a'..'z']
```

which tests if the character s[i] is in the set of all lowercase letters. This has the advantage of being easier to read than the earlier construction.

The following is an example that also illustrates the use of sets, and contains a very general function that you can be able to use in more than one program (it appears again in Program 22):

```
program Prompt_Test;


    type
        charset = set of CHAR;

    function Prompt(line: STRING;
                         legal_commands: charset): CHAR;
```

```
      {  Prompt prints line, then waits for a
        character in legal_commands to be typed.
        Its uppercase equivalent is returned. }

        var
            ch: CHAR;
        begin
            repeat
                WRITE(line);
                READ(ch);
                WRITELN;
                if ch in ['a'.. 'z']
                   then ch:= CHR(ORD(ch)
                              -ORD('a')+ORD('A'));
    until ch in legal_commands;
    Prompt:= ch;
  end; {Prompt}

begin
    repeat
        case Prompt('G(urgle, W(hir, S(plat, Q(uit: ',
                         ['G','W', 'S','Q']) of
            'G': WRITELN('gurrggle');
            'W': WRITELN('wwhhhirrr');
             'S': WRITELN('sppplaaat');
            'Q': EXIT(program);
        end;
    until FALSE;
end.
```

Program 14

The parameters to Prompt specify two things — the prompt line to be displayed, and the set of characters that correspond to valid commands. It reads characters indefinitely, until a valid command is typed, and then returns that character.

In the main program, the character that Prompt returns is used as the *selector* in a **case** statement. We have not seen the **case** statement before. It has the form:

```
case < selector> of
   < constant list> : < statement>
   ...              : ...
   end
```

The constant lists contain values of a particular scalar or subrange type, and can not overlap. The selector is a value of the same type as the constants in the constant lists. If the value of the selector matches one of the constants, the statement that follows the matching constant list is executed. If the value of the selector does not match any constant, the **case** statement falls through.

Note that the reserved word **end** pairs with the reserved word **case**.

A **case** statement is a valuable way of handling a multi-way branch. If you are a FORTRAN programmer, you can recognize the similarity to an assigned GO TO.

In the set legal_commands that is passed to Prompt, only uppercase characters need be specified. Prompt translates lowercase to uppercase. If you wish to use both lower- and uppercase commands, you can remove the **if** statement from Prompt.

Here is some output from the test program:

```
G(urgle, W(hir, S(plat, Q(uit: g
gurrggle
G(urgle, W(hir, S(plat, Q(uit: w
wwhhhirrr
G(urgle, W(hir, S(plat, Q(uit: G
gurrggle
G(urgle, W(hir, S(plat, Q(uit: s
sppplaaat
G(urgle, W(hir, S(plat, Q(uit: Q
```

A record value is a collection of values. Each value occupies a field of the record, and each field has a name. Different fields in a record can be of different types.

Here is an example of a record declaration:

```
type
    date__rec = record
                    day: day__range;
                    month: month__type;
                    year: year__range;
                end; {date__rec}

var date: date__rec;
```

Note that the reserved word **end** pairs with the reserved word **record**.

There are three fields in this record, each of a different type. Individual fields in a record are referred to by the name of the record variable, followed by a period (.), and the name of the field. For example:

```
date.day:= 15;
date.month:= march;
WRITE(date.year);
```

This sort of notation can get tedious, especially when you have nested records, so Pascal provides the **with** statement. A **with** statement names a record; within the **with**, the field names can be used without their prefixes:

```
with date do
   begin
        day:= 15;
        month:= march;
        year:= 1905;
   end;
```

A record is usually declared as a **type** rather than a **var** (though either is possible). This way, more than one variable can be of the same type of record; the record can even be the base type of an array.

Users of FORTRAN or BASIC can be familiar with creating *parallel arrays,* where data about a set of objects is stored in several arrays of different types. In Pascal, this construct can be simplified by declaring a record type that contains fields for the necessary types, then declaring a single array whose elements are of the record type.

A more complete description of records can be found in Part 1, Chapter 3.

The next sample program fragment illustrates the use of a record and the **with** statement. It is a procedure (with the necessary declarations) that accepts a record that contains a date, and changes it to the following date:

```
type
    day_range = 1 .. 31;
    year_range = 1980 ..2050;
    month_type = (jan, feb, mar, apr, may, jun,
                     jul, aug, sep, oct, nov dec);
    date_rec = record
                    day: day_range;
                    month: month_type;
                    year: year_range;
     end; {date_rec}

var   date: date_rec;

procedure Up_Date(var date: date_rec);

    {  Up_Date increments date to the next calendar day.
     The eventual overflow on the last day of the last
     year is ignored. }
```

```
var
   last_day: day_range;
begin
   with date do
      begin
         case month of
            feb: if (year mod 4 = 0)
                    and (year mod 100 <> 0)
                    then last_day:= 29
                    else last_day:= 28;
            apr,jun,sep,nov: last_day:= 30;
            jan,mar,may,jul,aug,oct,dec: last_day:= 31;
         end;
         if day < last_day
            then day:= day+1
            else
               begin
                  day:= 1;
                  if month < dec
                        then month:= SUCC(month)
                        else
                           begin
                              month:= jan;
                              year:= year+1
                           end;
               end;
      end;

end; {Up_Date}
```

Program 15

Note that the body of the procedure Up_Date is enclosed in a **with** statement. The identifiers day, month, and year used within that **with** are actually fields of the record date.

The algorithm first computes the last day of the current month. If that is equal to the day passed it, it resets day (date.day) to 1, and increments the month and year accordingly; otherwise, it merely increments day.

One further flexibility of field types in the record construct is the ability to create *variants* that allow a field to have more than one format. Here is an example:

```
StockItem = record
              Name: STRING;
              PartNum: INTEGER;
              case InStock: BOOLEAN of
                TRUE: (OnHand: INTEGER);
                FALSE: (Ordered: BOOLEAN;
                        NumOrdered: INTEGER)
          end;
```

In a value of type StockItem, the fields Name and PartNum are always present. InStock is another field. If it is TRUE, then the field OnHand is present and indicates the number of items in stock. If it is FALSE, then the fields Ordered and NumOrdered are both present; the first indicates whether replacements have been ordered, and the second indicates the number of replacements.

The variant field must be the last field in the record. Record variants (also called *case records*) can be used for both *clean* and *dirty* programming purposes; in either case, they must be used with caution. We will not go into detail here; record variants are fully described in Part 1, Chapter 3. A clean use of a variant appears in Program 22, and further examples of their use can be found in Chapter 8.

DYNAMIC VARIABLES AND POINTERS

In Pascal, *dynamic variables* are variables that a program can explicitly allocate and deallocate while it is running. They are usually used to implement linked lists (such as queues), search trees, and other (more or less complicated) data structures. This is accomplished by using pointers to the dynamic variables.

A pointer is usually used as one field of a record that contains other information; the pointer points to other records of the same type, creating a list, a tree, or some other data structure.

The declaration of a pointer binds it to another type:

```
i_ptr, nu_ptr: ^ INTEGER;
```

These pointers can now be used to dynamically reference (nameless) integer variables. Within the program, i_ptr refers to the pointer itself, while i_ptr ^ refers to the variable it points to.

Since the variable that a pointer refers to is *not* declared, the space it occupies must be allocated at runtime. This is done with the intrinsic procedure NEW:

NEW(pointer)	Allocates space for pointer^ (that is, for a variable of the pointer's base type)

After a NEW procedure, it is still up to the programmer to initialize the value of this new variable.

Thus, we could write:

```
NEW(i_ptr);          {Creates an integer variable}
i_ptr^:= 1812;       {Sets it equal to 1812}
nu_ptr:= i_ptr;      {   nu_ptr ^ now equals 1812}
```

In this example, both i_ptr and nu_ptr point to the same integer. But suppose we had written:

```
NEW(i_ptr);
NEW(nu_ptr);
i_ptr^:= 1812;
nu_ptr^:= i_ptr^;
```

In this case, i__ptr and nu__ptr each point to a separate variable and i__ptr is initialized to 1812, as before. The last assignment sets the variable that nu__ptr points to equal to the variable that i__ptr points to. In the first example, there was only one variable. In the second example, there are two.

Pointer values can be assigned, and compared using = and <>, but cannot be operated.

It is possible (and sometimes necessary) to remove a dynamic variable that has been allocated:

DISPOSE(pointer)	Removes pointer ∧ from memory ∧ and sets pointer = NIL

The predeclared word NIL (a reserved word in standard Pascal) represents a pointer to nothing, and is conventionally used to mark the end of a list. A pointer value can be compared to NIL, but an attempt to use it as a reference to a variable causes a run-time error or an operation on garbage data.

The next sample program fragment uses pointers in records, and a user-defined scalar type:

```
type
     sym__kind = (not__found,reserved,
                  predeclared,user__defined);
     sym__rec = record
                   ID: STRING;
                   Kind: sym__kind;
                   LLink,RLink: ∧ sym__rec;
                end; {sym__rec}
var
     tab__head: ∧ sym__rec;
     i: INTEGER;
     s: STRING;
```

```
function Find_Sym(s_id: STRING): sym_kind;
    {   Find symbol s_id in table, and return its kind. }
    var
        found: BOOLEAN;
        p:    sym_rec;
    begin
        p:= tab_head;
        found:= FALSE;
        while (p <> NIL) and not found do
            with p^ do
              if ID = s_id
                  then found:= TRUE
              else if s_id < ID
                  then p:= LLink
              else p:= RLink;
            if found
              then Find_Sym:= p^.Kind
              else Find_Sym:= not_found;
    end; {Find_Sym}

procedure Enter_Sym(s_id: STRING; s_kind: sym_kind);
    {   Enters symbol s_id in table and sets kind to s_kind. }
    var
        inserted: BOOLEAN;
        p,q: ^sym_rec;
    begin
        NEW(q);                 {allocate record}
        with q^ do
            begin               {initialize record}
                ID:= s_id; Kind:= s_kind;
                LLink:= NIL; RLink:= NIL;
            end;
        if tab_head = NIL          {enter record in table}
            then tab_head:= q
            else
```

```
          begin
            p:= tab__head;
            inserted:= FALSE;
            while not inserted do
               with p^do
                 begin
                    if s__id < ID
                       then
                          if LLink = NIL
                           then
                               begin
                                  LLink:= q;
                                  inserted:= TRUE
                               end
                            else p:= LLink
                       else
                          if RLink = NIL
                            then
                               begin
                                  RLink:= q;
                                  inserted:= TRUE
                               end
                            else p:= RLink
                 end;
          end;
   end; {Enter__Sym}
```

Program 16

Enter__Sym enters identifiers into a symbol table, and Find__Sym finds identifiers in the table. The table itself is structured as a binary search tree. Each node of the tree is represented by the record:

```
sym__rec = record
               ID: STRING;
               Kind: sym__kind;
               LLink,RLink:   sym__rec;
           end; {sym__rec}
```

The ID field contains the identifier. The Kind field indicates what sort of an identifier it is. The fields LLink and RLink point to a left and right subtree, respectively. If no such subtree exists, the pointer is equal to NIL.

One global pointer, tab__head, points to the top of the tree. The tree itself is allocated dynamically by Enter__Sym.

Enter__Sym firsts allocates a record for the tree. Then it initializes the record. The identifier and its kind are passed to Enter__Sym as parameters (s__id and s__kind), and the ID and Kind fields of the new record are initialized with these values. Both LLink and RLink are set to NIL.

If tab__head = NIL, then this is the first record in the tree, and tab__head is set to point to it. If tab__head is not NIL, Enter__Sym traverses the tree until it finds a record with a NIL link, and places the new record there by updating the NIL link to point to the new record. As it traverses the tree, it follows either an LLink or an RLink.

Whether Enter__Sym chooses an LLink or an RLink depends on the alphabetical order of the new identifier ('s__id < id'). The tree is a common form of binary search tree. For any node, all the identifiers in the left subtree (alphabetically) precede the identifier at the node, and all the identifiers in the right subtree follow the identifier at the node. Thus, the structure of the tree is based on the order in which identifiers are entered, but when the tree is traversed, the identifiers always appear in alphabetical order. Find__Sym assumes that the tree can be traversed in this way.

Find__Sym is passed an identifier, and returns its kind. It does this by simply traversing the tree until it finds a record whose ID field matches the parameter (s__id). If no identifier matches s__id, Find__Sym eventually reaches a NIL pointer, and halts its search, returning not__found as the symbol kind.

In both routines, note the use of a **with** statement to simplify the code.

The identifier's kind is the sort of information that could be used by a compiler. We use it to illustrate the general technique of associating information with an entry in a table by including that information as a field in the record type.

These routines are derived from the program that we used to format the sample programs before printing them.

3

Flow of Control

We have already encountered the major control constructs; **if**, **case**, **while**, **repeat**, and **for**. This chapter will not go over them, but will briefly deal with the **goto** statement, and the structure of procedures and functions.

Unconditional branching in a program is generally discouraged, because it is confusing to read and difficult to verify. Occasionally it is useful, especially in dealing with emergency situations. In Pascal, the **goto** statement causes an unconditional branch.

A **goto** or similar statement may be a familiar construct in the programming language that you have been using. In Pascal, the variety of flow-of-control statements we have already discussed usually makes the **goto** unnecessary.

Program 17 consists of two code fragments that accomplish the same thing; the second of the fragments uses a **goto**:

```
found:= FALSE;
i:= 0;
while not found and (i < n) do
    if a[i] = '*'
       then found:= TRUE
       else i:= i+1;
if found
    then WRITELN('found at ',i)
    elseWRITELN('not found');
```

. . .this code performs the same task as:

```
label 1;
. . .
for i:= 0 to n-1 do
    if a[i] = '*'
       then
           begin
              WRITELN('found at ', i);
              goto 1;
           end;
WRITELN('not found');
1: {next statement}
```

Program 17

The destination of a **goto** is a labeled statement. Labels are represented by integers in the range 1..9999. Every label that appears in a program must be declared. Label declarations *precede* constant declarations.

In UCSD Pascal, a label must be within the same block as the **goto** that names it (a block is the body of a main program or a routine). Standard Pascal does not have this restriction.

Virtually all Pascal programmers would describe the first example as structured, and second as unstructured. Even though using the **goto** saves us the use of a Boolean variable, the first fragment would be preferable. It is easier to read because of the Boolean, and because the order of execution does not jump about. A program that is easier to read is easier to debug.

FUNCTIONS AND PROCEDURES

To illustrate the use of routines in structuring a program, we have chosen a rather lengthy function. This function, Parse_Group, contains three nested procedures and one nested function.

Parse_Group accepts input that consists of integers or groups of integers, and builds a set that contains them. The input format is relatively loose — we attempt to compensate for the unpredictability of human operators or programmers. Parse_Group is a simple illustration of the form of a *recursive descent* parser. Such parsers are often important parts of compilers or interpreters for high-level programming languages, including the Pascal compiler itself.

Here is the program; it will be discussed in further detail:

```
const
    item_max = 100;

type
    item_range = 1..item_max;
    item_set = set of item_range;

function Parse_Group(var members: item_set): BOOLEAN;
    { Parse a group, which consists of fields separated by
      commas. The set of items selected is
      passed back in item_set. Parse_Group returns TRUE
      only if a legal group was parsed. }
    const
        EOS = '!';
    var
        s,prompt: STRING;
        p: INTEGER;

    procedure Parse_Error(msg: STRING);
        {  Points to the position in the line that the error
           was detected, and prints the error message. }
        begin
            WRITELN('    ': p+LENGTH(prompt),' -- ',msg)
            EXIT(Parse_Group);
      end;   {Parse_Error}

    procedure Skip_Spaces;
        begin
            while s[p] = ' ' do
                p:= p+1;
        end; {Skip_Spaces}
```

```
function Parse_Num: item_range;
   {  Scan off an integer between 1 and item_max, and
     return its value. }
   var
      n: INTEGER;
   begin
         Skip_Spaces;
         if not (s[p] in ['0'..'9'])
            then Parse_Error('expecting number');
         n:= 0;
         repeat
            n:= n*10 + ORD(s[p]) - ORD('0');
            p:= p+1;
            if n > item_max
                then Parse_Error('number out of range');
         until not (s[p] in ['0' ..'9']);
         Parse_Num:= n;
   end; {Parse_Num}

procedure Parse_Field;
   {  Parse a field, and add indicated items to the
     member set, provided there is no duplication. A
     field consists of a number, or number..number. }
   var
       item: INTEGER;
       start_item, end_item: item_range;
   begin
      start_item:= Parse_Num;
      Skip_Spaces;
      if s[p] '.'
         then end_item:= start_item
         else
            begin
               p:= p+1;
               if s[p] <> '.'
```

```
    then Parse__Error ('expecting ''.''');
         p:= p+1;
         end__item:= Parse__Num;
       end;
      if start__item > end__item
       then Parse__Error ('subrange must be in order');
      for item:= start__item to end__item do
      if item in members
         then Parse__Error('duplicate item in list')
         else members:= members + [item];
    end; {Parse__Field}

begin {Parse__Group}
    prompt:= 'enter group: ';
    WRITE(prompt);
    READLN(s);
{   tack on termination character }
    s:= CONCAT(s,EOS);
    Parse__Group:= FALSE;
    p:= 1;
    members:= [ ];

    Skip__Spaces;
    if s[p] <> EOS
       then
          begin
            Parse__Field;
            Skip__Spaces;
            while s[p] <> EOS do
              if s[p] <> ','
                 then Parse__Error('expecting '','''')
                 else
                   begin
                     p:= p+1;
                     Parse__Field;
                     Skip__Spaces;
                   end;
       end;
    Parse__Group:= TRUE;
end; {Parse__Group}
```

Program 18

The legal input to Parse_Group is any list of integers or subranges of INTEGER, provided it does not contain duplicate numbers or bad syntax.

These lists would be accepted by Parse_Group:

```
1,5,10
13,6..9
```

These examples show bad input and the error message printed by Parse_Group:

```
enter group: 7..4
                 — subrange must be in order

enter group: 1,3,5,,7
                   — expecting number

enter group: 2..6,5,8
                   — duplicate item in list
```

Program 18 is the longest example we have seen so far, but it should not be intimidating. In a brief outline, the function:

- Accepts a string from the user
- Tacks a termination character (EOS) onto the string
- Executes a loop that calls:
  - Parse_Field to read a single number or range of numbers
  - Skip_Spaces to scan over any spaces that may intervene before the next entry

The loop ends when the termination character is encountered.

It is the function Parse__Num (called by Parse__Field) that actually converts a character string into an integer; Parse__Field is the procedure that places that number in the set.

The way in which errors are handled is of special interest. Parse__Group is a BOOLEAN function: if it returns FALSE, then parsing has failed in some way.

The procedure Parse__Error is called from within Parse__Group. It prints an error message, and then calls:

```
EXIT(Parse__Group);
```

which aborts the parsing function.

The intrinsic EXIT is a UCSD Pascal extension. When it is passed the name of a routine, the routine returns immediately. It can also be called with:

```
EXIT(< program name> );
```

or:

```
EXIT(program);
```

which causes the program to terminate.

Note that in the main body of Parse__Group, its return value is initialized to FALSE, and is only set to TRUE when parsing is completed and no error has occurred. If it were set before this time, the call to EXIT might cause Parse__Group to incorrectly return TRUE.

Note that Parse__Num and Parse__Field use local variables, while the other nested routines do not. Parse__Error uses only a parameter, and Skip__Spaces uses the string s and the integer p, which are global to Skip__Spaces (but local to Parse__Group).

Program 18 uses sets in two ways. One is for range-checking, a use we have already seen. The other is to build the set "members". This is the main purpose of Parse_Group. The variable "members" is actually used in only two statements; it is initialized to the empty set in the main body of Parse_Group:

```
members:= [ ];
```

and only modified in the last line of Parse_Field:

```
else members:= members + [item];
```

By modifying *members* in only two places, we reduce the chance of error.

RECURSION

Recursion refers to the ability of a procedure or function to call itself. In Pascal this is possible, and can be a most powerful tool for keeping program code brief, elegant, and intelligible.

The limitation of recursion is that there is a certain overhead every time a procedure is called; parameters and addresses must be pushed onto the stack.

The *danger* of recursion is that a procedure or function might go on calling itself until the stack overflows. Our objective is to write programs that terminate cleanly and correctly. As with a **while** or **repeat** statement, a recursive routine must contain code that ensures its termination. In other words, there must be a point at which it does not call itself recursively, but instead returns in a normal fashion.

The scheme of a recursive routine that will terminate might be sketched as:

```
procedure recurse;

    begin
        if < terminating case> then
            {finish task} ...
        else
            begin
                ...
                recurse;
                ...
            end;
    end {recurse};
```

Because of the overhead, recursive routines should only be written to solve problems that are naturally recursive problems that lend themselves to a recursive solution, and that cannot be implemented otherwise without a good deal of overhead (such as an extra programmer- created stack).

Program 19 integrates a curve (finds the area under the curve) by a technique called *adaptive quadrature.* Adaptive quadrature is a naturally recursive algorithm that is relatively fast. In the parlance of numerical analysis, it converges quickly.

In our program, the procedure Adap_Quad merely sets up the problem and then calls Find_Area. It is Find_Area that actually calculates area and calls itself recursively.

Find_Area first attempts to solve the integral using Simpson's rule (this is a common means of calculating an integral by approximating a curve with a parabola). If this result comes within the given tolerance, Find_Area returns it (this is Find_Area's terminating case). If the result is not good enough, Find_Area calls itself again TWICE; once for the left half of the interval, and once for the right half. This is often called *double recursion.*

The effect of Find_Area's algorithm is to spend more time integrating a difficult interval of the curve, and less time on an interval that converges quickly. This is why the algorithm is called *adaptive*. If Find_Area were not recursive, writing an adaptive algorithm would be very difficult (and very messy).

```
program Test;

    const
        min_interval = 0.01;
        { the smallest interval which will
          continue to recurse }

    var
        tol,a,b,area: REAL;

    function Func(x: REAL): REAL;
        { Replace this function with whatever function
          is to be integrated. }
        begin
          Func:= SQRT(x);
        end; {Func}

    function Simpson
                        (a,fa: REAL; var m,fm: REAL;
                         b,fb: REAL): REAL;
      {  Compute the area of interval [a,b] by Simpson's rule.
         The value of the area is returned as the function result,
         and the mid-point (m) and associated function value (fm)
         are returned through var parameters. }
      begin
         m:= (a+b)/2;
         fm:= Func(m);
         Simpson:= (b-a)*(fa+4*fm+fb)/6;
      end; {Simpson}
```

```
function Find__Area
            (tol,aw,a,fa,m,fm,b,fb: REAL): REAL;
  {  Compute the area of interval [a,b] by the method of Adap-
     tive Quadrature. tol is the acceptable tolerance limit for
     this interval. Aw is the area of the whole interval, as com-
     puted by Simpson's method. If the computation is within
     tolerance, then just return. Otherwise, recurse to compute
     the area more accurately, or print a warning that area
     cannot be computed within tolerance with the current
     minimum interval. }
   var
      l,fl,r,fr: REAL;
      { l is mid-point of [a,m],
      r is mid-point of [m,b] }
      al,ar: REAL;
      { areas of left and right intervals }
   begin
      al:= Simpson(a,fa,l,fl,m,fm);
      ar:= Simpson(m,fm,r,fr,b,fb);
      if ABS((aw-al-ar)/aw) < = tol
          then Find__Area:= al+ar
      else if l-a < min__interval
          then
              begin
                  Find__Area:= al+ar;
                  WRITELN
                  ('warning: region [',a,
                  ',',b,'] not in tolerance');
              end
      else Find__Area:=
          Find__Area(tol/2,al,a,fa,l,fl,m,fm)
          + Find__Area(tol/2,ar,m,fm,r,fr,b,fb);
   end; {Find__Area}
```

```
procedure Adap_Quad
                (tol,a,b: REAL; var area: REAL);
   { Compute area bounded by interval [a,b] by setting
     up and calling the recursive Find_Area routine. }
   var
      aw,fa,fb,m,fm: REAL;
   begin
      fa:= Func(a);
      fb:= Func(b);
      aw:= Simpson(a,fa,m,fm,b,fb);
      area:= Find_Area(tol,aw,a,fa,m,fm,b,fb);
   end; {Adap_Quad}

begin
   WRITE('left bound: ');
   READLN(a);
   WRITE('right bound: ');
   READLN(b);
   WRITE('tolerance: ');
   READLN(tol);
   Adap_Quad(tol,a,b,area);
   WRITELN('area is: ',area);
end.
```

Program 19

For the function to be integrated, we have arbitrarily picked Pascal's SQRT intrinsic (we could change Func to calculate some other function, but we would have to recompile the program).

Here is output from some sample runs of the program:

```
left bound: 1
right bound: 5
tolerance: .001
area is: 6.78679

left bound: 0
right bound: 3
tolerance: .1
area is: 3.41141
```

```
left bound: 0
right bound: 3
tolerance: .01
warning: region [ 0.00000, 2.34375E−2] not in tolerance
area is: 3.46405

left bound: 0
right bound: 3
tolerance: .001
warning: region [ 0.00000, 2.34375E−2] not in tolerance
warning: region [ 2.34375E−2, 4.68750E−2] not in tolerance
warning: region [ 4.68750E−2, 7.03125E−2] not in tolerance
area is: 3.46405
```

One advantage of adaptive quadrature is that the function is evaluated only once at any given point; the result is passed on as a value parameter in further calls to Find_Area. Our version of Func is very short, but if Func were a complicated calculation, we would still be able to integrate it relatively quickly.

Note that the tolerance is entered by the user, but is divided by two every time Find_Area calls itself. If we did not improve the tolerance in this way, we would run the risk of converging too quickly. Even with this precaution, we also set a minimum size for the interval we can integrate, and Find_Area terminates on this case as well, after printing a warning that says the value returned may not be accurate.

The min_interval precaution is necessary because we do not know beforehand what Func returns (it may not even be continuous). We therefore give ourselves an escape route so that we do not continue calling Find_Area forever.

The variable names in this program are short and not terribly readable, but we kept them short because a call to Find_Area has so many parameters. If the call were longer than one line of code, it would be less readable, not more so! This was a tradeoff. Note that the names of parameters and local variables in Adap_Quad are the same as the parameters to Find_Area and Simpson. This helps make them comprehensible, because they correspond to the same objects; nevertheless, each is local to its own procedure, and is not global.

4

Input and Output

I/O is concerned with transferring information to and from files. In the p-System, a file is either a file on a block-structured device (such as a disk file), or a peripheral device (such as the console or a printer). This chapter provides some practical illustrations of I/O operations.

UCSD Pascal provides four kinds of I/O: character I/O, record I/O, block I/O, and device I/O. The first two are part of standard Pascal, the second two are UCSD extensions that provide low-level operations. The following is a summary of the four kinds:

- Character I/O
  - Deals with character strings (either numeric or literal)
  - Is sequential
  - Is automatically buffered

- Record I/O
  - Deals with information stored in records on disk files
  - Is either sequential or random-access
  - Is automatically buffered

- Block I/O
  - Deals with blocks (512 bytes) of untyped files
  - Is either sequential or random-access
  - Is not buffered
  - Is used when speed is important

- Device I/O
  - Deals with sequences of bytes
  - Is either sequential or random-access
  - Is not buffered
  - Is used to directly control physical devices (when speed or low-level control is important)

(Random-access is only available on disk files. In the p-System, devices that are not block-structured are handled sequentially.)

The program examples illustrate the syntax for declaring files. The standard Pascal intrinsics RESET and REWRITE can be used to open scratch files. They are more frequently used in a way that is peculiar to UCSD Pascal; associating an internal (Pascal) file name to an external (p-System) file name. Files opened in this way can be either discarded or saved with the UCSD intrinsic CLOSE.

Remember that a physical file on the p-System can be either a disk file or an actual device (for example, CONSOLE:, PRINTER:, REMIN:, REMOUT:). This is described in the *UCSD p-System Operating System Reference Manual*, and summarized in Part 1, Chapter 7. We will illustrate some of this usage in the following examples.

CHARACTER I/O

Character I/O consists of reading and writing information to and from files and character-oriented (serial) devices. We have already seen the intrinsics READ, READLN, WRITE, and WRITELN used to perform I/O from the console. These intrinsics can be used in much the same way when performing I/O from other devices or disk files.

The two sample programs that follow deal with arbitrary files. The user enters the names of the files that are to be used.

```
program lc_filter;
    { This program converts a text file to lowercase. }
    var
        i: INTEGER;
        s: STRING;
        in_file,out_file: TEXT;
```

```
begin
    WRITE('input file: ');
    READLN(s);
    RESET(in_file,s);
    WRITE('output file: ');
    READLN(s);
    REWRITE(out_file,s);
    while not EOF(in_file) do
        begin
            READLN(in_file,s);
            for i:= 1 to LENGTH(s) do
                if s[i] in ['A'.. 'Z']
                    then s[i]:= CHR(ORD(s[i])
                                -ORD('A') +ORD('a'));
            WRITELN(out_file,s);
        end;
    CLOSE(out_file,lock);
end.
```

Program 20

Program 20 converts a file of mixed upper- and lowercase characters to all lowercase. The mechanism that does this:

```
s[i]:= CHR(ORD(s[i])-ORD ('A')+ ORD('a'))
```

should be familiar from Program 14.

As shown, the names of both the input and output files are entered by the user. Both of them are declared as TEXT, which is equivalent to **file of** CHAR. In UCSD Pascal, a **file of** CHAR can be either a serial device, or a .TEXT file on the p-System.

The input file can be either a device or a file that has already been saved on disk. We open it with the call:

```
RESET(in_file, s);
```

The RESET function opens in_file, and associates it with the file named in the string s. (If the user types an unusable filename, the System responds with a runtime error.) RESET makes the file available to the program, and sets the implicit buffer at the beginning of the file, but it does not alter the file's contents.

The output file, on the other hand, is opened with:

```
REWRITE (out_file, s);
```

REWRITE not only makes the file available and restores the buffer to the beginning, as RESET does, it also opens a new (scratch) copy of the file that may supplant the old copy when the file is closed. We presume that the name the user gives for the output file will either be a new file, or an old file that is no longer needed. It could also be an output device.

In both the call to RESET and the call to REWRITE, the second parameter is optional (in standard Pascal, these intrinsics do not have a second parameter at all). When these routines are called without a second parameter, the p-System creates a scratchfile that is deleted when the file is CLOSEd or when the program has finished running. In actual practice, the second parameter is usually used: we are reading from an existing disk file, or creating a new one, or both, as in the sample program.

Note that once we have opened the files we need, READLN and WRITELN are used just as they would be in console I/O, except that their first parameter is the name of a file.

The termination condition of the program's main loop is:

```
while not EOF(in_file) do
```

The intrinsic function EOF stands for End Of File. It returns TRUE after the last record in a file has been read or written. In other words, the program loops until all of in_file has been read.

When a routine or a program terminates, all files that were used during its execution are automatically closed by the p-System. If the files were new (such as scratchfiles), they are deleted. If they already existed on disk, then they are left unchanged.

In our program, the file out_file was new, but since it contains our program's output, we want to keep it. The system's automatic housecleaning can be overridden by a call to the UCSD intrinsic CLOSE. In Program 20, the call that is actually used is:

```
CLOSE(out_file, lock);
```

The parameter lock instructs CLOSE to save the file on disk (under the name used to open it). The CLOSE intrinsic can also be calledwith the parameter purge, which causes the file to be deleted from the directory. (There are other possible parameters to CLOSE detailed in the Part 1, Chapter 7.)

For the file in_file, no call to CLOSE appears, since it is not necessary.

Our next example of character I/O is a program that does a linear regression on a file of numeric data:

```
program Linear_Regression;
   { This program does a linear regression (closest line fit) on
     data in a file. The input file must be a text file with two
     items of data (x,y) on each line. }
   var
      s: STRING;
      data: TEXT;
      n,x,y,y_intercept,slope,
      sumx,sumy,sumxy,sumxsq,temp: REAL;
   begin
      WRITE('data file: ');
      READLN(s);
      RESET(data,s);
      n:= 0; sumx:= 0; sumy:= 0;
      sumxy:= 0; sumxsq:= 0;
      while not EOF(data) do
```

```
        begin
           READLN(data,x,y);
           n:= n+1; sumx:= sumx+x; sumy:= sumy+y;
           sumxy:= sumxy+x*y; sumxsq:= sumxsq=x*x;
        end;
    temp:= n*sumxsq - sumx*sumx;
    y_intercept:= (sumy*sumxsq - sumxy*sumx)/temp;
    slope:= (n*sumxy - sumx*sumy)/temp;
    WRITELN('closest fit: y = ',
                 slope,'x + ',y_intercept);
end.
```

Program 21

Instead of an output file, we write the results directly to the console. The input file is opened just as the input file in Program 20. Like that file, it is declared as TEXT, but in this case each line of the textfile consists of two real constants, separated by one or more spaces.

If data1.text contains:

```
1    1
4    3
7    8.2
5    4
```

the program produces:

```
data file: data1.text
closest fit: y = 1.14933x + -8.34666E-1
```

If data2.text contains:

```
1    2
5    10
3    6
40   80
21   42
```

the program produces:

```
data file: data2.text
closest fit: y = 2.00000x + 0.00000
```

The first column of numbers contains values of x, and the second column contains corresponding values of y. The closest fit to the data is:

```
y = slope * x + intercept
```

where slope and intercept are calculated by the program. The formulas for these are:

```
slope = (n*SUM(x*y) - SUM(x)*SUM(y))
        /  (n*SUM(SQR(x)) - SQR(SUM(x)))

intercept = (SUM(y)*SUM(SQR(x)) - SUM(x*y)*SUM(x))
            /  (n*SUM(SQR(x)) - SQR(SUM(x)))
```

SUM is defined over all x and all y. Therefore, the program keeps a running total for each sum as it reads values from the disk file. When all the values have been read and summed, the slope and intercept are calculated using these formulas.

RECORD I/O

While character I/O is performed on text files and character devices, record I/O is performed on data files. The data is stored in its internal form (for example, integers are 16-bit words, reals are multiple-word floating point representations, and so forth). Because of this, data files are not readable as text (the editor cannot use them, for instance).

Print Queueing

Program 22 illustrates the use of the record I/O intrinsics. It is a long program, and you should not feel obliged to study all of it. We have included it because it illustrates so many practices that are typical of applications and systems programming in UCSD Pascal.

Print__Queuer maintains a file that is a *queue* (a first-in-first-out list) of names of files that are waiting to be printed. We presume that another program would do the actual printing. (This method of building a print queue, and then printing the files either later or concurrently, is known as *spooling*).

The Command Level

Like the p-System itself, Print__Queuer is menu-driven. When it is run, the first thing that appears on the screen is:

```
L(ist, I(nsert, D(elete, C(lear, Q(uit
```

This prompt is controlled by the procedure command:

```
procedure Command;
   var
      done: BOOLEAN;
   begin
      done:= FALSE;
      repeat
         case Prompt('L(ist,I(nsert,D(elete,C(lear, Q(uit ',
                        ['L', 'I', 'D', 'C', 'Q']) of

            'L': List__Command;
            'I': Insert__Command;
            'D': Delete__Command;
            'C': Clear__Command;
            'Q': done:= TRUE;
         end;
```

```
        until done;
      end; {Command}
```

The function prompt was introduced in Program 14. Command allows five different operations. The action of Q(uit should be obvious. The other four are described below. Before explaining them, we describe the structure of the print queue itself.

The Structure of the List

The print queue is contained in a single file:

```
f: file of print_rec;
```

The file contains two lists; a list of allocated records, each of which contains a filename, and a list of free records that are available for allocation. Neither list needs to be contiguous. Each record points to the record that follows it, much as a dynamic list is constructed with pointers. The pointers in f are simply integers that index records in the file, and are used by the intrinsic SEEK.

Each record in the file is of type print_rec, which is a type declared as:

```
print_rec = record
                case rec_kind of
                  info: (alloc_tail,free_tail,
                        last_block_alloc: INTEGER);
                  alloc,free: (name: STRING[30];
                              link: INTEGER);
              end;
```

This is our first full example of a variant record. Each print_rec record can have a group of three integers, or a string of 30 characters followed by an integer.

The form each record takes is determined by the *tag type* rec__kind, which is declared as:

```
rec__kind = (info,alloc,free);
```

In practice, only the *first* record of the file is an info record. All other records are either alloc(ated) or free. No tag variable is needed for the field, since we can tell what type the record is by its context.

The info record contains three integers:

- The alloc__tail integer points to the tail of the list of allocated records. These allocated records are the names of files to be printed.
- The free__tail integer points to the tail of the list of free records. (Both of these values each point to the tail, rather than the head, in order to make inserting a record easier.)
- The last__block__alloc integer points to the last allocated block in the file. This is so we can shrink the file if this record is deleted.

The remaining records in the file simply consist of a string of 30 characters, and an integer that points to the following record in the list. If the record is allocated, then the string is the name of a file to be printed. If the record is free, then the string is never used (it contains garbage).

Clearing the List

The simplest command is C(lear. It calls the following procedure:

```
procedure Clear__Command;
   {  Empties the queue. }
```

```
begin
    f^.alloc_tail:= 0;
    f^.free_tail:= 0;
    f^.last_block_alloc:= 0;
    SEEK(f,0); PUT(f);
    WRITELN;
    WRITELN('    queue cleared');
end; {Clear_Command}
```

The Clear_Command only alters the first (info) record of the file. It may be that the other records in the file contain information, but since the info record does not recognize that, they will be ignored.

Listing the List

The L(ist command prints all of the names in the allocated list. The procedure that does this is:

```
procedure List_Command;
    {  Lists the names of files in the queue in
       reverse order (first file in the queue
       is at the bottom of the list). }
    var
       i: INTEGER;
    begin
       SEEK(f,0); GET(f);
       i:= f   .alloc_tail;
       WRITELN;
       WRITELN('files queued: ');
          while i <> 0 do
             begin
                SEEK(f,i); GET(f);
                WRITELN(' ',f   .name);
                i:= f   .link;
             end;
    end;{List_Command}
```

Refer to the info record to find out the index of the tail of the allocated list, and then traverse the list, printing each file name as you come across it. The link field of the head of the list will be 0, and this terminates the loop. Note that the files are printed in reverse order.

Inserting File Names

The I(nsert command has a much more complicated procedure:

```
procedure Insert_Command;
    { Inserts a file in the queue. If a free record is avail-
      able, it is used. Otherwise a new block is allocated
      at the end of the queue file. }
    var
      s: STRING[30];
      free_block, temp: INTEGER;
    begin
        WRITELN;
        WRITE('   file to insert: ');
        READLN(s);
        SEEK(f,0); GET(f);
        if f^.free_tail <> 0
          then
            begin
                free_block:= f^.free_tail;
                SEEK(f,free_block); GET(f);
                temp:= f^.link;
                SEEK(f,0); GET(f);
                f^.free_tail:= temp;
                SEEK(f,0); PUT(f);
            end
          else
            begin
                last_block:= last_block+1;
                free_block:= last_block;
            end;
        SEEK(f,0); GET(f);
        temp:= f∧.alloc__tail;
        f∧.alloc__tail:= free__block;
        SEEK(f,0); PUT(f);
        f∧.name:= s;
        f∧.link:= temp;
        SEEK(f,free__block); PUT(f);
    end; {Insert__Command}
```

First, the procedure gets the name of the file to be added to the list. It does not check whether this is a valid file name, although that would not be a bad idea — the reader may wish to contemplate how to write a file name-checking procedure.

The info record is then examined to see if there is a free record. If there is, it is removed from the free list, and added to the tail of the alloc list. If there is not, the file is extended by a single record.

Finally, the index of the new record is put in the info block, and both the string and the link are put in the new record. The value of the link is the previous tail of the alloc list.

Note that if the free list is empty, the **if** statement only updates some bookkeeping information. The final PUT automatically extends the length of the file.

Deleting File Names

The D(elete command is a little more involved than I(nsert, but performs the same basic operations:

```
procedure Delete__Command;
    {  Deletes a file from the queue. If the file deleted was
       in the last block of the queue, last__block is decre-
       mented, to shorten the resulting queue. }
```

```
var
    temp,prev,i: INTEGER;
    found: BOOLEAN;
    s: STRING;
begin
    WRITELN;
    WRITE('   delete what file: ');
    READLN(s);
    SEEK(f,0); GET(f);
    prev:= 0; i:= f∧.alloc_tail;
    found:= FALSE;
    while (i <> 0) and not found do
        begin
            SEEK(f,i); GET(f);
            if f∧.name = s
              then found:= TRUE
              else begin prev:= i; i:= f∧.link; end;
      end;
  if found
      then
          begin
              temp:= f∧.link;
              SEEK(f,prev); GET(f);
              if prev = 0
                  then f∧.alloc_tail:= temp
                  else f∧.link:= temp;
              SEEK(f,prev); PUT(f);
              if i = last_block
                  then last_block:= last_block-1
                  else
                      begin
                          SEEK(f,0); GET(f);
                          temp:= f∧.free_tail;
                          f∧.free_tail:= i;
                          SEEK(f,0); PUT(f);
                          SEEK(f,i); GET(f);
                          f∧.name:= '<free>';
                          f∧.link:= temp;
                          SEEK(f,i); PUT(f);
                      end;
```

```
                WRITELN('   file deleted')
            end
         else
            begin
                WRITELN;
                WRITELN('   not found');
            end;
end; {Delete_Command}
```

After accepting a file name, the procedure must search the alloc list for that record. If it is not there, it prints an error message; otherwise, it places that record on the free list (setting the filename to <free> to avoid confusion), and updates the info record accordingly. Note that if the name to be deleted is the last in the file, the variable last_block is simply decremented.

The Main Program

We are now ready to look at the main program:

```
begin
   {$I-}
   RESET(f,'PRINT.FILES');
   if IORESULT <> 0
      then
         begin
            REWRITE(f,'PRINT.FILES');
            Clear_Command;
         end;
   SEEK(f,0); GET(f);
   last_block:= f^.last_block_alloc;
   {$I+}
   Command;
   SEEK(f,0); GET(f);
   f^.last_block_alloc:= last_block;
   SEEK(f,0); PUT(f);
   SEEK(f,last_block_alloc); GET(f);
   CLOSE(f,crunch);
end.
```

The first line of the main program is:

```
{$I-}
```

This is our first example of a *compiler option.* Normally, the compiler generates code that automatically checks all I/O operations. In this case, we want I/O checking turned off so we can use the UCSD intrinsic function IORESULT. When we finish initializing the print queue file, we turn I/O checking back on with:

```
{$I+}
```

The full set of compiler options is described in Part 1, Chapter 10.

We want to turn off I/O checking because we want to be able to maintain the print queue file over several different runs of the program. If the file already exists, we must open it with a RESET (REWRITE would open a new copy). But if the file does not exist, we are required to open it with a REWRITE. The simplest way for the program to tell if the file is already on disk is to try opening it with a RESET. If that fails, IORESULT returns an error number, and we know that we must use REWRITE.

If we did not turn off I/O checking, any error values returned by IORESULT would be intercepted by the program at runtime, and cause the program to halt.

After calling REWRITE, the program calls Clear_Command to initialize the info record.

Once the file has been successfully opened, the last_block variable is initialized, and then the command is called. Command monitors most of the program's work.

We use last__block to optimize the length of the file when it is closed. This is why its value is maintained by the I(nsert and D(elete commands. When the user Q(uit's and Command terminates, last__block is used to update the info record, and then we do a GET on the last__block record. The only purpose of this GET is to set things up for the call to CLOSE:

```
CLOSE(f,crunch);
```

Crunch is a close option we have not seen before. When a file is closed with the crunch option, all records from the last record accessed to the end of the file are deleted. If there are any unused records between last__block and the end of the file, they are deleted when the file is closed. This is a way of saving disk space.

Notice that even though the program is more than 170 lines long, we have only two global variables: f and last__block.

Here is output from a sample run:

```
L(ist, I(nsert, D(elete, C(lear, Q(uit i

   file to insert: QUEUER.LST

L(ist, I(nsert, D(elete, C(lear, Q(uit i

   file to insert: LINREG.OUT

L(ist, I(nsert, D(elete, C(lear, Q(uit l

files queued:
   LINREG.OUT
   QUEUER.LST

L(ist, I(nsert, D(elete, C(lear, Q(uit i

   file to insert: OWL.TEXT
```

```
L(ist, I(nsert, D(elete, C(lear, Q(uit d

   delete what file: LINREG.OUT
   file deleted

L(ist, I(nsert, D(elete, C(lear, Q(uit l

files queued:
   OWL.TEXT
   QUEUER.LST

L(ist, I(nsert, D(elete, C(lear, Q(uit q
```

The Whole Thing

Here is a listing of the entire program:

```
program Print_Queuer;

     type
          charset = set of CHAR;
          rec_kind = (info,alloc,free);
          print_rec = record
          case rec_kind of
              info: (alloc_tail,free_tail,
              last_block_alloc: INTEGER);
              alloc,free: (name: STRING[30];
        link: INTEGER);
          end;
     var
          f: file of print_rec;
          last_block: INTEGER;

     function Prompt
        (line: STRING; legal_commands: charset): CHAR;
     {  Prompt prints a promptline and returns a
        response character in legal_commands. }
     var
```

```
        ch: CHAR;
    begin
        repeat
            WRITELN;
            WRITE(line);
            READ(ch);
            WRITELN;
            if ch in ['a'.. 'z']
          then ch:= CHR(ORD(ch)-
              -ORD('a')+ ORD('A'));
    until ch in legal_commands;
    prompt:= ch;
end; {prompt}

procedure List_Command;
    {   Lists the names of files in the queue in
      reverse order (first file in the queue
      is at the bottom of the list). }
    var
    i: INTEGER;
    begin
      SEEK(f,0); GET(f);
      i:= f^.alloc_tail;
      WRITELN;
      WRITELN('files queued: ');
      while i <> 0 do
        begin
            SEEK(f,i); GET(f);
            WRITELN(' ',f^.name);
            i:= f^.link;
        end;
    end; {List_Command}

    procedure Insert_Command;
        {   Inserts a file in the queue. If a free
          record is available, it is used. Otherwise
          a new block is allocated at the end of the
          queue file. }
        var
            s: STRING[30];
            free_block, temp: INTEGER;
    begin
        WRITELN;
        WRITE('   file to insert: ');
        READLN(s);
        SEEK(f,0); GET(f);
        if f∧.free_tail <> 0
           then
               begin
                       free_block:= f∧.free_tail;
                       SEEK(f,free_block); GET(f);
                       temp:= f∧.link;
                       SEEK(f,0); GET(f);
                       f∧.free_tail:= temp;
                       SEEK(f,0); PUT(f);
              end
           else
              begin
                      last_block:= last_block+1;
                      free_block:= last_block;
              end;
       SEEK(f,0); GET(f);
       temp:= f∧.alloc_tail;
       f∧.alloc_tail:Q= free_block;
       SEEK(f,0); PUT(f);
       f∧.name:= s;
       f∧.link:= temp
       SEEK(f,free_block); PUT(f);
   end; {Insert_Command}

procedure Delete_Command;
{  Deletes a file from the queue. If the file deleted was
   in the last block of the queue, last_block is decre-
   mented, to shorten the resulting queue. }
   var
       temp,prev,i: INTEGER;
       found: BOOLEAN;
       s: STRING;
```

```
begin
    WRITELN;
    WRITE('  delete what file: ');
    READLN(s);
    SEEK(f,0); GET(f);
    prev:= 0; i:= f∧.alloc__tail;
    found:= FALSE;
    while (i <> 0) and not found do
       begin
           SEEK(f,i); GET(f);
           if f∧.name = s
              then found:= TRUE
              else begin prev:= i; i:= f∧.link; end;
       end;
    if found
       then
           begin
               temp:= f∧.link;
               SEEK(f,prev); GET(f);
               if prev = 0
                   then f∧.alloc__tail:= temp
                   else f∧.link:= temp;
               SEEK (f,prev); PUT(f);
               if i = last__block
                   then last__block:= last__block-1
                   else
                       begin
                           SEEK(f,0); GET(f);
                           temp:= f∧.free__tail;
                           f∧.free__tail:= i;
                           SEEK(f,0); PUT(f);
                           SEEK(f,i); GET(f);
                           f∧.name:= '<free>';
                           f∧.link:=temp;
                           SEEK (f,i); PUT(f);
                      end;
               WRITELN('  file deleted')
           end
       else
```

```
              begin
                 WRITELN;
                 WRITELN('   not found');
              end;
end; {Delete_Command}

procedure Clear_Command;
   {   Empties the queue. }
   begin
         f^.alloc_tail:= 0;
         f^.free_tail:= 0;
         f^.last_block_alloc:= 0;
         SEEK(f,0); PUT(f);
         WRITELN;
         WRITELN('   queue cleared');
   end; {Clear_Command}

procedure Command;
   var
      done: BOOLEAN;
   begin
      done:= FALSE;
      repeat
         case Prompt('L(ist, I(nsert, D(elete, C(lear,
            Q(uit',['L', 'I', 'D', 'C', 'Q']) of
            'L': List_Command;
            'I': Insert_Command;
            'D': Delete_Command;
            'C': Clear_Command;
            'Q': done:= TRUE;
         end;
      until done;
   end; {Command}
```

```
begin
   {$I-}
   RESET(f,'PRINT.FILES');
   if IORESULT <> 0
      then
         begin
            REWRITE(f,'PRINT.FILES');
            Clear_Command;
         end;
   SEEK(f,0); GET(f);
   last_block:= f^.last_block_alloc;
   {$I+}
   Command;
   SEEK(f,0); GET(f);
   f  .last_block_alloc:= last_block;
   SEEK(f,0); PUT(f);
   SEEK(f,last_block_alloc); GET(f);
   CLOSE(f,crunch);
end.
```

Program 22

BLOCK I/O

Block I/O is used to transfer large portions of files. The block I/O intrinsics deal with multiples of blocks. To the p-System, a block is 512 bytes. It is the unit of storage on *block-structured* devices such as floppy disks.

Block I/O is intentionally fast, simple, and does little error checking. In other words, it is used when efficiency is important and structure is irrelevant. It follows that block I/O should be used with caution.

Program 23 simply compares two files one block at a time, and prints a message that tells whether they are the same or different:

```
program File_Compare;
    {  File_Compare compares two files
      (of any type) for equality. }
    type
        block = packed array[0..511] of CHAR;
    var
        a_file,b_file: file;
        a_buf,b_buf: block;
        s: STRING;
        same: BOOLEAN;
    begin
        repeat
            WRITE('file a: ');
            READLN(s);
            {$I-} RESET(a_file,s); {$I+}
        until IORESULT = 0;
        repeat
            WRITE('file b: ');
            READLN(s);
            {$I-} RESET(b_file,s); {$I+}
        until IORESULT = 0;
        same:= TRUE;
        while same and not EOF(a_file)
                    and not EOF(b_file) do
            begin
                same:= (1 = BLOCKREAD(a_file,a_buf,1))
                            and (1 = BLOCKREAD(b_file,b_buf,1));
                if same
                        then same:= (a_buf = b_buf);
            end;
        if not same or not EOF(a_file)
              or not EOF(b_file)
            then WRITELN('files different')
            else WRITELN('files same');
end.
```

Program 23

Note that the files are declared without a type. The block I/O intrinsics can handle only untyped files. As the comment at the beginning of the program indicates, the external files can be of any kind.

The third parameter to BLOCKREAD is the number of blocks to transfer. The BLOCKREAD function returns the number of blocks that actually were transferred. If this number is not equal to the parameter, then something went wrong. We infer from this that the files are not the same.

For more information about block I/O, refer to Part 1, Chapter 7.

DEVICE I/O

Device I/O is used for efficiency, or when direct control of a peripheral device is required. The UCSD intrinsics that perform device I/O are all described in Part 1, Chapter 7.

Program 24 copies all of track 0 from one disk to another. On many implementations of the p-System, this track contains the system bootstrap program (and nothing else):

```
program Boot_Copy;

    const
        sectors_per_track = 26;
        bytes_per_sector = 128;
    var
        i: INTEGER;
        buf: packed array[1..bytes_per_sector] of CHAR;
```

```
begin
    WRITELN;
    WRITELN('Bootstrap Copy Program');
    WRITELN;
    WRITELN('      place source disk in drive 4');
    WRITELN('      and destination disk in drive 5');
    WRITELN;
    WRITE ('      press < return>  to continue --');
    READLN;
    for i:= 1 to sectors_per_track do
        begin
            UNITREAD(4,buf,0,i-1,2);
            UNITWRITE(5,buf,0,i-1,2);
        end;
    WRITELN;
    WRITELN('    copy complete');
end.
```

Program 24

The parameters to UNITREAD and UNITWRITE indicate:

- Ahe device number
- The memory buffer
- The number of bytes to transfer (has no meaning when the transfer is in physical sector mode)
- The block number (or sector number, in physical sector mode)
- The mode (2 indicates physical sector mode)

This program will only work if both disks have the same format. The format is *hard coded* in the form of constants. For the program to run with disks of a different format, the values of sectors_per_track and bytes_per_sector would have to be changed.

Please note that this program is shown as a demonstration only, and is not meant to be used with your Texas Instruments Professional Computer.

Concurrency

Code is said to execute concurrently when it runs at the same time as another piece of code. In UCSD Pascal, a routine called a **process** can be run concurrently with the main program and other **processes**. For concurrent execution to truly happen, each process must have its own processor. Since the p-System runs on only one processor, concurrent execution must be simulated. The system runs only one process at a time, but co-ordinates process execution to achieve the appearance of concurrency.

On hardware with multiple processors, concurrency can be a means of speeding up execution. This is not the case with the p-System. Because of the overhead involved in switching processes, using concurrency can even slow a program down somewhat. But concurrent algorithms can greatly improve the conceptual organization of a program, and should not be overlooked. Concurrency can be especially useful for systems programming, I/O handling, and interrupt handling.

Concurrency is not available in standard Pascal. It is described in Part 1, Chapter 9.

A process is not called. An instance of a process is started by a call to the intrinsic START. The same process can be started a number of times; each time, a new instance of the process is begun.

When a process is STARTed, it is given a unique value of the type PROCESSID. The program can examine this value, but cannot change it. A START process is also given a stack of its own on the heap and a priority. The priority is an integer in the range 0..255.

A call to START may specify the size of the process's stack, and the process's priority. The default stack size is 200 words, and the default priority is 128.

Once a START process begins, it either runs to completion, runs until it must wait for another process to do something, or runs until it is interrupted. When one of these events happens, the processor is given to the waiting process with the highest priority.

It is important that a process be able to communicate with other processes in order to synchronize activity and ensure that one process does not interfere with another's operations. Process communication is accomplished with SEMAPHOREs and the intrinsics SIGNAL and WAIT.

A semaphore consists of a count, which is a positive integer value, and a queue of processes that are waiting for that semaphore. If the count equals zero, a process waiting for the semaphore cannot run.

Before a semaphore may be used, it must be initialized by a call to the intrinsic SEMINIT. For example:

```
SEMINIT(my_turn,1);
```

The first parameter is the name of the semaphore, the second is its count.

Once a semaphore has been initialized, two or more processes can use it to co-ordinate their execution. The intrinsic procedures that allow this are:

- WAIT(sem)
  If sem is available (count > 0) then decrement count and keep executing, otherwise wait until the count > 0.

- SIGNAL(sem)
  If count = 0 and a process is waiting, then start another process, otherwise increment count.

The initial value of a semaphore's count can be thought of as the number of resources the semaphore controls. A process WAITing for a semaphore only stops executing if count = 0.

It is common to initialize a semaphore's count to 0 or 1. This implies that only one process at a time can use the semaphore's resource. A semaphore with an initial count of 1 or 0 is often called a *Boolean semaphore.*

One important use of Boolean semaphores is to ensure the mutual exclusion of processes handling one resource. Suppose we have a number of processes that all use the printer. Only one process may do so at a time. We declare a semaphore printer__avail, and initialize it with:

```
SEMINIT(printer__avail, 1);
```

In each process, we bracket the printer-driving code with WAIT and SIGNAL:

```
WAIT(printer__avail);
{code that uses the printer}
SIGNAL(printer__avail);
```

This ensures that no two processes will use the printer at the same time. A portion of code that is protected against interference in this way is called a *critical section.*

Boolean semaphores can also be used to synchronize the activity of two co-operating processes (coroutines, as opposed to subroutines). In the following sample program, there is a process called Player. Player is STARTed twice, and each instance of Player plays one side of a simple two-person game.

```
program Nim;
    const
        stack__size = 2000;
        pile__max = 10;
    type
        pos__int = 0..MAXINT;
        pile__range = 0..pile__max;
    var
        turn__a,turn__b: SEMAPHORE;
        pid__a,pid__b: PROCESSID;
        pile: array[pile__range] of pos__int;
        num__piles: pile__range;
```

```
function Random(lob,hib: INTEGER): INTEGER;
   {  Random returns a random number between
     lob (low bound) and hib (high bound). }
   begin
     Random:= (lob+hib) div 2;
   end; {Random}

functionOdd_Int.(a,b: pos_int): pos_int;
   {  Odd_Int returns the exclusive-or of a and b. }
   begin
     Odd_Int:= ORD((ODD(a) and not ODD(b))
                  or (ODD(b) and not ODD(a)));
   end; {Odd_Int}

   {======= === Player ========== = }

process Player
         (name: STRING; var my_turn,
          your_turn: SEMAPHORE);
  {  Player mimics a player of the game of Nim.
     Player is an expert at the game of Nim, and will force a
     win whenever it is possible to do so. }
  var
     i,p,ones,all,x: pile_range;
     sum,c: pos_int;
     choice: array[pile_range] of pile_range;
  begin
     repeat

       WAIT(my_turn);

       ones:= 0; all:= 0; sum:= 0;
       for i:= 1 to num_piles do
          if pile[i] > 0 then
             begin
                sum:= Odd_Int(sum,pile[i]);
                all:= all+1; choice[all]:= i;
                if pile[i] = 1 then ones:= ones+1;
                if pile[i] > 1 then p:= i;
             end;
```

```
{only one pile non-one}
   if all-ones = 1 then
       if ODD(ones)
          then c:= pile[p]
          else c:= pile[p]-1
   else if sum < > 0 then
{can pick winning move}
       begin
          x:= 0;
          for i:= 1 to num__piles do
              if pile[i] > Odd__Int(sum,pile[i]) then
                 begin
                    x:= x+1;
                    choice[x]:= i;
                 end;
          p:= choice[Random(1,x)];
          c:= pile[p] - Odd__Int(sum,pile[p]);
       end
     else if all > 0 then
{must make random move}
       begin
          p:= choice[Random(1,all)];
          c:= Random(1,pile[p]);
       end;

   if all < > 0 then
       begin
          if (all = 1) and (ones = 1)
           then WRITELN(name,': I lost')
           else WRITELN(name,': pile=',p,',
             count=',c);
          pile[p]:= pile[p] - c;
       end;

   SIGNAL(your__turn);

  until all = 0;
end; {Player}
```

```
begin
   WRITE('enter piles: ');
   num_piles:= 0;
   while not EOLN and (num_piles < pile_max) do
      begin
          num_piles:= num_piles+1;
          READ(pile[num_piles]);
      end;
   SEMINIT(turn_a,1);
   SEMINIT(turn_b,0);
   START(Player('player a',turn_a,turn_b),
     pid_a,stack_size);
   START(Player('player b',turn_b,turn_a),
     pid_b,stack_size);
end.
```

Program 25

The game this program plays is called Nim. The rules of Nim are simple; there are a number of piles (up to ten) of matchsticks (the quantity is represented by a positive integer). At each turn, a player can remove as many matchsticks as he desires from a single pile. The player who is forced to remove the last matchstick loses the game.

The process Player is an expert at Nim. The main program STARTs two instances of this process, and they play against each other. Each instance does its best to win. The turns are coordinated by the semaphores my_turn and your_turn.

The semaphores are needed to ensure that each instance plays in its proper turn. The turns are needed to ensure that the processes' resource, the array of matchstick piles, is not tampered with out of sequence.

The user determines the number of piles and the quantity of matchsticks in each. The first thing the main program does is read this data as a single input line of integers. Then it initializes the appropriate semaphores and starts the two instances of Player. Here is output from a sample run of Nim:

```
enter piles: 7   12   9
player a: pile=1, count=2
player b: pile=2, count=6
player a: pile=3, count=6
player b: pile=2, count=3
player a: pile=1, count=5
player b: pile=2, count=2
player a: pile=3, count=3
player b: I lost
```

Each player waits on my_turn and signals your_turn. Note that these are local to Player, and for the scheme to work, they must be called with the GLOBAL semaphores turn_a and turn_b. The main program must initialize turn_a and turn_b correctly.

A single semaphore would not work; only one process at a time would modify the piles, but there would be no guarantee that the turns alternated. The outcome of the game would be at the whim of the system's process queuing.

The purpose of Odd_Int is to do a Boolean XOR of its two parameters. All the piles are summed in this way; the result is an integer value where a 1 bit represents an odd number of 1's in that column. A winning move is one that makes all of the column sums even.

The player's strategy (which is optimal) is as follows:

- If only one pile contains more than one matchstick, then make an odd number of piles by taking either the whole pile, or all but one.
- Use odd_int to sum the piles. If its result is not 0, then choose a move that makes it 0.
- Make a random move.

Odd_Int performs a trick that we will discuss in Chapter 8. The function Random works, but does not pick a very random number; it could be replaced with a better algorithm.

Handling events (such as interrupts) is another use for semaphores. A semaphore can be bound to a hardware interrupt (or other implementation- defined event) by a call to the intrinsic ATTACH:

```
ATTACH(sem, event_number);
```

The event numbers that you can use on the Texas Instruments Professional Computer can be found in the *UCSD p-System Internal Architecture Guide.*

Once a semaphore has been attached to an interrupt, the semaphore must remain in main memory as long as it remains attached.

Program 26 is a simple example of a semaphore attached to the interrupt from a hardware clock. For the purpose of this example pretend that the appropriate event number is 5:

```
program Clock;

    const
        clock_vector = 5;

    var
        s: SEMAPHORE;
        pid: PROCESSID;

    process tick_tock;
        begin
           repeat
               WAIT(s);
               WRITELN('tick');
               WAIT(s);
               WRITELN('tock');
           until FALSE;
    end; {tick_tock}

    begin
       SEMINIT(s,0);
       ATTACH(s, clock_vector);
       START(tick_tock,pid,500,200);
       repeat
       until FALSE;
    end.
```

Program 26

The process notes the passage of two clock ticks, and prints an appropriate message for each. Since its statements are enclosed in a:

```
repeat
    ...
until FALSE;
```

loop, and since the program does not terminate (it uses the same construct), tick_tock executes indefinitely.

The endless loop construct is actually quite common when writing processes that handle events such as I/O interrupts. We want them to continue doing their job until the program terminates or the system halts.

The only other thing to notice about this program is that we have given tick_tock a larger stack and a higher priority than the defaults. Having a high priority is common for an event-controlled process. When the event occurs, we want the process to do its job as soon as possible, and then go back to sleep.

Units and Separate Compilation

Units in UCSD Pascal are a means of separate compilation. They can be used to:

- Create library packages which can contain both declarations and routines
- Reduce the amount of code that needs to be compiled at one time
- Limit the amount of code that may need to be recompiled during maintenance
- Improve communication when more than one programmer is working on the same project

A good example of all of these advantages is the p-System's operating system, which is an extremely large program consisting of more than 20 units.

For a full description of units, refer to Part 1, Chapter 4. Another example of a unit appears in Chapter 8 as Program 30.

The sample program for this chapter is a small package of routines that handles an integer stack:

```
unit Stack_Ops;

  interface

    procedure Push(n: INTEGER);
    function Pop: INTEGER;
```

```
implementation

    const
        max = 100; {size of stack}
    var
        tos: INTEGER;
        stack: array[1..max] of INTEGER;

    procedure Error(message: STRING);
        {   Prints message on stack overflow
              or underflow. }
        begin
           WRITELN(message);
           EXIT(program);
        end; {Error}

    procedure Push{n: integer};
        {   Pushes n on stack. }
        begin
           if tos > = max
               then Error('stack overflow');
           tos:= tos+1;
           stack[tos]:= n;
        end; {Push}

    function Pop{: integer};
        {   Pops top of stack and returns value. }
        begin
           if tos < 1
               then Error('stack underflow');
           Pop:= stack[tos];
           tos:= tos-1;
        end; {Pop}

    begin
        tos:= 0; {initialize stack}
    end.
```

Program 27

A program that uses Stack_Ops can use Push and Pop as though they were declared within the program. The program cannot use objects declared within the **implementation** part — the constant *max*, the *tos* and *stack* variables, and the procedure Error.

In the **implementation** part, the headings for Push and Pop show the parameter list and function type surrounded by comment delimiters. This is just a memory aid for the programmer and program reader.

The initialization code sets up the stack by setting tos (the top of stack) to zero.

Here is a brief program that uses this unit:

```
program Uses_Unit;

    uses {$U Stack_Ops.CODE} Stack_Ops;

    begin
        Push(4);
        Push(5);
        Push(6);
        WRITELN(Pop,' ',Pop,' ',Pop);
    end.
```

The **uses** declaration must appear immediately after the **program** heading. In a unit that used this unit, the **uses** declaration could appear immediately after the reserved word **interface** or the reserved word **implementation**.

If we were to change the **implementation** part of Stack_Ops (for example, we recompile it with max = 500), we would not need to recompile Uses_Unit. If we changed the **interface** part, on the other hand, Uses_Unit would need to be recompiled.

When a unit is in a particular codefile (a library) other than *SYSTEM.LIBRARY, the program may specify this with the $U compiler option, for example:

```
program Uses_Unit;

{$U STACK.CODE}
uses Stack_Ops;
```

For further information regarding referencing libraries, refer to Part 1, Chapter 4, and the *UCSD p-System Operating System Reference Manual.*

7

Memory Management

Under the p-System, main memory is divided into three resources that all compete for space. These resources are as follows:

- Stack — Used for storage of static variables, expression evaluation, and bookkeeping information for procedure and function calls; grows from high memory toward low memory

- Heap — Used for storage of dynamic variables, and stacks of subordinate processes; grows from low memory toward high memory

- Codepool — Contains code segments; floats between the stack and the heap

When no more main memory space is available, a *stack overflow* error occurs (even if it is the heap that needs more space). This is a fatal error; the system halts and then reinitializes itself. Program results may be lost.

Remember that if you are using extended memory, the codepool occupies a different page (64K-byte area) than the stack and the heap. This helps protect you against stack overflow errors; it also improves the execution speed of the p-System and many programs.

Little can be done to manage stack and heap space; internal system structures already attempt to be space-efficient. One thing a program can do is allocate a data buffer whose size is variable. It can also be helpful to pack records and arrays. For example, the declaration:

```
example: array [1..1000] of BOOLEAN;
```

requires 1,000 words, while the declaration:

```
example: packed array [1..1000] of BOOLEAN;
```

requires only 63 words.

Often, the algorithm that the programmer chooses determines how efficient the program will be, in terms of both time and space.

In the codepool, the unit of code is the segment. Only the segment that is currently executing need be in memory, and the programmer can take care that other segments are swapped out and do not occupy space that may be needed for data.

In general, a single program compilation creates a single code segment. Likewise, a unit occupies a single segment. The programmer can place a routine in a code segment of its own by preceding the routine heading with the reserved word **segment**. For example:

```
segment procedure memory__hog;

segment function save__space (size: INTEGER): INTEGER;

segment process seldom;
```

Since a segment routine must be read into memory whenever it is needed, routines that are declared as **segments** should be routines that are rarely called (perhaps just once per execution, or not during every execution). Initialization and termination routines are good candidates for this.

The programmer can further control the residence of segments by using the following UCSD intrinsic procedures:

- MEMLOCK(seg__names)
  Seg__names is a string of segment names, separated by commas; segments that are named cannot be swapped out

- MEMSWAP(seg__names)
  Allows the segments named in seg__names to again be swapped out to disk

MEMLOCK can speed up a program, by forcing a segment that is frequently used to remain in main memory. Unless MEMLOCK is used, any segment is swappable. The programmer should be careful to call MEMSWAP once a locked segment is no longer needed; otherwise, it will remain in memory and can cause a stack overflow.

The name of a segment is the name of the program, unit, or routine (only the first eight characters are used).

When a program must allocate a large data buffer, but is intended to run on many different machines whose memory requirements will differ, it is possible to make the buffer variable in size by using the intrinsics VARVAIL and VARNEW. This is illustrated by the next program:

```
program Myprog;

    const
        res__segs = 'myprog,fileops,pascalio';
        slop = 2000;
    type
        byte = 0..255;
        large__buf = array[0..32000] of INTEGER;
    var
        buf: ^ large__buf;
        buf__size: INTEGER;
```

```
begin
    buf__size:= VARAVAIL(res__segs) - slop;
    if VARNEW(buf,buf__size) = 0
       then Error('problem in allocating buffer');

    {  A buffer of buf__size words has been allocated.
       buf^[0] through buf  [buf__size - 1] may
                now be accessed. }

 ...

end.
```

Program 28

The intrinsic function VARAVAIL is passed a list of segment names. It returns the number of words of memory that would be available if main memory were to contain both the segments named in the list, and any other segments that have already been MEMLOCKed.

In Program 28, VARAVAIL is passed the string myprog, fileops,pascalio. Myprog is the program itself. FILEOPS and PASCALIO are operating system segments that are frequently called. If the buffer we wish to allocate were to prevent any of these segments from being loaded into memory, the program would have to halt with a stack overflow.

The buffer is allocated the number of words returned by VARAVAIL, plus a slop of 2,000 words. The slop allows for other uses of memory such as data space for the rest of the program, operating system overhead that we cannot predict, processes that might compete with Myprog, and so forth.

The intrinsic function VARNEW(POINTER, COUNT) simply does a NEW(POINTER) on count words. It returns the number of words allocated. If it cannot allocate the full count words, it returns zero.

We provide for a very large data buffer by the declaration:

```
large__buf = array[0..32000] of INTEGER;
```

Note that this must be a type declaration. If it were a variable declaration, the entire area would be allocated. Instead, we declare:

```
buf: ∧ large__buf;
```

This allows us to allocate as much or as little as we want.

Note that we must reference the array as a dynamic variable, for example:

```
buf∧[0]:= 1;
```

8

Advanced Techniques

This chapter describes some techniques that are more powerful than the (relatively) straightforward programming practices we have already discussed. These techniques are crucial to systems programmers and often to programmers of large applications. In general, they provide more efficient ways of solving certain problems. Some of them might be considered dirty tricks.

The chief danger of these techniques is that they allow the programmer to write code that is almost unintelligible. This defeats most of the reasons for using Pascal in the first place! Another danger is that there is little error checking, so the programmer must be quite cautious. Finally, because these techniques usually depend on some detail of implementation, they are not likely to be portable to other implementations of Pascal.

CHARACTER INTRINSICS

The UCSD intrinsics MOVELEFT, MOVERIGHT, FILLCHAR, and SCAN are provided for manipulating arrays of characters—often large ones. In fact, they do no type checking on their arguments, so they can be used with any type of data. They are often used in conjunction with the SIZEOF intrinsic.

For example, FILLCHAR can be used to quickly initialize an entire array:

```
FILLCHAR(A,SIZEOF(A),0);
```

This would fill the array A with zeroes.

In another example, MOVELEFT (or MOVERIGHT) can be used to assign a value of one type to a value of any other type:

```
type
  I: INTEGER;
  TWOBYTES: packed array [0..1] of 0..255;
...
MOVELEFT(I,TWOBYTES,2);
```

This would assign the integer I to TWOBYTES, which is a pair of bytes. This would allow the program to examine the high byte and the low byte of the integer individually.

This technique requires both caution and a knowledge of how the data types are represented internally (see Part 1, Chapter 3).

Further examples of these intrinsics appear in Program 30.

RECORD VARIANTS

Variant fields of a record can be used to convert data from one type to another. This can be useful when Pascal does not support a particular type conversion. A byte or a word can be constructed out of individual sub-fields. This is especially useful when the full word is to be used on the machine (or P-machine) level (for example, the word is used as a single instruction or a memory address).

Program 29 is a procedure that shows a variant field used to construct a hexadecimal value from an integer:

```
procedure Write_Hex(n: INTEGER);
   {  Writes n as 4 hex digits. }
   var
      i: INTEGER;
      hex: packed array[0..15] of CHAR;
      both: record case BOOLEAN of
               TRUE: (w: INTEGER);
               FALSE: (h: packed array[1..4] of 0..15);
            end; {both}
   begin
      hex:= '0123456789ABCDEF';
      with both do
         begin
            w:= n;
            for i:= 4 downto 1 do
               WRITE(hex[h[i]]);
         end;
   end; {Write_Hex}
```

Program 29

The variant field describes a memory location as both an integer (16 bits), and a packed array of four 4-bit fields.

Each 4-bit field is directly converted to a hex digit by using its value to index the array *hex*.

MEMORY ADDRESSES

Pointers are implemented as 16-bit memory addresses. By using them in a record variant as previously described, they can be made to point at anything in memory. This is an important technique in systems programming. Its pitfalls should be apparent.

Program 30 is a screen-handling unit for a computer that has a memory- mapped screen. The screen buffer is located at 1000H. Each byte in the buffer represents a character on the screen. All we need to do is update the buffer, and the hardware refreshes the screen. The screen displays 24 * 80 characters, so the program represents the buffer as an array of the same size.

```
unit Screen_Routines;

    interface

        procedure Clear_Screen;
        procedure Set_Cursor(r,c: INTEGER);
        procedure Write_Ch(ch: CHAR);
        procedure New_Line;
        procedure Back_Space;
```

```
implementation

    const
        row__max = 23;
        col__max = 79;
    type
        screen__map = packed array
            [0..row__max,0..col__max] of CHAR;
    var
        screen: ^ screen__map;
        row, col: INTEGER; {cursor position}

    procedure Clear__Screen;
        {   Fills screen with spaces and homes cursor. }
        begin
            FILLCHAR(screen^, SIZEOF(screen__map),' ');
            row:= 0; col:= 0;
        end; {Clear__Screen}

    procedure Scroll__Screen;
        { Scrolls screen up one line. }
        begin
            MOVELEFT(screen^[1,0],
            screen^[0,0],row__max*(col__max+1));
            FILLCHAR(screen^[row__max,0], col__max
            +1,' ');
             row:= row__max;
         end;
```

```
procedure Set_Cursor{r,c: integer};
    {   Sets random cursor position. }
    begin
        row:= r;
        col:= c;
    end; {Set_Cursor}
procedure Write_Ch{ch: char};
    {   Writes character on screen. }
    begin
        screen∧[row,col]:= ch;
        col:= (col+1) mod (col_max+1);
        if col = 0 then
            begin
                row:= (row+1) mod (row_max+1);
                if row = 0
                    then Scroll_Screen;
            end;
    end; {Write_Ch}

procedure New_Line;
    {   Performs carriage-return and
        line-feed on screen. }
    begin
        col:= 0;
        row:= (row+1) mod (row_max+1);
        if row = 0
            then Scroll_Screen;
    end; {New_Line}

procedure Back_Space;
    {   Moves cursor back one space on line }
    begin
        if col > 0
            then col:=col-1;
    end; {Back_Space}
```

```
procedure Init__Screen;
    {  Sets screen to point to buffer, and clears screen. }
    var
        mem__ptr: record case BOOLEAN of
                        TRUE: (i: INTEGER);
                        FALSE: (p: ∧ screen__map);
                  end;
    begin
        mem__ptr.i:= 4096; {address of buffer = 1000H}
        screen:= mem__ptr.p;
        Clear__Screen;
    end;

begin
    Init__Screen;
end.
```

Program 30

The only place in this unit where we use tricks with pointers is in the procedure Init__Screen. The record variant is used to make an absolute address (the field i) equivalent to a pointer (p). The pointer to the screen buffer (screen) is then initialized to the value of p.

In the procedure Clear__Screen, we use FILLCHAR to quickly fill the entire screen with blanks. In the procedure Scroll__Screen, we call MOVELEFT to move the screen up one line, and then call FILLCHAR to set the last line to blanks. If we wanted a procedure that scrolled the screen down one line, we could use MOVERIGHT in a manner similar to MOVELEFT.

The character intrinsics manipulate the array much faster than the usual assignments within **for** loops would. They are ideal for situations, like screen display, where speed is truly important.

ORD(ODD)

Since Booleans are represented as 16-bit quantities, and since comparison operators only test the low-order bit of a Boolean value, the ODD intrinsic actually does nothing more than allow an integer to be treated as a Boolean. In a similar fashion, the ORD intrinsic merely allows its parameter to be treated as an integer, since this is the internal representation of all scalar types.

These facts can be useful, because the operators **and**, **or**, and **not** actually do logical operations on full words; each bit is set appropriately.

The combination of the ODD function and the Boolean operators allow bit-wise operations on integer values.

For example, the following expression has the effect of masking I down to its four low-order bits. The ORD intrinsic allows the result to again be treated as an integer.

```
I:= ORD(ODD(I) and ODD(15));
```

We used this technique in Program 25 (and promised to explain it later) because our strategy for Nim required us to perform a bit-wise XOR on integers:

```
function Odd_Int(a,b: pos_int): pos_int;
   {  Odd_Int returns the exclusive-or of a and b. }
   begin
      Odd_Int:= ORD((ODD(a) and not ODD(b))
                         or (ODD(b) and not ODD(a)));
   end; {Odd_Int}
```

The expression is complicated because there is no single operator that performs a bit-wise exclusive-or.

CONCLUSION

This is the end of the Programmer's Guide. While we hope that it has helped you, we hope that it is not the end of your study.

Among the many books available, one recommended most highly is Niklaus Wirth's *Algorithms + Data Structures = Programs* (Englewood Cliffs, NJ: Prentice-Hall, Inc., 1976).

It seems that the most successful and innovative software (including languages and operating systems) has been invented, not with the grandiose aim of solving all programming problems for all time, but with the modest aim of solving a particular problem in the most elegant way possible. Pascal itself was invented for the purpose of teaching programmers. In light of this, we encourage you to apply your skills toward problems that interest you, and to approach them in a spirit of craftsmanship.

Lexical Standards

THE CHARACTER SET

The letters A..Z, a..z, the digits 0..9, the special characters:

( ) [ ] { } + – * / < = > : ; , ' _

as well as blanks (' ') and < return> .

The other printable characters are:

! @ # $ % & ? | \ " ` ~

SPECIAL SYMBOLS

. , ; : ' ( ) [ ] { } + – * / = < > ^

:= .. <= < > >= (* *) ***

RESERVED WORDS

An asterisk indicates reserved words not in standard Pascal.

and	goto	record
array		repeat
	if	
begin	*implementation	*segment
	in	*separate
case	*interface	set
const		
	label	then
div		to
do	mod	type
downto		
	not	*unit
else		until
end	of	*uses
*external	or	
		var
file	packed	
for	procedure	while
forward	*process	with
function	program	

IDENTIFIERS

- Can contain letters, digits, or the underscore (_)
- The first character must be a letter
- The underscore is ignored
- Uppercase and lowercase are equivalent
- Only the first eight characters determine uniqueness
- Cannot cross a line boundary

COMMENTS

- Are delimited by { } or (* *)
- Delimiters cannot be mixed
- Comments with the same kind of delimiter cannot be nested
- Comments with different kinds of delimiter can be nested
- A comment can be longer than one line
- A comment with $ immediately after the left delimiter indicates a compiler option

PREDECLARED IDENTIFIERS

An asterisk indicates predeclared identifiers not in standard Pascal.

ABS
ARCTAN
*ATAN
*ATTACH
*INSERT
*BLOCKREAD
*BLOCKWRITE

BOOLEAN

CHAR
CHR
*CLOSE
*CONCAT
*COPY
COS

*DELETE
DISPOSE

EOF
EOLN
*EXIT
EXP

FALSE
*FILLCHAR

GET
*GOTOXY

*HALT

*IDSEARCH
INPUT
INTEGER
INTERACTIVE
*IORESULT

*KEYBOARD

*LENGTH
LN
*LOG

*MARK
MAXINT
*MEMAVAIL
*MEMLOCK
*MEMSWAP
*MOVELEFT
*MOVERIGHT

NEW
NIL

*ODD
ORD
OUTPUT

PAGE
*PMACHINE
*POS
PRED
*PROCESSID
PUT
*PWROFTEN

READ
READLN
REAL
*RELEASE
RESET
REWRITE
ROUND

*SCAN
*SEEK
*SEMAPHORE
*SEMINIT
*SIGNAL
SIN
*SIZEOF
SQR
SQRT
*START
*STR
*STRING
SUCC

TEXT
*TIME
*TREESEARCH
TRUE
TRUNC

*UNITBUSY
*UNITCLEAR
*UNITREAD
*UNITSTATUS
*UNITWAIT
*UNITWRITE

*VARAVAIL
*VARDISPOSE
*VARNEW

*WAIT
WRITE
WRITELN

UCSD Pascal Syntax

COMPARISONS

Comparisons are operators that return the type BOOLEAN.

Comparisons on ordered types:

=	Equal to
<>	Not equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to

Ordered types include all numeric types, all scalar and subrange types, the type STRING, and **packed array of** CHAR.

For BOOLEAN values, these comparisons may be interpreted as:

=	Equal to
<>	Not equal to or XOR (exclusive or)
<=	Implies
>	Does not imply
>=	Is implied by
<	Is not implied by

Comparisons on unpacked **records** or **arrays** of the same type and dimensions:

=	Equal to
<>	Not equal to

Comparisons on sets:

=	Equal to
<>	Not equal to
>=	Is a superset of
<=	Is a subset of
in	Membership

OPERATIONS

INTEGER operations:

+	Addition
−	Subtraction
*	Multiplication
div	Integer division
mod	Remainder after division

The second operand of a **div** or **mod** cannot be zero.

*, **div**, and **mod** have precedence over + and −.

REAL operations:

+	Addition
−	Subtraction
*	Multiplication
/	Real division

The second operand of a / (slash) cannot be zero.

The * (asterisk) and / (slash) have precedence over the + and − (plus and minus operations).

Long INTEGER operations are the same as for INTEGER, except that **mod** is not allowed.

BOOLEAN operations:

not	Negation (a unary operator)
and	Conjunction
or	Union (inclusive or)

not has precedence over **and**, which has precedence over **or**.

Set operations:

+	Union
−	Intersection
*	Difference

The * (asterisk) has precedence over the + and − (plus and minus) operation.

STATEMENTS

A statement can be a null statement. Among other things, this accounts for the extra semicolon at the end of a compound statement or the statement list in **repeat**.

```
assignment−statement = variable−name ":=" expression

case−statement =
        "case" expression "of"
            constant−list ":" statement
            { ";" constant−list ":" statement } [";"]
        "end"

compound−statement =
        "begin" statement { ";" statement } "end"

for−statement =
        "for" var−id ":=" start−value "to" stop−value
           "do" statement
        |
        "for" var−id ":=" start−value "downto" stop−value
           "do" statement
```

goto-statement = "**goto**" label

if−statement =
    "**if**" Boolean−expression "**then**" statement
    ["**else**" statement]

procedure−call = procedure−name ["(" parameter−list ")"]

repeat−statement =
    "**repeat**"
        statement { ";" statement }
    "**until**" Boolean−expression

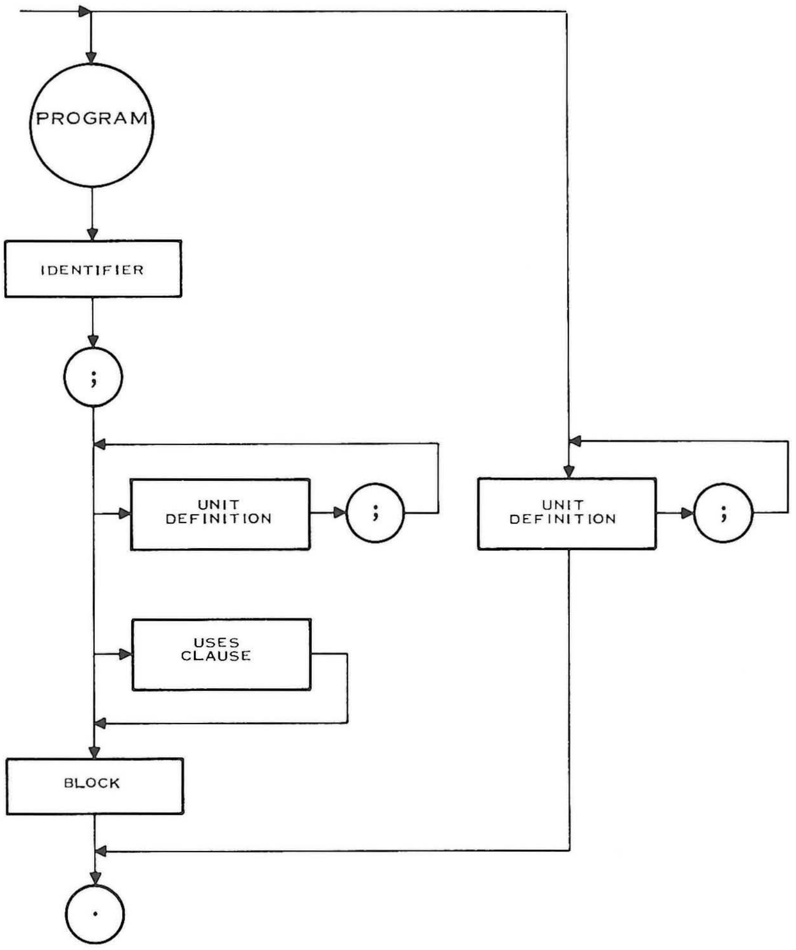
while−statement =
    "**while**" Boolean−expression "**do**" statement

with−statement =
    "**with**" record-id-list "**do**" statement

RAILROAD DIAGRAMS

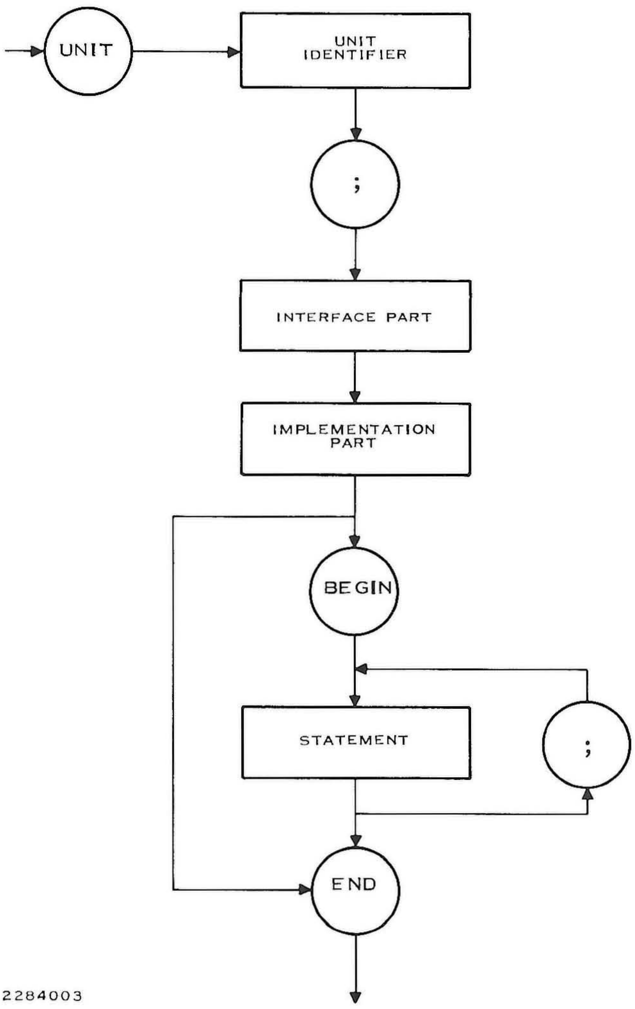
These figures are a concise representation of UCSD Pascal syntax.

<COMPILATION>



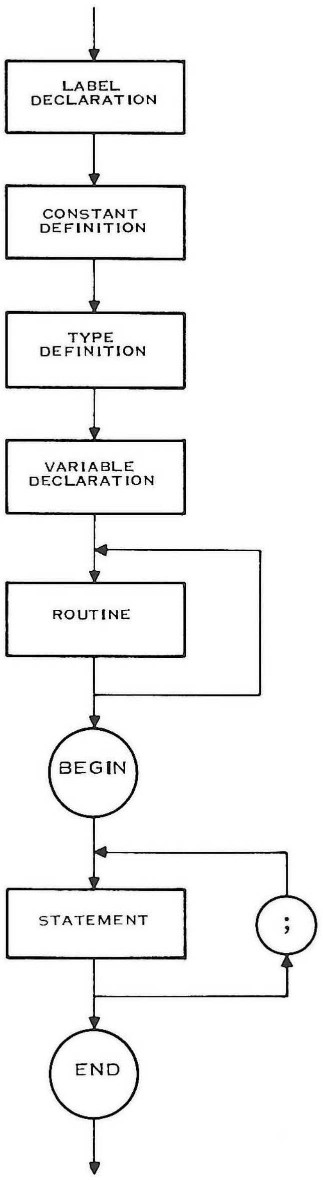
2284002

<UNIT DEFINITION>


UNIT
UNIT IDENTIFIER
;
INTERFACE PART
IMPLEMENTATION PART
BEGIN
STATEMENT
;
END

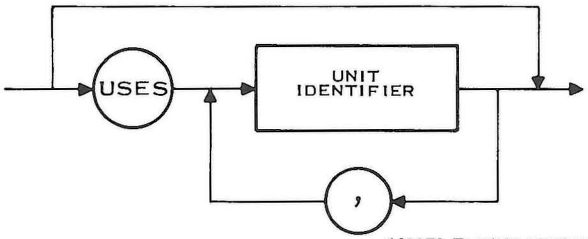
2284003

<BLOCK>

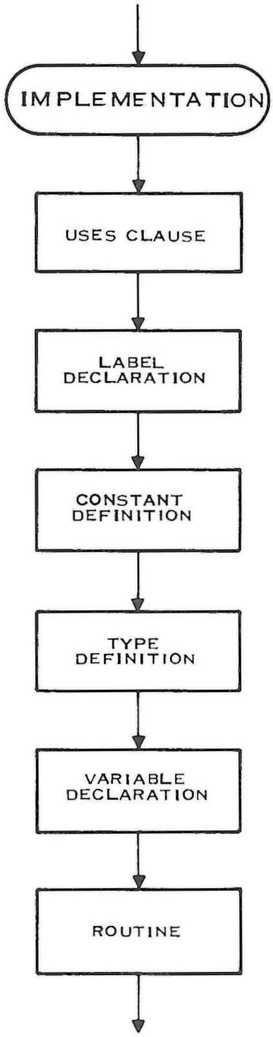

LABEL DECLARATION
CONSTANT DEFINITION
TYPE DEFINITION
VARIABLE DECLARATION
ROUTINE
BEGIN
STATEMENT
;
END

2284004

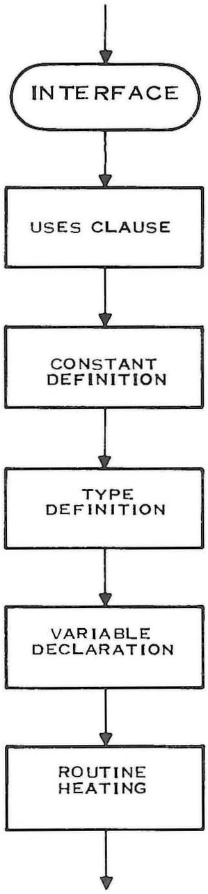
<USES CLAUSE>



<IMPLEMENTATION PART>

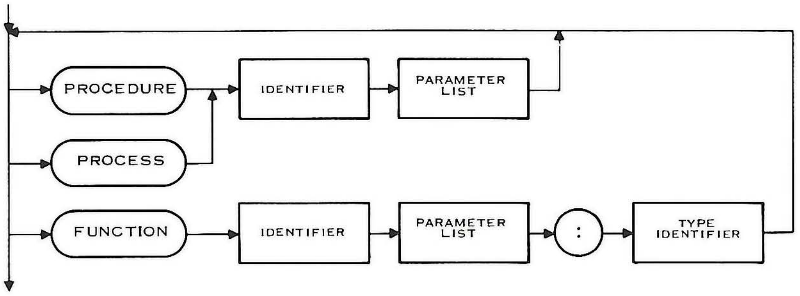


<INTERFACE PART>

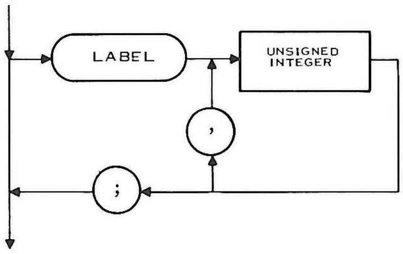


2284005

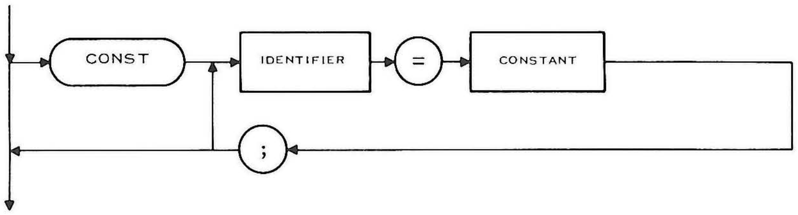
<ROUTINE HEADING>


PROCEDURE
PROCESS
FUNCTION
IDENTIFIER
PARAMETER LIST
IDENTIFIER
PARAMETER LIST
:
TYPE IDENTIFIER

<LABEL DECLARATION>

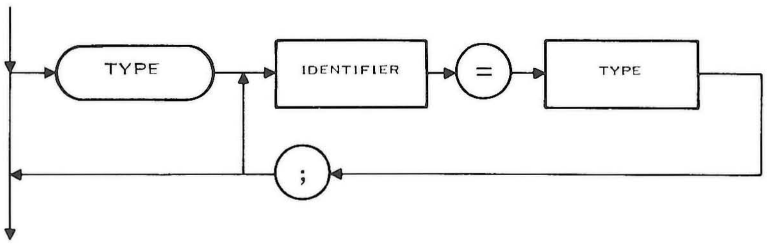

LABEL
UNSIGNED INTEGER
,
;

<CONSTANT DEFINITION>

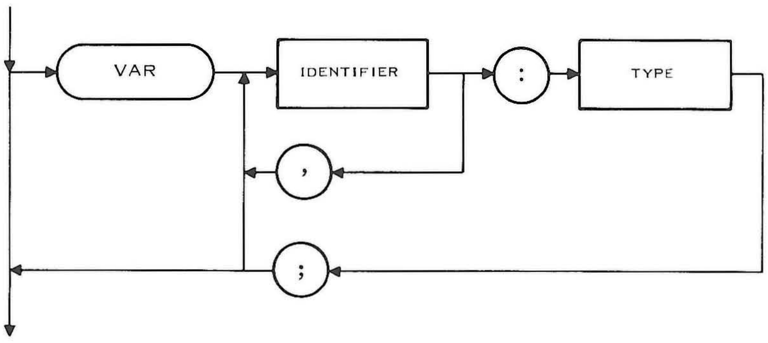

CONST
IDENTIFIER
=
CONSTANT
;

2284006

<TYPE DEFINITION>

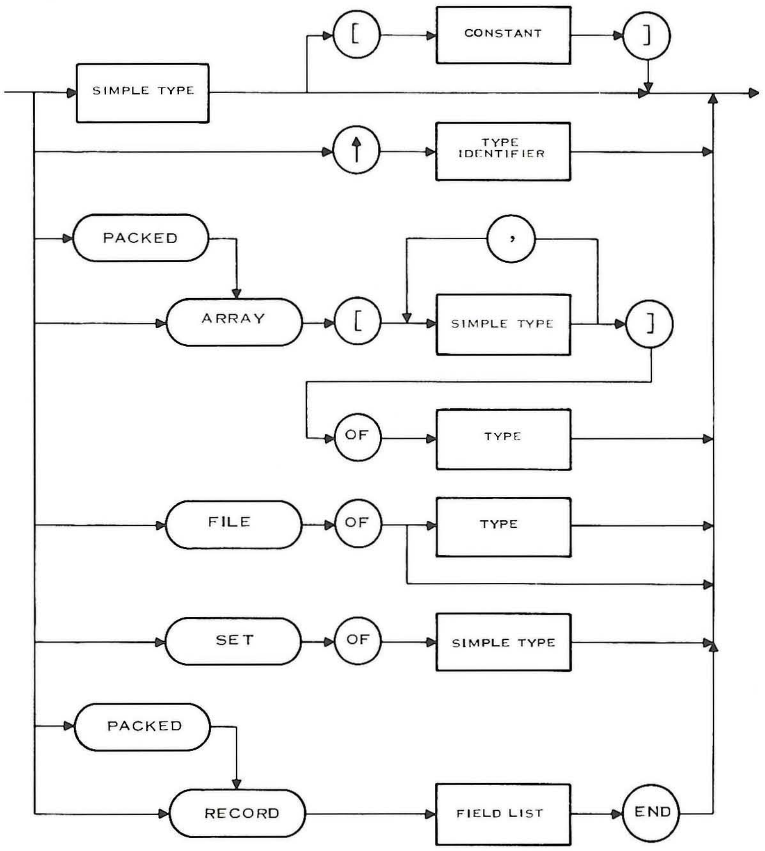


<VARIABLE DECLARATION>



2284007

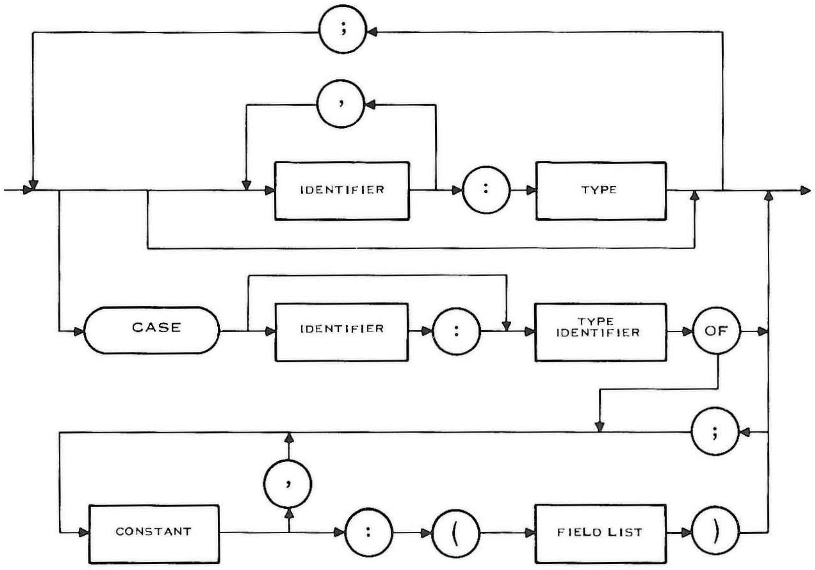
<TYPE>



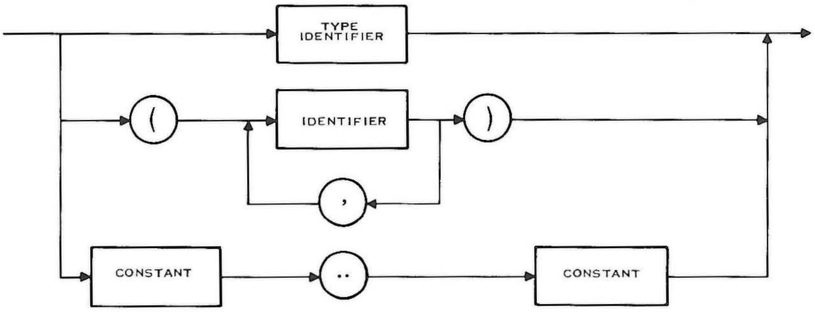
SIMPLE TYPE
[
CONSTANT
]
↑
TYPE IDENTIFIER
PACKED
ARRAY
[
,
SIMPLE TYPE
]
OF
TYPE
FILE
OF
TYPE
SET
OF
SIMPLE TYPE
PACKED
RECORD
FIELD LIST
END

2284008

<FIELD LIST>

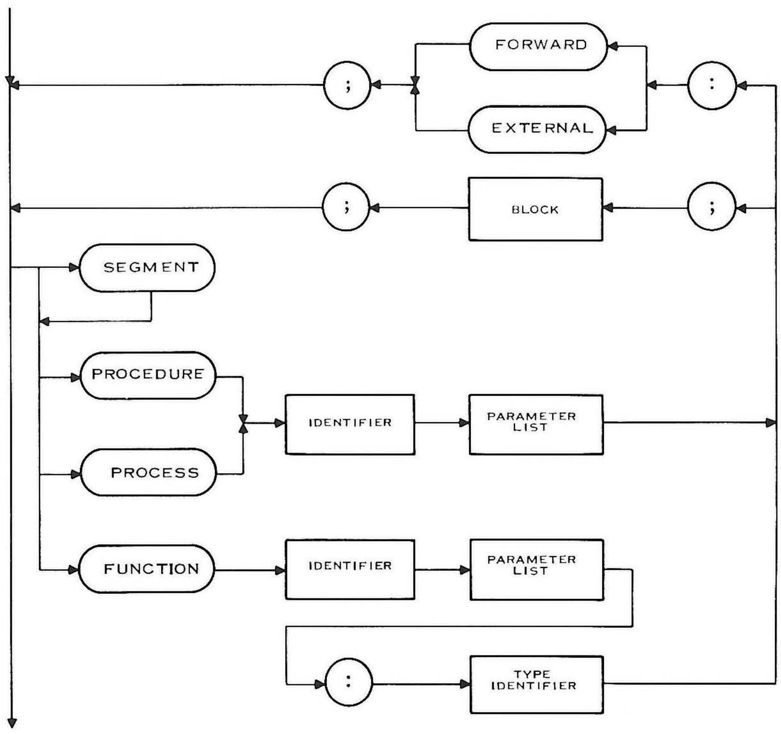

;
,
IDENTIFIER
:
TYPE
CASE
IDENTIFIER
:
TYPE IDENTIFIER
OF
;
,
CONSTANT
:
(
FIELD LIST
)

<SIMPLE TYPE>


TYPE IDENTIFIER
(
IDENTIFIER
)
,
CONSTANT
..
CONSTANT

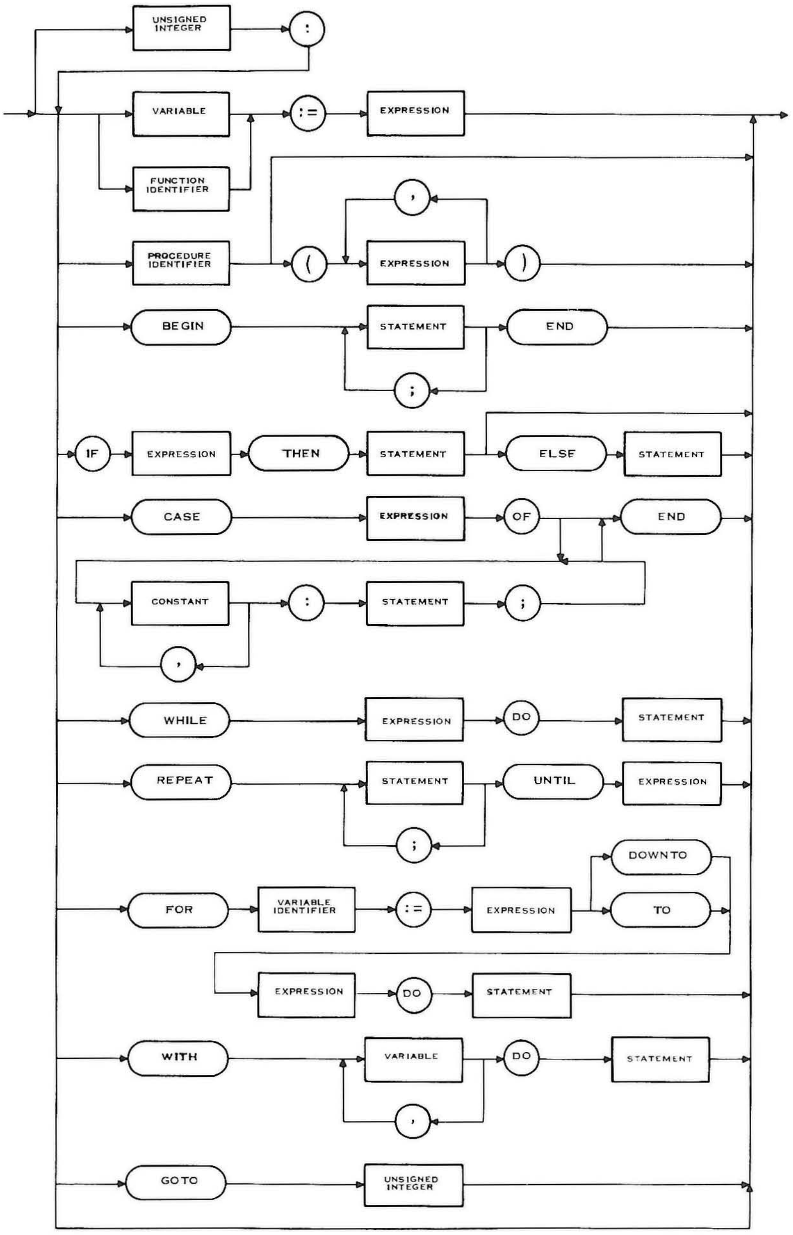
2284009

< ROUTINE >



2284010

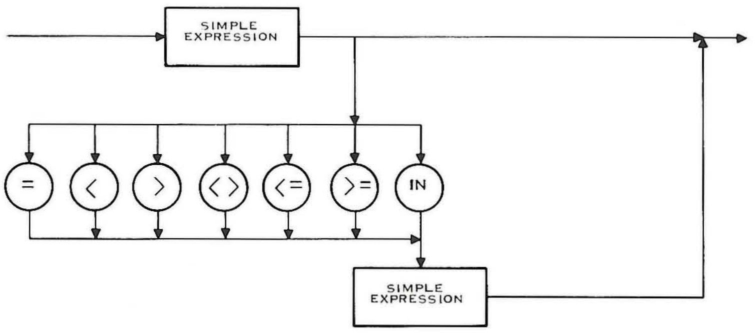
<STATEMENT>



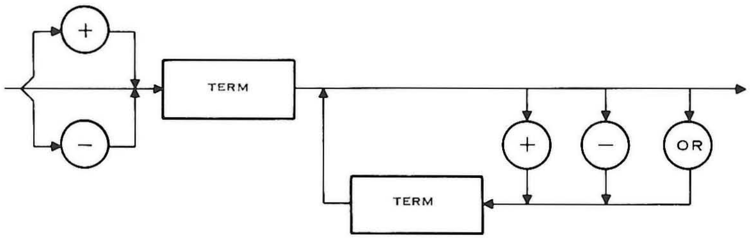
UNSIGNED INTEGER
:
VARIABLE
:=
EXPRESSION
FUNCTION IDENTIFIER
PROCEDURE IDENTIFIER
(
,
EXPRESSION
)
BEGIN
STATEMENT
;
END
IF
EXPRESSION
THEN
STATEMENT
ELSE
STATEMENT
CASE
EXPRESSION
OF
END
CONSTANT
,
:
STATEMENT
;
WHILE
EXPRESSION
DO
STATEMENT
REPEAT
STATEMENT
;
UNTIL
EXPRESSION
FOR
VARIABLE IDENTIFIER
:=
EXPRESSION
DOWNTO
TO
EXPRESSION
DO
STATEMENT
WITH
VARIABLE
,
DO
STATEMENT
GOTO
UNSIGNED INTEGER

2284011

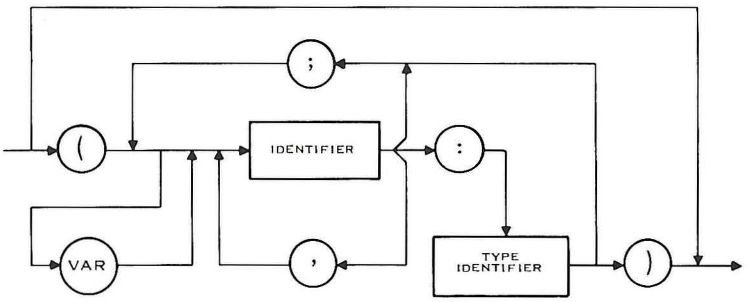
< EXPRESSION >



< SIMPLE EXPRESSION >

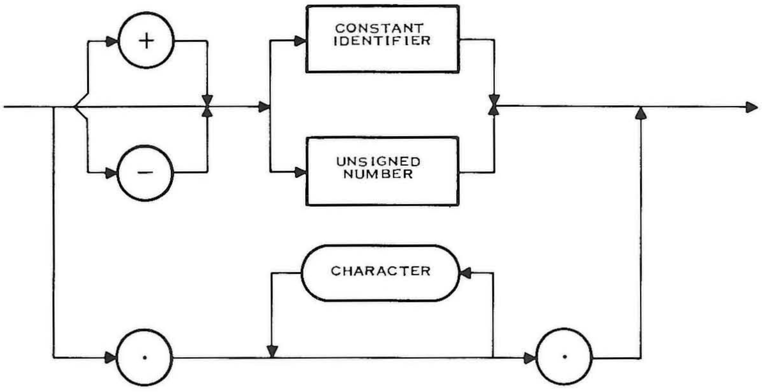


< PARAMETER LIST >



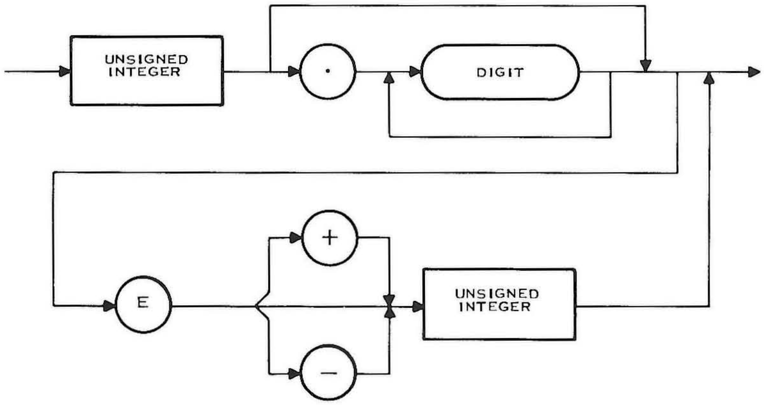
2284012

<CONSTANT>



+
−
CONSTANT IDENTIFIER
UNSIGNED NUMBER
CHARACTER
'
'

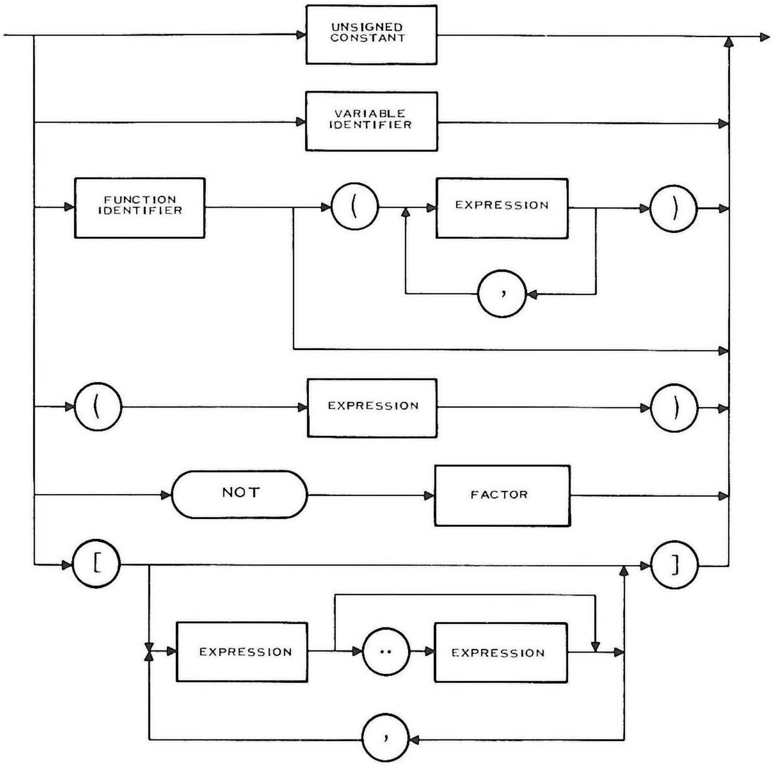
<UNSIGNED NUMBER>



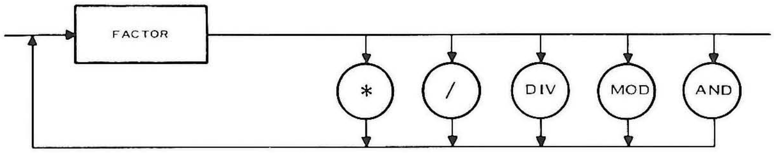
UNSIGNED INTEGER
.
DIGIT
E
+
−
UNSIGNED INTEGER

2284013

<FACTOR>

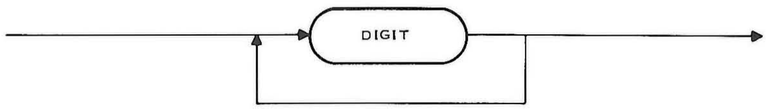

UNSIGNED CONSTANT
VARIABLE IDENTIFIER
FUNCTION IDENTIFIER
(
EXPRESSION
)
,
(
EXPRESSION
)
NOT
FACTOR
[
]
EXPRESSION
..
EXPRESSION
,

<TERM>

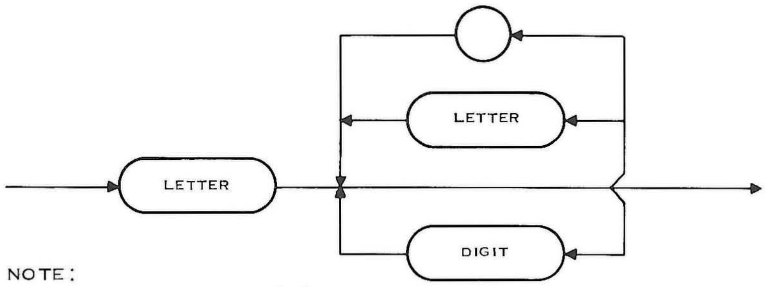

FACTOR
*
/
DIV
MOD
AND

2284014

<UNSIGNED INTEGER>



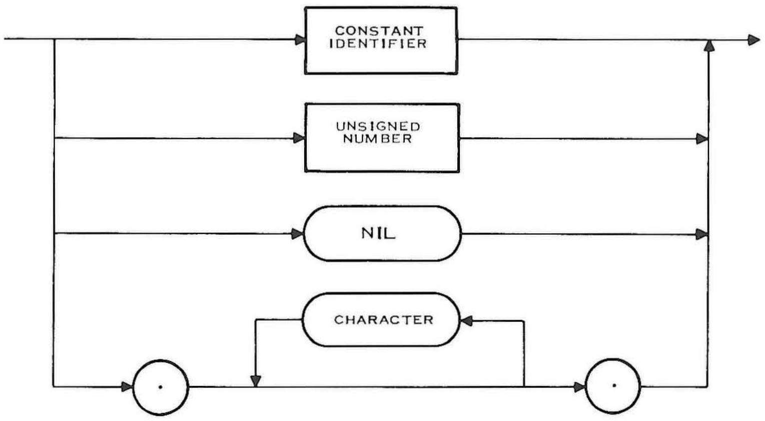
<INDENTIFIER>



NOTE:

THE UNDERSCORE CHARACTER '–' IS ACCEPTED BUT NOT SIGNIFICANT

<UNSIGNED CONSTANT>



2284015

Intrinsics

For your reference, here is an alphabetical list of the intrinsic procedures available in UCSD Pascal, along with the parameters they require (optional parameters are enclosed in brackets: [ ]).

```
ABS(X)
ARCTAN(X)
ATAN(X)
ATTACH(SEM, I_VEC)

BLOCKREAD(FILENAME, BUFFER,
               COUNT [, RELBLOCK])
BLOCKWRITE(FILENAME, BUFFER,
               COUNT [, RELBLOCK])

CHR(I)
CLOSE(FILENAME [, OPTION])
CONCAT(SOURCE1, SOURCE2, ... , SOURCEn)
COPY(SOURCE, INDEX, SIZE)
COS(X)

DELETE(DESTINATION, INDEX, SIZE)
DISPOSE(POINTER {, FIELD_TAG })

EOF [(FILENAME)]
EOLN [(FILENAME)]
EXIT(program)
EXIT(PROGRAM_NAME)
EXIT(ROUTINE_NAME)
EXP(X)

FILLCHAR(DESTINATION, LENGTH, CHARACTER)

GET(FILENAME)
GOTOXY(X, Y)
```

HALT

INSERT(SOURCE, DESTINATION, INDEX)
IORESULT

LENGTH(SOURCE)
LN(X)
LOG(X)

MARK(POINTER)
MEMAVAIL
MEMLOCK(SEG_LIST)
MEMSWAP(SEG_LIST)
MOVELEFT(SOURCE, DESTINATION, LENGTH)
MOVERIGHT(SOURCE, DESTINATION, LENGTH)

NEW(POINTER {, FIELD_TAG })

ODD(I)
ORD(SCALAR_VALUE)

PAGE(FILENAME)
POS(PATTERN, SOURCE)
PRED(SCALAR_VALUE)
PUT(FILENAME)
PWROFTEN(I)

READ([FILENAME ,] VAR1, VAR2, ..., VARn)
READLN([FILENAME ,] VAR1, VAR2, ..., VARn)
RESET(FILENAME [, EXT_FILE])
REWRITE(FILENAME [, EXT_FILE])
RELEASE(POINTER)
ROUND(X)

SCAN(LENGTH, < partial expression> , SOURCE)
SEEK(FILENAME, INDEX)
SEMINIT(SEM, COUNT)
SIGNAL(SEM)
SIN(X)
SIZEOF(VAR_OR_TYPE_ID)
SQR(X)

SQRT(X)
START(< process call> [, PROC__ID [, STACKSIZE [, PRIORITY]]])
STR(I, STR)
SUCC(SCALAR__VALUE)

TIME(HIWORD, LOWORD)
TRUNC(X)

UNITCLEAR(DEVICE__NUMBER)
UNITREAD (DEVICE__NUMBER, BUFFER, LENGTH [,
  [BLOCKNUMBER] , FLAG])
UNITWRITE(DEVICE__NUMBER, BUFFER, LENGTH [,
  [BLOCKNUMBER] , FLAG])

VARAVAIL(SEG__LIST)
VARDISPOSE(POINTER, COUNT)
VARNEW (POINTER, COUNT)

WAIT(SEM)
WRITE([FILENAME ,] VAL1, VAL2, ..., VALn)
WRITELN([FILENAME ,] VAL1, VAL2, ..., VALn)

D

Syntax Errors

1: Error in simple type
2: Identifier expected
3: unimplemented error
4: ')' expected
5: ': ' expected
6: Illegal symbol (terminator expected)
7: Error in parameter list
8: 'OF' expected
9: '(' expected
10: Error in type
11: '[' expected
12: ']' expected
13: 'END' expected
14: ';' expected
15: Integer expected
16: '=' expected
17: 'BEGIN' expected
18: Error in declaration part
19: error in < field-list>
20: '.' expected
21: '*' expected
22: 'INTERFACE' expected
23: 'IMPLEMENTATION' expected
24: 'UNIT' expected

50: Error in constant
51: ': =' expected
52: 'THEN' expected
53: 'UNTIL' expected
54: 'DO' expected
55: 'TO' or 'DOWNTO' expected in for statement
56: 'IF' expected
57: 'FILE' expected
58: Error in < factor> (bad expression)
59: Error in variable

60:	Must be of type 'SEMAPHORE'
61:	Must be of type 'PROCESSID'
62:	Process not allowed at this nesting level
63:	Only main task may start processes
101:	Identifier declared twice
102:	Low bound exceeds high bound
103:	Identifier is not of the appropriate class
104:	Undeclared identifier
105:	sign not allowed
106:	Number expected
107:	Incompatible subrange types
108:	File not allowed here
109:	Type must not be real
110:	< tagfield> type must be scalar or subrange
111:	Incompatible with < tagfield> part
112:	Index type must not be real
113:	Index type must be a scalar or a subrange
114:	Base type must not be real
115:	Base type must be a scalar or a subrange
116:	Error in type of standard procedure parameter
117:	Unsatisfied forward reference
118:	Forward reference type identifier in var declaration
119:	Re-specified params not OK for a forward procedure
120:	Function result type must be scalar, subrange or pointer
121:	File value parameter not allowed
122:	A forward function's result type can't be re-specified
123:	Missing result type in function declaration
124:	F-format for reals only
125:	Error in type of standard procedure parameter
126:	Number of parameters does not agree with declaration
127:	Illegal parameter substitution
128:	Result type does not agree with declaration
129:	Type conflict of operands
130:	Expression is not of set type
131:	Tests on equality allowed only
132:	Strict inclusion not allowed
133:	File comparison not allowed
134:	Illegal type of operand(s)
135:	Type of operand must be Boolean
136:	Set element type must be scalar or subrange

137: Set element types must be compatible
138: Type of variable is not array
139: Index type is not compatible with the declaration
140: Type of variable is not record
141: Type of variable must be file or pointer
142: Illegal parameter solution
143: Illegal type of loop control variable
144: Illegal type of expression
145: Type conflict
146: Assignment of files not allowed
147: Label type incompatible with selecting expression
148: Subrange bounds must be scalar
149: Index type must be integer

150: Assignment to standard function is not allowed
151: Assignment to formal function is not allowed
152: No such field in this record
153: Type error in read
154: Actual parameter must be a variable
155: Control variable cannot be formal or non-local
156: Multidefined case label
157: Too many cases in case statement
158: No such variant in this record
159: Real or string tagfields not allowed
160: Previous declaration was not forward
161: Again forward declared
162: Parameter size must be constant
163: Missing variant in declaration
164: Substitution of standard proc/func not allowed
165: Multidefined label
166: Multideclared label
167: Undeclared label
168: Undefined label
169: Error in base set
170: Value parameter expected
171: Standard file was re-declared
172: Undeclared external file
173: FORTRAN procedure or function expected
174: Pascal function or procedure expected
175: Semaphore value parameter not allowed

182:	Nested UNITs not allowed
183:	External declaration not allowed at this nesting level
184:	External declaration not allowed in INTERFACE section
185:	Segment declaration not allowed in INTERFACE section
186:	Labels not allowed in INTERFACE section
187:	Attempt to open library unsuccessful
188:	UNIT not declared in previous uses declaration
189:	'USES' not allowed at this nesting level
190:	UNIT not in library
191:	Forward declaration was not segment
192:	Forward declaration was segment
193:	Not enough room for this operation
194:	Flag must be declared at top of program
195:	Unit not importable
201:	Error in real number—digit expected
202:	String constant must not exceed source line
203:	Integer constant exceeds range
204:	8 or 9 in octal number
250:	Too many scopes of nested identifiers
251:	Too many nested procedures or functions
252:	Too many forward references of procedure entries
253:	Procedure too long
254:	Too many long constants in this procedure
256:	Too many external references
257:	Too many externals
258:	Too many local files
259:	Expression too complicated
300:	Division by zero
301:	No case provided for this value
302:	Index expression out of bounds
303:	Value to be assigned is out of bounds
304:	Element expression out of range
398:	Implementation restriction
399:	Implementation restriction
400:	Illegal character in text
401:	Unexpected end of input
402:	Error in writing code file, not enough room
403:	Error in reading include file

404:	Error in writing list file, not enough room
405:	'PROGRAM' or 'UNIT' expected
406:	Include file not legal
407:	Include file nesting limit exceeded
408:	INTERFACE section not contained in one file
409:	Unit name reserved for system
410:	disk error
500:	Assembler error

E

American Standard Code for Information Interchange (ASCII) Characters

0 000 00 NUL	32 040 20 SP	64 100 40 @	96 140 60 ‛
1 001 01 SOH	33 041 21 !	65 101 41 A	97 141 61 a
2 002 02 STX	34 042 22 “	66 102 42 B	98 142 62 b
3 003 03 ETX	35 043 23 #	67 103 43 C	99 143 63 c
4 004 04 EOT	36 044 24 $	68 104 44 D	100 144 64 d
5 005 05 ENQ	37 045 25 %	69 105 45 E	101 145 65 e
6 006 06 ACK	38 046 26 &	70 106 46 F	102 146 66 f
7 007 07 BEL	39 047 27 ′	71 107 47 G	103 147 67 g
8 010 08 BS	40 050 28 (	72 110 48 H	104 150 68 h
9 011 09 HT	41 051 29)	73 111 49 I	105 151 69 i
10 012 0A LF	42 052 2A *	74 112 4A J	106 152 6A j
11 013 0B VT	43 053 2B +	75 113 4B K	107 153 6B k
12 014 0C FF	44 054 2C ,	76 114 4C L	108 154 6C l
13 015 0D CR	45 055 2D −	77 115 4D M	109 155 6D m
14 016 0E SO	46 056 2E .	78 116 4E N	110 156 6E n
15 017 0F SI	47 057 2F /	79 117 4F O	111 157 6F o
16 020 10 DLE	48 060 30 0	80 120 50 P	112 160 70 p
17 021 11 DC1	49 061 31 1	81 121 51 Q	113 161 71 q
18 022 12 DC2	50 062 32 2	82 122 52 R	114 162 72 r
19 023 13 DC3	51 063 33 3	83 123 53 S	115 163 73 s
20 024 14 DC4	52 064 34 4	84 124 54 T	116 164 74 t
21 025 15 NAK	53 065 35 5	85 125 55 U	117 165 75 u
22 026 16 SYN	54 066 36 6	86 126 56 V	118 166 76 v
23 027 17 ETB	55 067 37 7	87 127 57 W	119 167 77 w
24 030 18 CAN	56 070 38 8	88 130 58 X	120 170 78 x
25 031 19 EM	57 071 39 9	89 131 59 Y	121 171 79 y
26 032 1A SUB	58 072 3A :	90 132 5A Z	122 172 7A z
27 033 1B ESC	59 073 3B ;	91 133 5B [	123 173 7B {
28 034 1C FS	60 074 3C <	92 134 5C \	124 174 7C |
29 035 1D GS	61 075 3D =	93 135 5D]	125 175 7D }
30 036 1E RS	62 076 3E >	94 136 5E	126 176 7E ∼
31 037 1F US	63 077 3F ?	95 137 5F _	127 177 7F DEL

Index

Page numbers in brown ink indicate pages in Part 2.

THREE-MONTH LIMITED WARRANTY TEXAS INSTRUMENTS PROFESSIONAL COMPUTER SOFTWARE MEDIA

TEXAS INSTRUMENTS INCORPORATED EXTENDS THIS CONSUMER WARRANTY ONLY TO THE ORIGINAL CONSUMER/PURCHASER.

WARRANTY DURATION

The media is warranted for a period of three (3) months from the date of original purchase by the consumer.

Some states do not allow the exclusion or limitation of incidental or consequential damages or limitations on how long an implied warranty lasts, so the above limitations or exclusions may not apply to you.

WARRANTY COVERAGE

This limited warranty covers the cassette or diskette (media) on which the computer program is furnished. It does not extend to the program contained on the media or the accompanying book materials (collectively the Program). The media is warranted against defects in material or workmanship. THIS WARRANTY IS VOID IF THE MEDIA HAS BEEN DAMAGED BY ACCIDENT, UNREASONABLE USE, NEGLECT, IMPROPER SERVICE, OR OTHER CAUSES NOT ARISING OUT OF DEFECTS IN MATERIALS OR WORKMANSHIP.

PERFORMANCE BY TI UNDER WARRANTY

During the above three-month warranty period, defective media will be replaced when it is returned postage prepaid to a Texas Instruments Service Facility listed below or an authorized Texas Instruments Professional Computer Dealer with a copy of the purchase receipt. The replacement media will be warranted for three months from date of replacement. Other than the postage requirement (where allowed by state law), no charge will be made for the replacement. TI strongly recommends that you insure the media for value prior to mailing.

WARRANTY AND CONSEQUENTIAL DAMAGES DISCLAIMERS

ANY IMPLIED WARRANTIES ARISING OUT OF THIS SALE INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED IN DURATION TO THE ABOVE THREE-MONTH PERIOD. TEXAS INSTRUMENTS SHALL NOT BE LIABLE FOR SPECIAL, COLLATERAL, INCIDENTAL, OR CONSEQUENTIAL COSTS, EXPENSES, OR DAMAGES INCURRED BY THE CONSUMER OR ANY OTHER USER ARISING OUT OF THE PURCHASE OR USE OF THE MEDIA. THESE EXCLUDED DAMAGES INCLUDE, BUT ARE NOT LIMITED BY, COST OF REMOVAL OR REINSTALLATION, OUTSIDE COMPUTER TIME, LABOR COSTS, LOSS OF GOODWILL, LOSS OF PROFITS, LOSS OF SAVINGS, OR LOSS OF USE OR INTERRUPTION OF BUSINESS.

LEGAL REMEDIES

This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

TEXAS INSTRUMENTS CONSUMER SERVICE FACILITIES

U.S. Residents:

Texas Instruments
Service Facility
P.O. Box 1444, MS 7758
Houston, Texas 77001

Canadian Residents:

Geophysical Service Inc.
41 Shelley Road
Richmond Hill, Ontario
Canada L4C 5G4

Consumers in California and Oregon may contact the following Texas Instruments offices for additional assistance or information.

Texas Instruments
Consumer Service
831 South Douglas St.
Suite 119
El Segundo, California 90245
(213) 973-2591

Texas Instruments
Consumer Service
6700 S.W. 105th
Kristin Square, Suite 110
Beaverton, Oregon 97005
(503) 643-6758

IMPORTANT NOTICE OF DISCLAIMER REGARDING THE PROGRAM

The following should be read and understood before using the software media and Program.

TI does not warrant that the Program will be free from error or will meet the specific requirements of the purchaser/user. The purchaser/user assumes complete responsibility for any decision made or actions taken based on information obtained using the Program. Any statements made concerning the utility of the Program are not to be construed as expressed or implied warranties.

COPYRIGHT

TEXAS INSTRUMENTS PROFESSIONAL COMPUTER
UCSD Pascal
2232401-0001

Original Issue: 15 April 1983

Your Name: ______________________________

Company: ______________________________

Telephone: ______________________________

Department: ______________________________

Address: ______________________________

City/State/Zip Code: ______________________________

Your comments and suggestions assist us in improving our products. If your comments concern problems with this manual, please list the page number.

Comments:

This form is not intended for use as an order blank.

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 6189 HOUSTON, TX

POSTAGE WILL BE PAID BY ADDRESSEE

Texas Instruments Incorporated
Attn: Marketing M/S 7896
P.O. Box 1444
Houston, TX 77001

FOLD

TEXAS INSTRUMENTS PROFESSIONAL COMPUTER
UCSD Pascal
2232401-0001

Original Issue: 15 April 1983

Your Name: ________________________________

Company: ________________________________

Telephone: ________________________________

Department: ________________________________

Address: ________________________________

City/State/Zip Code: ________________________________

Your comments and suggestions assist us in improving our products. If your comments concern problems with this manual, please list the page number.

Comments:

This form is not intended for use as an order blank.

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 6189 HOUSTON, TX

POSTAGE WILL BE PAID BY ADDRESSEE

Texas Instruments Incorporated
Attn: Marketing M/S 7896
P.O. Box 1444
Houston, TX 77001

FOLD

TEXAS INSTRUMENTS PROFESSIONAL COMPUTER
UCSD Pascal
2232401-0001

Original Issue: 15 April 1983

Your Name: ________________________________

Company: ________________________________

Telephone: ________________________________

Department: ________________________________

Address: ________________________________

City/State/Zip Code: __________________________

Your comments and suggestions assist us in improving our products. If your comments concern problems with this manual, please list the page number.

Comments:

This form is not intended for use as an order blank.

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 6189 HOUSTON, TX

POSTAGE WILL BE PAID BY ADDRESSEE

Texas Instruments Incorporated
Attn: Marketing M/S 7896
P.O. Box 1444
Houston, TX 77001

FOLD

TEXAS INSTRUMENTS

2232401-0001

Printed in U.S.A.