sun microsystems

Pascal Programmer's Guide *for the* Sun Workstation

Sun Microsystems, Inc. • 2550 Garcia Avenue • Mountain View, CA 94043 • 415-960-1300

Part No: 800-1176-01
Revision D of 15 May, 1985

Credits and Acknowledgements

The first version of this *Pascal Programmer's Guide for the Sun Workstation* was originally produced by William N. Joy, Susan L. Graham, Charles B. Haley, Marshall Kirk McKusick, and Peter B. Kessler of the Computer Science Division, Department of Electrical Engineering and Computer Science, at the University of California at Berkeley.

The financial support of the first and second authors' work by the National Science Foundation under grants MCS74-07644-A04, MCS78-07291, and MCS80-05144, and the first author's work by an IBM Graduate Fellowship are gratefully acknowledged.

History of the Implementation

The first Berkeley system was written by Ken Thompson in early 1976. The main features of the present system were implemented by Charles Haley and William Joy during the latter half of 1976. Versions of this system have been in use since January, 1977.

The system was moved to the VAX-11 by Peter Kessler and Kirk McKusick with the porting of the interpreter in the spring of 1979, and the implementation of the compiler in the summer of 1980.

The whole system was moved to the Sun Workstation in 1983 by Peter Kessler and Kirk McKusick.

Copyrights

Revision History

Version	Date	Comments
A	October 1980	This revision was the Berkeley Pascal User's Manual Version 2.0.
B	July 1983	This revision was the Berkeley Pascal User's Manual Version 3.0. Sun Microsystems updates to this manual removed references to the CYBER 6000 implementation details.
C	7 January 1984	Minor corrections and updates.
D	19 November 1984	2.0 α release
D	5 February 1985	2.0 β release
D	15 May 1985	2.0 release

Contents

Contents

Preface

Pascal is available on Sun Workstations as an extended version of the Berkeley Pascal system distributed with UNIX 4.2BSD. The Berkeley Pascal compiler (*pc*) is part of the Sun languages software you received with your workstation. It is supported by the same profilers, debuggers, and libraries available in C and In addition, the Berkeley Pascal system includes a statement-level execution profiler (*pxp*), and a cross-reference generator (*pxref*). The original Berkeley Pascal Interpreter (*pi,px,pix*) remains available and is upwardly compatible with *pc*. Most Pascal programs that run on 4.2BSD should port easily to Sun Workstations.

The Berkeley Pascal system on Suns supports Level 1 ISO Standard Pascal, which includes conformant array parameters and type-safe procedures and functions as parameters. In addition, *pc* supports separate compilation and several extensions for improved access to facilities of UNIX.†

This Programmer's Guide describes how to use *pc*, *pi*, *px*, *pix*, and *pxp*. Details of interactive programs and programs combining Pascal with other languages are also given. A number of examples are provided, including many dealing with input and output.

This manual consists of four chapters and seven appendices:

Chapter 1 is an overview of the system and provides some introductory examples.

Chapter 2 discusses the error diagnostics produced by the translators *pc*, *pi*, the Pascal library, and the interpreter *px*.

Chapter 3 describes input and output and gives special attention to interactive programs and features unique to UNIX.

Chapter 4 gives details on the components of the system and explanation of all relevant options.

Chapter 5 describes the calling sequence used by Pascal for use when calling Pascal routines from other languages.

Chapter 6 describes how to call routines written in C from Pascal programs.

Chapter 7 describes how to call FORTRAN routines from Pascal programs.

Chapter 8 describes Berkeley Pascal's extensions relative to the ISO Pascal Standard, and Sun's extensions relative to 4.2BSD.

Appendix A is a Pascal language reference summary.

Appendix B is an appendix to the Jensen and Wirth Pascal Report defining the Berkeley implementation of the Pascal language, primarily providing historical notes.

Appendix C is a bibliography.

Appendix D contains the manual pages relevant to Pascal.

Chapter 1

Basic UNIX Pascal

The Sun Workstation provides the following Pascal facilities:

- *pc*, a compiler
- *pi*, an interpreter code translator
- *px*, an interpreter
- *pix*, a translator and interpreter (combines the functions of *pi* and *px*)
- *pxp*, a execution profiler
- *pxref*, a cross-reference generator

Pascal's calling conventions are the same as in C, with `var` parameters passed by address and other parameters passed by value.

Both *pc* and *pi* support ISO dp7185 Level 1 Standard Pascal, including conformant array parameters. In addition, *pc* contains several extensions. Deviations from the standard are noted in the **BUGS** section of the *pc*(1) manual page.

In addition to *pc*, there are other tools you might find helpful for creating Pascal programs.

Text Editing The major text editor for source programs is *vi* (vee-eye), the visual display editor. It has considerable power because it offers the capabilities of both a line and a screen editor. *vi* also provides several commands for editing programs, which are options you can set in the editor. Two examples are the ***autoindent*** option, which supplies white space at the beginning of a line, and the ***showmatch*** option, which shows matching parentheses. For more information, see the *Editing and Text Processing* manual section on *vi*.

Debug Aids There are two main debugging tools available on the Sun system:

dbx	a symbolic debugger that understands Pascal, C, and FORTRAN-77 programs.
adb	an interactive, general-purpose, low-level debugger that is not as easy to use as *dbx*.

man pages The on-line documentation consists of pages from the *Commands Reference Manual* called manual or 'man' pages. The applicable manual pages for Pascal are

1. *pc*(1)
2. *pi*(1)
3. *pix*(1)
4. *px*(1)
5. *pxp*(1)
6. *pxref*(1)

To get more information about the syntax for a command, you can display any of the manual pages on your screen by typing

```
hostname% man pc
```

Other manuals Other Sun manuals containing information on editing or using Pascal are

1. *Editing and Text Processing on the Sun Workstation*
2. *Programming Tools for the Sun Workstation*
3. *Commands Reference Manual for the Sun Workstation*
4. *System Interface Manual for the Sun Workstation*

Before reading on, make sure that you are familiar with basic editing techniques used on the Sun Workstation and with writing standard Pascal programs.

1.1. *pc*

pc is the Sun Pascal compiler. If given an argument file named *filename*.p, *pc* compiles the file and leaves the result in an executable file called (by default) `a.out`.

A program can be separated into more than one *.p* file. *pc* can compile a number of *.p* files into object files having the extension *.o* in place of *.p*. Object files can then be loaded to produce an executable *a.out* file. Exactly one object file must supply a `program` statement to successfully create an executable *a.out* file. The rest of the files must consist only of declarations that logically nest within the program.

References to objects shared between separately compiled files are allowed if the objects are declared in included header files having the extension *.h*. Header files may only be included at the outermost level, and thus declare only globally available objects.

To allow external functions and procedures to be declared, an `external` directive has been added, which used similarly to the `forward` directive but can only appear in *.h* files. A binding phase of the compiler checks that declarations are used consistently, to enforce the type-checking rules of Pascal.

Other language processors that create object files can be loaded together with object files created by *pc*. The functions and procedures they define must be declared in *.h* files included by all the *.p* files that call those routines.

For example, consider the example file `greetings.p`:

```
program greetings(output);
begin
        writeln('Hello, world')
end.
```

Compile the program with *pc* then run it as follows:

```
hostname% pc greetings.p
hostname% a.out
Hello, world!
hostname%
```

1.2. *pi*

pi is the Pascal interpreter code translator. It translates `.p` files and puts the interpreter code into the file `obj`. As an example, *pi* translates and runs `greetings.p` as follows:

```
hostname% pi greetings.p

hostname% obj
Hello, world!

1 statement executed in 0.00 seconds cpu time.
hostname%
```

1.3. *pix*

pix combines the functions of *pi* and *px* into one command. Translate and interpret `greeting.p` with *pix* as follows:

```
hostname% pix greetings.p
Execution begins...
Hello, world!

Execution terminated.

1 statements executed in 0.00 seconds cpu time.
hostname%
```

1.4. Formatting the Program Listing

It is possible to use special lines within the source text of a program to format the program listing. An empty line prints without a line number. A line containing only a control-L (formfeed) character causes a page eject in the listing with the corresponding line number suppressed. With *pi*, the **-n** command line option begins each listed *include* file on a new page with a banner line.

1.5. Execution Profiling

An execution profile consists of a structured listing of (all or part of) a program with information about the number of times each statement in the program was executed for a particular run of the program. These profiles can be used for several purposes. In a program that was abnormally terminated due to excessive looping, recursion, or a program fault, the counts can help you to locate the error. Zero counts mark portions of the program that were not executed; during the early debugging stages they should prompt new test data or a reexamination of the program logic. The profile is perhaps most valuable, however, in drawing attention to the (typically small) portions of the program that dominate execution time. This information can be used for source-level optimization.

1.5.1. An Example

A prime number is a positive integer with exactly two divisors, itself and one. The program `primes` determines the first few prime numbers. In translating the program, the **−z** option is specified on the command line to *pc*. This option causes the compiler to generate counters and additional code that record the number of times each statement in the program was executed, which enables *pxp* statement profiling.[1] Thus, the program is translated as follows:

```
hostname% pc -z primes.p
```

Run `primes` as follows:

```
hostname% a.out

     2      3      5      7     11     13     17     19     23     29
    31     37     41     43     47     53     59     61     67     71
    73     79     83     89     97    101    103    107    109    113
   127    131    137    139    149    151    157    163    167    173
   179    181    191    193    197    199    211    223    227    229

hostname%
```

When execution of the program completes (either normally or abnormally) the statement counts are written to the file ***pmon.out*** in the current directory. [2] By running ***pxp*** with the source file containing the program and (implicitly) the file ***pmon.out*** as arguments you can prepare an execution profile. This results in the following output:

[1] The counts are completely accurate only in the absence of runtime errors and nonlocal **goto** statements. This is not generally a problem, however, as in structured programs nonlocal **goto** statements occur infrequently, and counts are incorrect after abnormal termination only when the *upward look* (described below) to get a count passes a suspended call point.

[2] *pmon.out* is similar to *mon.out* and *gmon.out*, which are used by *prof*(1) and *gprof*(1), respectively. Note that both of these profilers are available under *pc*.

```
hostname% pxp primes.p
Berkeley Pascal PXP -- Version 2.14 (11/2/84)

Sat Jan 12 10:01 1985  primes.p

Profiled Sat Jan 12 10:02 1985

     1          1.---|program primes(output);
     2               |const
     2               |    n = 50;
     2               |    n1 = 7; (*n1 = sqrt(n)*)
     3               |var
     3               |    i, k, x, inc, lim, square, l: integer;
     4               |    prim: boolean;
     5               |    p, v: array [1..n1] of integer;
     6               |begin
     7               |    write(2: 6, 3: 6);
     7               |    l := 2;
     8               |    x := 1;
     8               |    inc := 4;
     8               |    lim := 1;
     8               |    square := 9;
     9               |    for i := 3 to n do begin (*find next prime*)
     9           48.---|    repeat
    11               76.---|    x := x + inc;
    11                     |    inc := 6 - inc;
    12                     |    if square <= x then begin
    13                  5.---|    lim := lim + 1;
    14                       |    v[lim] := square;
    14                       |    square := sqr(p[lim + 1])
    14                     |    end;
    16                     |    k := 2;
    16                     |    prim := true;
    17                     |    while prim and (k < lim) do begin
    18                157.---|    k := k + 1;
    19                       |    if v[k] < x then
    19                    42.---|    v[k] := v[k] + 2 * p[k];
    20                       |    prim := x <> v[k]
    20                     |    end
    20                     |until prim;
    23                 |    if i <= n1 then
    23              5.---|    p[i] := x;
    24                 |    write(x: 6);
    24                 |    l := l + 1;
    25                 |    if l = 10 then begin
    26              5.---|    writeln;
    26                   |    l := 0
    26                 |    end
    26               |    end;
    29               |    writeln
    29               |end.
```

The header lines in the outputs of *pc*, *pi*, and *pix* indicate the version of the translator and execution profiler in use at the time this program was prepared. The time given with the file name (also on the header line) indicates the time of last modification of the program source file. This time serves to version stamp the input program. ***pxp*** also indicates the time when the profile data was gathered.

To determine the number of times a statement was executed, look up to the left of the statement and finds the corresponding vertical bar '¦'. If this vertical bar is labeled with a count, then that count gives the number of times the statement was executed. If the bar is not labeled, look upwards in the listing to find the first '¦' above the original one that has a count to find the answer. Thus, in our example, ***k*** was incremented 157 times on line 18, while the ***write*** procedure call on line 24 was executed 48 times (as given by the count on the `repeat` on line 9).

More information on ***pxp*** can be found in its manual section ***pxp***(1) and in the "Options available in px," "Options available in pc," and "Separate Compilation with pc" sections in Chapter 4.

Chapter 2

Error Diagnostics

This section discusses the error diagnostics of the programs ***pi***, ***pc***, ***px*** and ***pix***. See the manual section ***pix***(1) and the "Options common to ***pi***, ***pc***, and ***px***" section found in Chapter 4 for more details. All the diagnostics given by ***pi*** are also given by ***pc***.

2.1. Translator Syntax Errors

This section describes some common syntax errors in Pascal programs and how the compiler handles them.

2.1.1. Illegal Characters

Characters such as '$', '!', and '@' are not part of Pascal. If they are found in the source program and are not part of a string constant, a character constant, or a comment, they are considered to be illegal characters. This can happen if you leave off an opening string quotation mark ('). Most nonprinting characters in your input are also illegal except in character constants and character strings. Except for the tab and formfeed characters, which are used to format the program, nonprinting characters in the input file print as the character '?' in your listing.

2.1.2. String Errors

There is no character string of length zero in Pascal. Consequently the input '′′' is not acceptable. Similarly, encountering an end-of-line after an opening string quotation mark (') without first encountering the matching closing quote yields the diagnostic "Unmatched ′ for string."

Programs containing '#' characters (other than in column 1) can produce this diagnostic. This is because early implementations of Pascal used '#' as a string delimiter. In the Sun implementation, '#' is used for #include and preprocessor directives and must begin in column 1.

2.1.3. Digits in Numbers

This part of the language is a minor nuisance. Pascal requires digits in real numbers both before and after the decimal point. Thus the following statements, which look quite reasonable to FORTRAN users, generate diagnostics in Pascal:

```
Fri Dec 21 14:14 1984  digits.p:
     4  r := 0.;
e -------------^--- Digits required after decimal point
     5  r := .0;
e -----------^--- Digits required before decimal point
     6  r := 1.e10;
e -------------^--- Digits required after decimal point
     7  r := .05e-10;
e -----------^--- Digits required before decimal point
```

These contructs are also illegal as data input to variables that *read* statements whose arguments are variables of type *real*.

2.1.4. Replacements, Insertions, and Deletions

When a syntax error is encountered in the input text, the parser invokes an error recovery procedure. This procedure examines the input text immediately after the point-of-error and uses a set of simple corrections to see whether to allow the analysis to continue. These corrections involve replacing an input token with a different token or inserting a token. Most of these changes do not cause fatal syntax errors. The exception is the insertion of or replacement with a symbol such as an identifier or a number; in these cases, the recovery makes no attempt to determine which identifier or what number should be inserted. Thus, these are considered fatal syntax errors.

Consider the following example:

```
hostname% pix -l synerr.p
Berkeley Pascal PI -- Version 3.5 (1/22/85)

Fri Dec 21 14:14 1984  synerr.p

     1  program syn(output);
     2  var i, j are integer;
e ---------------^--- Replaced identifier with a ':'
     3  begin
     4          for j :* 1 to 20 begin
e ---------------------^--- Replaced '*' with a '='
e -------------------------------^--- Inserted keyword do
     5                  write(j);
     6                  i = 2 ** j;
e ------------------------^--- Inserted ':'
E -----------------------------^--- Inserted identifier
     7                  writeln(i))
E --------------------------------^--- Deleted ')'
     8          end
     9  end.
hostname%
```

The only surprise here may be that Pascal does not have an exponentiation operator, hence the complaint about '**'. This error illustrates that if you assume that the language has a feature that it doesn't have, the translator diagnostic may not indicate this specifically, since it is unlikely to recognize the construct you supply.

2.1.5. Undefined or Improper Identifiers

If an identifier is encountered in the input but is undeclared, the error recovery mechanism replaces it with an identifier of the appropriate class. Further references to this identifier are summarized at the end of the containing `procedure` or `function` or at the end of the `program` if the reference occurred in the main program. Similarly, if an identifier is used in an inappropriate way, (for example, if a `type` identifier is used in an assignment statement, or if a simple variable is used where a `record` variable is required) a diagnostic is produced and an identifier of the appropriate class inserted. Further incorrect references to this identifier are flagged only if they involve incorrect use in a different way, with all incorrect uses being summarized in the same way as undeclared variable uses are.

2.1.6. Expected Symbols, Malformed Constructs

If none of the corrections mentioned above appears reasonable, the error recovery examines the input to the left of the point of error to see if there is only one symbol that can follow this input. If this is the case, the recovery prints a diagnostic which indicates that the given symbol was 'expected.'

In cases where none of these corrections resolve the problems in the input, the recovery may issue a diagnostic that indicates "malformed" input. If necessary, the translator can then skip forward in the input to a place where analysis can continue. This process may cause some errors in the missed text to be skipped.

Consider the following example:

```
tutorial% pix -l synerr2.p
Berkeley Pascal PC -- Version 3.5 (1/22/85)

Fri Dec 21 14:14 1984  synerr2.p:
     1  program synerr2(input,outpu);
     2  integer a(10)
E ------^--- Malformed declaration
     3  begin
     4          read(b);
E -------------------^--- Undefined variable
     5          for c := 1 to 10 do
E ------------------^--- Undefined variable
     6                  a(c) := b * c;
E ----------------------^--- Undefined procedure
E ---------------------------^--- Malformed statement
     7  end.
E 1 - File outpu listed in program statement but not declared
In program synerr2:
  E - a undefined on line 6
  E - b undefined on lines 4
  E - c undefined on line 5 6
tutorial%
```

Here `output` is misspelled and given a FORTRAN style variable declaration that the translator diagnosed as a 'Malformed declaration.' On line 6, parentheses are used for subscripting (as in FORTRAN) rather than the square brackets that are used in Pascal, so the translator noted that *a* was not defined as a `procedure` (delimited by parentheses in Pascal). As it's not permissible to assign values to procedure calls, the translator diagnosed a malformed statement at the point of assignment.

2.1.7. Expected and Unexpected End-of-file, "QUIT"

If the translator finds a complete program, but there is more (noncomment) text in the input file, then it indicates that an end-of-file is expected. This situation may occur after a bracketing error, or if too many ends are present in the input. The message may appear after the recovery says that it "Expected `.'", since a period (.) is the symbol that terminates a program.

If severe errors in the input prohibit further processing, the translator may produce a diagnostic message followed by "QUIT". Examples include nonterminated comments and lines longer than 1024 characters. Consider the following example:

```
tutorial% pix -l mism.p
Berkeley Pascal PI -- Version 3.5 (1/22/85)

Fri Dec 21 14:14 1984  mism.p

     1  program mismatch(output)
     2  begin
e ------^--- Inserted ';'
     3          writeln('***');
     4          { The next line is the last line in the file }
     5          writeln
E ---------------------^--- Malformed declaration
  ---------------------^--- Unexpected end-of-file - QUIT
tutorial%
```

In this case, the end of file was reached before an *end* delimiter.

2.2. Translator Semantic Errors

The following sections explain the typical formats and terminology used in Pascal error messages. For more detailed descriptions of diagnostic messages, refer to Cooper's ***Standard Pascal User Reference Manual*** [1].

2.2.1. Format of the Error Diagnostics

In the example program above, the error diagnostics from the Pascal translator include the line number in the text of the program, as well as the text of the error message. While this number is most often the line where the error occurred, it can refer to the line number containing a bracketing keyword like `end` or `until`. If so, the diagnostic may refer to the previous statement. This occurs because of the method the translator uses for sampling line numbers. The absence of a trailing ';' in the previous statement causes the line number corresponding to the `end` or `until` to become associated with the statement. As Pascal is a free-format language, the line number associations can only be approximate and may seem arbitrary in some cases.

2.2.2. Incompatible Types

Since Pascal is a strongly-typed language, many type errors can occur, which are called ***type clashes*** by the translator. The types allowed for various operators in the language are summarized on page 43 of Cooper [1]. It is important to know that the Pascal translator, in its diagnostics, distinguishes among the following type classes:

```
array      Boolean    char       file       integer
pointer    real       record     scalar     string
```

These words are used in many error messages. Thus, if you tried to assign an `integer` value to a `char` variable you would receive a diagnostic like

```
Fri Dec 21 14:14 1984  clash.p:
E 7 - Type clash: integer is incompatible with char
  ... Type of expression clashed with type of variable in assignment
```

In this case, one error produced a two-line error message. If the same error occurs more than once, the same explanatory diagnostic is given each time.

2.2.3. Scalar

The only class whose meaning is not self-explanatory is `scalar`. Scalar has a precise meaning in the Pascal standard where, in fact, it refers to `char`, `integer`, `real`, and `Boolean` types as well as the enumerated types. For the purposes of the Pascal translator, `scalar` in an error message refers to a user-defined, enumerated type, such as `ops` in the example above or `color` in

```
type color = (red, green, blue)
```

For integers, the more explicit denotation `integer` is used. Although it's correct in the context of the *User Guide* to refer to an integer variable as a `scalar` variable, the interpreter ***pi*** prefers more specific identification.

2.2.4. Function and Procedure Type Errors

For built-in procedures and functions, two kinds of errors occur. If the routines are called with the wrong number of arguments a message like

```
Fri Dec 21 14:14 1984  sin1.p:
E 12 - sin takes exactly one argument
```

is displayed. If the type of an argument is wrong, you receive a message like

```
Fri Dec 21 14:14 1984  sin2.p:
E 12 - sin's argument must be integer or real, not char
```

2.2.5. Procedures and Functions as Parameters

In standard Pascal, procedures and functions used as formal parameters can be declared with (nested) parameter lists of their own. In Jensen and Wirth's Pascal there are no nested parameter lists; therefore, no argument checking is possible in calls made to parametric procedures and functions. Berkeley Pascal requires you to use parametric procedures to conform to the standard, so programs ported from early implementations of Pascal may require modification.

2.2.6. Can't Read and Write Scalars, etc.

Error messages stating that scalar (user-defined) types cannot be written to and from files are often mysterious. In fact, if you define

```
        type color = (red, green, blue)
```

standard Pascal does not associate these constants with the strings 'red', 'green', and 'blue' in any way. An extension has been added to Berkeley Pascal that allows enumerated types to be read and written; however, if the program is to be portable, you must write your own routines to perform these functions. Standard Pascal only allows the reading of characters, integers and real numbers from text files (not strings or Booleans). It's possible to make a

```
        file of color
```

but the representation is binary rather than a string.

2.2.7. Expression Diagnostics

The diagnostics for semantically ill-formed expressions are very explicit as this sample translation:

```
hostname% pi -l expr.p
Berkeley Pascal PI -- Version 3.5 (1/22/85)

Fri Dec 21 14:14 1984  expr.p

     1  program x(output);
     2  var
     3          a: set of char;
     4          b: Boolean;
     5          c: (red, green, blue);
     6          p: ^ integer;
     7          A: alfa;
     8          B: packed array [1..5] of char;
     9  begin
    10          b := true;
    11          c := red;
    12          new(p);
    13          a := [];
    14          A := 'Hello, yellow';
    15          b := a and b;
    16          a := a * 3;
    17          if input < 2 then writeln('boo');
    18          if p <= 2 then writeln('sure nuff');
    19          if A = B then writeln('same');
    20          if c = true then writeln('hue''s and color''s')
    21  end.
E 14 - Constant string too long
E 15 - Left operand of and must be Boolean, not set
E 16 - Cannot mix sets with integers and reals as operands of *
E 17 - files may not participate in comparisons
E 18 - pointers and integers cannot be compared - operator was <=
E 19 - Strings not same length in = comparison
E 20 - scalars and Booleans cannot be compared - operator was =
e 21 - Input is used but not defined in the program statement
In program x:
  w - constant green is never used
  w - constant blue is never used
  w - variable B is used but never set
hostname%
```

This example is admittedly far fetched, but illustrates that the error messages are clear enough to allow you to easily determinate the problem in the expressions.

2.2.8. Type Equivalence

The Pascal translator produces several diagnostics that complain about 'non-equivalent types.' In general, Pascal considers variables to have the same type only if they are declared with the same constructed type or type identifier. Thus, the variables *x* and *y* declared as

```
var
    x: ^ integer;
    y: ^ integer;
```

do not have the same type. The assignment

```
        x := y
```

produces the diagnostic messages

```
        Fri Dec 21 14:14 1984  typequ.p:
        E 7 - Type clash: non-identical pointer types
          ... Type of expression clashed with type of variable in assignment
```

It is always necessary to declare a type such as

```
        type intptr = ^ integer;
```

and use it to declare

```
        var x: intptr; y: intptr;
```

Note that if we had initially declared

```
        var x, y: ^ integer;
```

then the assignment statement would have worked. The statement

```
        x^ := y^
```

is allowed in either case. Since the parameter to a `procedure` or `function` must be declared with a type identifier rather than a constructed type, it is always necessary to declare any type that is used in this way.

2.2.9. Unreachable Statements

Sun Pascal flags unreachable statements. Such statements usually correspond to errors in the program logic. Note that a statement is considered to be reachable if there is a potential path of control, even if it can never be taken. Thus, no diagnostic is produced for the statement:

```
        if false then
            writeln('impossible!')
```

2.2.10. `gotos` in Structured Statements

The translator detects and complains about `goto` statements that transfer control into structured statements (e.g., `for` and `while`). It does not allow such jumps, nor does it allow branching from the `then` part of an `if` statement into the `else` part. Such checks are made only within the body of a single procedure or function.

2.2.11. Unused Variables, Never-Set Variables

Although *pi* always clears variables to zero at `procedure` and `function` entry, *pc* does not unless runtime checking is enabled using the **−C** option. It is not good programming practice to rely on this initialization. To discourage this practice, and to help detect errors in program logic, *pi* flags as a 'w' warning error the following:

- Use of a variable that is never assigned a value.
- A variable that is declared but never used, distinguishing between those variables whose values are computed but that are never used, and those that are completely unused.

In fact, these diagnostics are applied to all declared items. Thus a `const` or a `procedure` that is declared but never used is flagged. The **−w** option of *pi* may be used to suppress these warnings; (see "Options" and "Options Common to pi, pc, and pix" in Chapter 4).

Note: Since variable uses and assignments are not tracked across separate commpilation units, *pc* ignores uninitialized and unused variables in the global scope. This also applies to fields or records whose types are global.

2.3. Translator Panics, I/O Errors

2.3.1. *Panics*

One class of error that rarely occurs, but that causes termination of all processing when it does, is a *panic*. A panic indicates a translator-detected internal inconsistency. A typical panic message is

```
snark (rvalue) line=110 yyline=109
Snark in pi
```

If you receive such a message, the translation is quickly and (perhaps) ungracefully terminated. Save a copy of your program to inspect later, then contact Technical Support at Sun Microsystems. If you were making changes to an existing program when the problem occurred, you may be able to work around the problem by determining which change caused the `snark` and making a different change or error correction to your program.

Panics are also possible in *px*, particularly if range checking is disabled with the `−t` option.

2.3.2. *Out of Memory*

If you receive an "out of space" message from the translator during translation of a large `procedure` or `function` or one containing a large number of string constants, you can either break the offending procedure or function into smaller pieces or increase the maximum data segment size using the *limit* command of *csh*(1).

In practice, the compiler rarely runs out of memory on Sun workstations.

2.3.3. *I/O Errors*

Other errors that you may encounter when running *pi* relate to input/output. If *pi* cannot open the file you specify, or if the file is empty, an error occurs.

2.4. Runtime Errors in *pix*

The second example illustrates one run-time error. Here are general descriptions of run-time errors. The more unusual interpreter error messages are explained briefly in the manual pages section for *px*(1).

2.4.1. Start-up Errors

These errors occur when the object file to be executed is not available or appropriate. Typical errors here are caused by the specified object file not existing, not being a Pascal object, or not being accessible to the user.

2.4.2. Program Execution Errors

These errors occur when the program interacts with the Pascal runtime environment in an inappropriate way. Typical errors are values or subscripts out of range, bad arguments to built-in functions, exceeding the statement limit because of an infinite (or very long) loop, or running out of memory[3]. The interpreter produces a traceback after the error occurs, showing all the active routine calls, unless the **−p** option was disabled when the program was translated. Unfortunately, no variable values are given and no way of extracting them is available.

As an example of such an error, assume that you have accidentally declared the constant ***n1*** to be 6, instead of 7 on line 2 of the program 'primes' (as given in the ***Execution profiling*** section in Chapter 1). If you run this program, you get the following response:

```
hostname% pix primes.p
Execution begins...
     2     3     5     7    11    13    17    19    23    29
    31    37    41    43    47    53    59    61    67    71
    73    79    83    89    97   101   103   107   109   113
   127   131   137   139   149   151   157   163   167
Subscript value of 7 is out of range

        Error in "error"+8 near line 14.
Execution terminated abnormally.

996 statements executed in 0.32 seconds cpu time.
```

The interpreter indicates that the program terminated abnormally due to a subscript out of range near line 14, which is eight lines into the body of the program `primes`.

2.4.3. Interrupts

If a program running under *px* is interrupted while executing, and the **−p** option was not specified to *pi*, then a traceback is printed.[4] The file *pmon.out* of profile information is written if the program was translated with the **−z** option enabled (*pc*, *pi*, or *pix*).

[3] The checks for running out of memory are not foolproof and there is a chance that the interpreter will fault, producing a core image, when it runs out of memory. This situation occurs very rarely.

[4] Occasionally, the Pascal system is in an inconsistent state when this occurs (for example, when an interrupt terminates a **procedure** or **function** entry or exit). In this case, the traceback only contains the current routine. A reverse call-order list of procedures is not given.

2.4.4. I/O Interaction Errors

The final class of interpreter errors results from inappropriate interactions with files, including your Sun workstation. Included here are bad formats for integers and real numbers (such as no digits after the decimal point) when reading.

2.4.5. Runtime Errors in pc

Programs compiled with ***pc*** use the same library routines as the interpreter ***px***. They detect essentially the same error conditions as ***px***, but support error diagnosis and debugging differently from ***px***.

When an error is detected in a program compiled by ***pc***, a message describing the error is printed and the program is aborted, producing a core image. For example, consider the previous example, which was run using ***pix***. Compiling this program using the **-C** and **-g** options and running it you get

```
hostname% pc primes.p -C -g
hostname% a.out

     2     3     5     7    11    13    17    19    23    29
    31    37    41    43    47    53    59    61    67    71
    73    79    83    89    97   101   103   107   109   113
   127   131   137   139   149   151   157   163   167   173
   179   181   191   193   197   199   211   223   227   229

hostname%
```

Note that unlike ***px*** or ***pix***, no traceback is produced. However, since the program was compiled with the **-g** option you can use the ***dbx*** debugger to help debug the program.

2.5. Comparing

This section lists differences between the compiler ***pc*** and the interpreter programs ***pi/px/pix***. ***pi*** may be used for small, easy-to-debug programs having negligible execution time. For most applications, ***pc*** is a better choice.

2.5.1. Language Features of pc Not Supported by pi

Both ***pi*** and ***pc*** support ISO standard Pascal with numerous extensions. However, some extensions supported by ***pc*** are not supported by ***pi***. In most cases, these are related to separate compilation or to compatibility with other languages:

- The predefined type `shortreal` (IEEE single-precision floating point)
- External procedure declarations
- Bitwise logical operations on integral types
- Preprocessor facilities other than file inclusion

For details on these and other extensions, see Appendix A.

2.5.2. Separate Compilation

pi compiles a single source file, which can contain `#include` commands. As in standard Pascal, programs compiled by *pi* must be organized as a single unit, including the outer `begin...end` block and the `program` heading. *pc* allows programs to be developed as separately compiled source modules.

2.5.3. Access to UNIX

Programs processed by *pi* cannot call library or UNIX system routines, except for routines predefined by *pi* and built into *px*. Program modules compiled by *pc* can call C library routines directly; details are given in Appendix D.

2.5.4. Performance

pi compiles more quickly than *pc*, but at the expense of execution efficiency. It has been estimated that *pi* compiles a source program at five times the speed of *pc*, but programs compiled by *pi* and interpreted by *px* can run 30 times slower than the same programs compiled by *pc*.

These numbers vary according to the application and release version of Berkeley Pascal running on Sun Workstations. They are given here, however, to illustrate the performance tradeoffs between compiled and interpreted programs.

2.5.5. Debugging

Programs compiled by *pc* can be debugged using either the assembly-level debugger (*adb*) or the source-level debugger (*dbx*). There is no debugger running on Sun Workstations for interpreted Pascal programs, so programs compiled by *pi* must be debugged with `writeln` and `assert` statements.

Chapter 3

Input and Output

This chapter describes features of the Pascal input/output environment, with special consideration of the features specific to interactive programs.

3.1. Introduction

In Berkeley Pascal, the predefined file variables `input` and `output` are equivalent to the UNIX standard input and output files (known as `stdin` and `stdout`). Consequently, Pascal programs can be easily used in the UNIX environment to read or write files by using the shell to redirect `stdin` and `stdout`. For example, consider the following program, which copies input to output:

```
program copy(input,output);
var c: char;
begin
   while not eof to begin
      while not eoln do begin
         read(ch);
         write(ch);
      end;
      readln;
      writeln;
   end;
end.
```

Assume that the program above is saved in a file called `copy.p`. First, you would compile it and produce a program called `copy` as follows:

```
hostname% pc copy.p -o copy
```

Next, run the program. Since the standard files `input` and `output` default to the terminal, the program simply echoes each line typed, terminating when a line beginning with an end-of-file (control-D) character is typed.

```
hostname% copy
hello, are you listening?
hello, are you listening?
goodbye, I must go now.
goodbye, I must go now.
(CTRL-D)
hostname%
```

By using the shell's ">" operator to redirect output, you can create a short text file called `data`.

```
hostname% copy > data
hello, are you listening?
goodbye, I must go now.
(CTRL-D)
hostname%
```

Using the same program, but with the "<" operator to redirect input the file prints on the terminal:

```
hostname% copy < data
hello, are you listening?
goodbye, I must go now.
hostname%
```

There are other ways to associate Pascal file variables with UNIX files. One simple way, which is restrictive but usually portable to other Pascal systems, is to list the name of the file as a file variable in the `program` statement. The Pascal library associates the file variable with a file of the same name. For example, the following program copies a UNIX file named `data` to output:

```
program copydata(data,output);
var c: char;
    data: text'
begin
   reset(data);
   while not eof(data) do begin
      while not eoln(data) do begin
         read(data,ch);
         write(ch);
      end;
      readln(data);
      writeln;
  end;
end.
```

Assuming that the file `data` is still in the current directory and the `copydata` program is saved in `copydata.p`, you can compile and run the program as follows:

```
hostname% pc copydata.p -o copydata
hostname% copydata
hello, are you listening?
goodbye, I must go now.
hostname%
```

There are other more flexible ways to associate Pascal file variables with UNIX files; for example, actual filenames can be taken from string constants or variables, or from command-line arguments using the built-in procedures `argc` and `argv`. Details are given in later sections of this manual.

3.2. eof and eoln

An extremely common problem encountered by new users of Pascal, especially in the interactive environment offered by UNIX, relates to the definitions of `eof` and `eoln`. These functions are supposed to be defined at the beginning of execution of a Pascal program, indicating whether the input device is at the end of a line or the end of a file (or neither). Setting `eof` or `eoln` actually corresponds to an implicit read in which the input is inspected, but not "used up." In fact, there is no way the system can know whether the input is at end-of-file or the end of a line unless it attempts to read a line from it. If the input is from a previously created file, then this reading can take place without runtime action by the user. However, if the input is from a terminal, then the input is what you type. [5] If the system does an initial read automatically at the beginning of program execution, and if the input is a terminal, the user would have to type some input before execution could begin. This would make it impossible for the program to begin by prompting for input.

Berkeley Pascal has been designed so that an initial read is not necessary. At any given time, the Pascal system may or may not know whether the end-of-file and end-of-line conditions are true. Thus, internally, these functions can have three values: TRUE, FALSE, and "I don't know yet; if you ask me I'll have to find out." All files remain in this last, indeterminate state until the Pascal program requires a value for `eof` or `eoln` either explicitly or implicitly, for example, in a call to `read`. The important point to note here is that if you force the Pascal system to determine whether the input is at the end-of-file or the end-of-line, it is necessary for it to attempt to read from the input.

Consider the following example code:

```
while not eof do begin
    write('number, please? ');
    read(i);
    writeln('that was a ', i: 2)
end
```

At first glance, this may appear to be a correct program for requesting, reading and echoing numbers. Notice, however, that the `while` loop asks whether `eof` is true before the request is printed. This forces the Pascal system to decide whether the input is at the end-of-file. The Pascal system gives no messages; it simply waits for the user to type a line. By producing the desired prompting before testing `eof`, the following code avoids this problem:

```
write('number, please ?');
while not eof do begin
    read(i);
    writeln('that was a ', i:2);
    write('number, please ?')
end
```

You must still type a line before the `while` test is completed, but the prompt asks for it. This example, however, is still not correct. To understand why, it is first necessary to know that

[5] It is not possible to determine whether the input is a terminal, as the input may appear to be a file but actually be a *pipe*, the output of a program which is reading from the terminal.

there is a blank character at the end of each line in a Pascal text file. When reading integers or real numbers, the `read` procedure is defined so that when only blanks are left in the file, a zero value is returned and the end-of-file condition is set. If, however, there is a number remaining in the file the end-of-file condition is not set even if it is the last number, since `read` never reads the blanks after the number (and there is always at least one blank). Thus, the modified code still puts out a spurious

```
        that was a 0
```

at the end of a session when end-of-file is reached. The simplest way to correct the problem in this example is to use the procedure `readln` instead of `read`. In general, unless you test the end-of-file condition both before and after calls to `read` or `readln`, there will be inputs that cause your program to attempt to read past the end-of-file.

3.3. More About eoln

To have a good understanding of when `eoln` is true it is necessary to know that in any file there is a special character indicating end-of-line, and that in effect, the Pascal system always reads one character ahead of the Pascal `read` commands.[6] For instance, in response to `read(`*ch*`)`, the system sets ***ch*** to the current input character and gets the next input character. If the current input character is the last character of the line, then the next input character from the file is the newline character, the normal UNIX line separator. When the `read` routine gets the newline character, it replaces that character by a blank (causing every line to end with a blank) and sets `eoln` to TRUE. `eoln` is TRUE as soon as you read the last character of the line and before you read the blank character corresponding to the end of line. Thus, it is almost always a mistake to write a program that deals with input in the following way:

```
read(ch);
if eoln then
    Done with line
else
    Normal processing
```

as this almost always has the effect of ignoring the last character in the line. The `read(`*ch*`)` belongs as part of the normal processing.

Given this framework, it is not hard to explain the function of a `readln` call, which is defined as:

```
while not eoln do
    get(input);
get(input);
```

This advances the file until the blank corresponding to the end-of-line is the current input symbol and then discards this blank. The next character available from `read` is the first character of the next line, if one exists.

[6] In Pascal terms, `read(`*ch*`)` corresponds to 'ch := input^; get(input)'.

3.4. Output Buffering

A final point about Pascal input/output concerns the buffering of the file `output`. It is extremely inefficient for the Pascal system to send each character to the user's terminal as the program generates it for output — even less efficient if the output is the input of another program such as the line printer daemon *lpr*(1). To gain efficiency, the Pascal system "buffers" the output characters (i.e., it saves them in memory until the buffer is full and then emits the entire buffer in one system interaction). However, to allow interactive prompting to work as in the example given above, this prompt must be printed before the Pascal system waits for a response. For this reason, Pascal normally prints all the output that has been generated for the file `output` whenever one of the following occurs:

- a `writeln` occurs
- the program reads from the terminal
- the procedure `message` or `flush` is called

Thus, in the code sequence

```
for i := 1 to 5 do begin
    write(i: 2);
    Compute a lot with no output
end;
writeln
```

the output integer does not print until the `writeln` occurs. The delay can be somewhat disconcerting, and you should be aware that it does occur. By setting the **−b** option to 0 before the `program` statement by inserting a comment of the form

```
(*$b0*)
```

you can cause `output` to be completely unbuffered, with a corresponding large degradation in program efficiency. Option control in comments is discussed in the "Using Options" section in Chapter 4.

3.5. Files, Reset, and Rewrite

It is possible to use extended forms of the built-in functions `reset` and `rewrite` to get more general associations of UNIX file names with Pascal file variables. When a file other than `input` or `output` is to be read or written, then the reading or writing must be preceded by a `reset` or `rewrite` call. In general, if the Pascal file variable has never been used before, there will be no UNIX filename associated with it. By mentioning the file in a `program` statement, however, we can cause a UNIX file with the same name as the Pascal variable to be associated with it. If we do not mention a file in the `program` statement and use it for the first time with the statement

```
reset(f)
```

or

```
rewrite(f)
```

then the Pascal system generates a temporary name of the form `tmp.x` for some character `x`, and associates this UNIX filename with the Pascal file. The first such generated name is 'tmp.1' and the names continue by incrementing the filename extension through the ASCII set. The advantage of using such temporary files is that they are automatically `removed` by the Pascal system as soon as they become inaccessible. They are not removed, however, if a runtime error causes termination while they are in scope.

To cause a particular UNIX pathname to be associated with a Pascal file variable you can give that name in the `reset` or `rewrite` call. For example, you could have associated the Pascal file `data` with the file `primes` (see "Translator Syntax Errors" section in Chapter 2) by doing:

```
        reset(data, 'primes')
```

instead of a simple

```
        reset(data)
```

In this case it is not essential to mention `data` in the program statement, but it is still a good idea because it serves as an aid to program documentation. The second parameter to `reset` and `rewrite` can be any string value, including a variable. Thus the names of UNIX files to be associated with Pascal file variables can be read in at runtime. Full details on filename/file variable associations are given in "Restriction and Limitations" section of Appendix A.

3.6. Argc and Argv

Each UNIX process receives a variable-length sequence of arguments, each of which is a variable-length character string. The built-in function `argc` and the built-in procedure `argv` can be used to access and process these arguments. The value of the function `argc` is the number of arguments to the process. By convention, the arguments are treated as an array and indexed from 0 to `argc—1`, with the zeroth argument being the name of the program being executed. The rest of the arguments are those passed to the command on the command line. Thus, the command

```
        tutorial% obj  /etc/motd  /usr/dict/words  hello
```

invokes the program in the file `obj` with `argc` having a value of 4. The zeroth element accessed by `argv` is `obj`, the first `/etc/motd`, and so on.

Pascal does not provide variable-size arrays, nor does it allow character strings of varying length. For this reason, `argv` is a procedure and has the syntax

```
        argv(i, a)
```

where `i` is an integer and `a` is a string variable. This procedure call assigns the (possibly truncated or blank-padded) `i'th` argument of the current process to the string variable `a`. The file manipulation routines `reset` and `rewrite` strip trailing blanks from their optional second arguments so that this blank padding is not a problem in the usual case where the arguments are filenames.

The Berkeley Pascal program `kat` illustrates the use of `arc` and `argv`, which can be used with the same syntax (except for the options to `cat`) as the UNIX system program `cat(1)`.

First compile the program:

```
        hostname% pc kat.p -o kat
```

Then run the program:

```
hostname% kat kat.p
program kat(input, output);
var
    ch: char;
    i: integer;
    name: packed array [1..100] of char;
begin
    i := 1;
    repeat
        if i < argc then begin
            argv(i, name);
            reset(input, name);          {nonstandard}
            i := i + 1
        end;
        while not eof do begin
            while not eoln do begin
                read(ch);
                write(ch)
            end;
            readln;
            writeln
        end
    until i >= argc
end { kat }.
tutorial%
```

Note that the `reset` call to the file `input` may not be allowed on other systems. As this program deals mostly with `argc` and `argv` and UNIX system-dependent considerations, portability is of little concern.

If this program is in the file `kat.p`, then do the following:

```
tutorial% pi kat.p
tutorial% mv obj kat
tutorial% kat kat.p
program kat(input, output);
var
    ch: char;
    i: integer;
    name: packed array [1..100] of char;
begin
    i := 1;
    repeat
        if i < argc then begin
            argv(i, name);
            reset(input, name);
            i := i + 1
        end;
        while not eof do begin
            while not eoln do begin
                read(ch);
                write(ch)
            end;
            readln;
            writeln
        end
    until i >= argc
end { kat }.

1152 statements executed in 0.36 seconds cpu time.

hostname%  kat
This is a line of text.
This is a line of text.
The next line contains only an end-of-file (an invisible control-d!)
The next line contains only an end-of-file (an invisible control-d!)

288 statements executed in 0.10 seconds cpu time.
hostname%
```

Thus, if it is given arguments, `kat` (like `cat`) copies each one in turn. If no arguments are given, it copies from the standard input. Thus it works as it did before, with

```
tutorial% kat < kat.p
```

now equivalent to

```
tutorial% kat kat.p
```

although the mechanisms are quite different in the two cases.

Chapter 4

System Component Details

4.1. Using Options

The programs *pi*, *pc*, and *pxp* take several options.[7] There is a standard UNIX convention for passing options to programs on the command line, which is followed by the Berkeley Pascal system programs. As you saw in previous examples, option-related arguments consist of the character '–' followed by a single character option name.

Except for the **–b** option, which takes a single-digit value, each option may be set on (enabled) or off (disabled). When an on/off-valued option appears on the command line of *pi* or *px*, it inverts the default setting of that option. Thus

```
hostname% pi -l foo.p
```

enables the listing option **–l**, since it is off by default, while

```
hostname% pi -t foo.p
```

disables the run-time tests option **–t**, since it is on by default.

In addition to inverting the default settings of *pi* options on the command line, it is also possible to control them within the body of the program by using comments of the special form:

```
{$l-}
```

The opening comment delimiter, which could also be a '(*', is immediately followed by the character '$'. After the '$', which signals the start of the option list, you can place a sequence of letters and option controls, separated by commas. The most basic actions for options are to set them, thus

```
{$l+ Enable listing}
```

or to clear them

```
{$t-,p- No run-time tests, no post mortem analysis}
```

Notice that '+' always enables an option and '–' always disables it, no matter what the default is. Thus '–' has a different meaning in an option comment than it has on the command line. As shown in the examples, normal comment text may follow the option list.

[7] As *pix* uses *pi* to translate Pascal programs, it takes the options of *pi* also. We refer to them here, however, as *pi* options.

4.2. Options Common to *pi*, *pc*, and *pix*

The following options are common to both the compiler and the interpreter. Refer to the appropriate manual page in Appendix G for a summary of the options to each command. With each option the default setting (the setting it would have if it appeared on the command line), and a sample command using the option are given.

4.2.1. **—L** — *Map Identifiers and Keywords to Lower Case*

Programs transported from other systems are often written with mixed-case identifiers and keywords. This option cleans up such a program for use with Berkeley Pascal.

4.2.2. **—b** — *Buffering of the File* `output`

The **—b** option controls the buffering of the file `output`. The default is line buffering, with flushing at each reference to the file `input` and under certain other circumstances detailed in "Options Available in pc" section found later in this chapter. Mentioning **—b** on the command line, that is:

```
hostname% pi -b assembler.p
```

makes standard output block-buffered, where a block is some system-defined number of characters. The **—b** option can also be controlled in comments. Unique among the Berkeley Pascal options, it takes a single-digit value rather than an on or off setting. A value of 0, that is

```
{$b0}
```

makes `output` unbuffered. Any value two or greater causes block buffering and is equivalent to the flag on the command line. The option control comment setting **—b** must precede the `program` statement.

4.2.3. **—i** — *Include File Listing*

The **—i** option takes the name of an `include` file, `procedure` or `function` name and causes it to be listed while translating[8]. Typical uses would be

```
hostname% pix -i scanner.i compiler.p
```

to make a listing of the routines in the file `scanner.i`, and

```
hostname% pix -i scanner compiler.p
```

to make a listing of only the routine `scanner`. This option is especially useful for conservation-minded programmers who are making partial program listings.

[8] **Include** files are discussed in the "Multi-file programs" section later in this chapter.

4.2.4. `-l` — *Make a Listing*

The `-l` option enables a listing of the program. `-l` is off by default. When specified on the command line, it creates a header line identifying the version of the translator in use and a line giving the modification time of the file being translated to appear before the actual program listing. The `-l` option is pushed and popped by the `-i` option at appropriate points in the program.

4.2.5. `-s` — *Standard Pascal Only*

The `-s` option causes many of the features of Berkeley Pascal implementation that are not found in standard Pascal to be diagnosed as 's' warning errors. This option is off by default and is enabled when mentioned on the command line. Some of the features that are diagnosed are nonstandard procedures and functions, extensions to the procedure ***write***, and padding of constant strings with blanks. In addition, all letters are mapped to lower case except in strings and characters, so that the case of keywords and identifiers is effectively ignored. The `-s` option is most useful when a program is to be transported.

4.2.6. `-t` *and* `-C` — *Runtime Tests*

These options control the generation of tests that subrange variable values are within bounds at runtime. *pi* defaults to generating tests and uses the option `-t` to disable them. *pc* defaults to not generating tests, and uses the option `-C` to enable them. Disabling runtime tests also causes `assert` statements to be treated as comments.[9]

4.2.7. `-w` — *Suppress Warning Diagnostics*

The `-w` option, which is on by default, allows the translator to print a number of warnings about inconsistencies it finds in the input program. Turning this option off with a comment of the form

```
{$w-}
```

or on the command line

```
hostname% pi -w tryme.p
```

suppresses these diagnostics.

4.2.8. `-z` — *Generate Counters for a pxp Execution Profile*

The `-z` option, off by default, enables the production of execution profiles. Specifying `-z` on the command line:

```
hostname% pi -z foo.p
```

or enabling it in a comment before the `program` statement, causes *pi* and *pc* to insert code in

[9] See the section on the **`Assert statement`** in Appendix E for details.

the program to count the number of times each statement was executed. An example of using *pxp* is given in the "Execution profiling" section in Chapter 1; its options are described in the "Options Available in pxp" section later in this chapter. Note that the **−z** option cannot be used on separately-compiled programs.

4.3. Options Available in *pi*

4.3.1. **−p** — *Post-Mortem Dump*

The **−p** option is on by default, and causes the runtime system to initiate a post-mortem trace-back when an error occurs. The **−p** option also makes *px* count statements in the executing program, enforcing a statement limit to prevent infinite loops. Specifying **−p** on the command line disables these checks and the ability to produce this post-mortem analysis. It does make smaller and faster programs, however. It is also possible to control the **−p** option in comments. To prevent the post-mortem traceback on error, **−p** must be off at the end of the `program` statement.

4.3.2. **−o** — *Redirect the Output File*

−o is used to specify the output file used in place of *a.out*. Its typical use is to name the compiled program using the root of the filename of the Pascal program. Thus,

```
hostname% pc -o myprog myprog.p
```

causes the compiled program *myprog.p* to be called *myprog*.

4.4. Options Available in *px*

The first argument to *px* is the name of the file containing the program to be interpreted. If no argument is given, then the file *.obj* is executed. If more arguments are given, they are available to the Pascal program by using the built-ins `argc` and `argv` as described in the "argc and argv" section in Chapter 3.

px can also be invoked automatically. In this case, whenever a Pascal object file name is given as a command, the command will be executed with *px* prepended to it; that is

```
hostname% obj primes
```

is converted to read

```
hostname% px obj primes
```

4.5. Options Available in *pc*

4.5.1. **–S** *— Generate Assembly Language*

The program is compiled and the assembly language output is left in the file ***sourcefile.s***. Thus,

```
        hostname% pc -S foo.p
```

places the assembly language translation of `foo.p` in the file `foo.s`. No executable file is created.

4.5.2. **–g** *— Symbolic Debugger Information*

The **–g** option causes the compiler to generate information needed by ***dbx***(1), the source-level debugger.

4.5.3. **–o** *— Redirect the Output File*

–o is the same as in "Options Available in ***pi***."

4.5.4. **–p** *and* **–pg** *— Generate an Execution Profile*

The compiler produces code that counts the number of times each routine is called. The profiling is based on a periodic sample taken by the system, rather than by inline counters (as with ***pxp***). This results in less degradation in execution, but with a loss in accuracy. **–p** causes a `mon.out` file to be produced for ***prof***(1). **–pg** causes a `gmon.out` file to be produced for ***gprof***(1), a more sophisticated profiling tool.

4.5.5. **–O** *— Run the Object Code Optimizer*

The output of the compiler is run through the object code optimizer. This causes an increase in compile time in exchange for a decrease in compiled code size and execution time.

4.5.6. **–P** *— Partial Evaluation of Boolean Expressions*

Partial evaluation semantics are used on the boolean operators `and` and `or`. Left-to-right evaluation is guaranteed and the second operand is evaluated only if necessary to determine the result.

4.5.7. **–I***dir — Specify Directories for Include Files*

`#include` files whose names don't begin with "/" are always searched for first in the directory of the ***file*** argument, then in directories named in **–I** options, then in ***user/include/pascal.***

4.5.8. **−Dname=***def — Define Name to Preprocessor*

Define ***name*** to the preprocessor, as if by `#define`. If no definition is given, the name is defined as "1."

4.5.9. **−U***name — Undefine Name to the Preprocessor*

This option removes any initial definition of ***name***.

4.5.10. **−fsky** *— Generate In-Line Code for SKY Board*

Generates code that assumes the presence of a SKY floating-point processor board. Programs compiled with this option can only be run in systems that have a SKY board installed. Without the **−fsky** option, the SKY board runs slower, since it uses the SKY board library routines. If any part of a program is compiled using this option, you must also use this option when linking with the *pc* command, since different startup routines are used to initialize the SKY board.

4.6. Options Available in *pxp*

On its command line, ***pxp*** takes a list of options followed by the program filename, which must end in '*.p*' (as it must for ***pi***, *pc*, and ***pix***). ***pxp*** produces an execution profile if any of the **−z**, **−t**, or **−c** options are specified on the command line. If none is specified, then ***pxp*** functions as a program reformatter.

It is important to note that only the **−z** and **−w** options of ***pxp***, which are common to ***pi***, *pc*, and ***pxp*** can be controlled in comments. All other options must be specified on the command line to have any effect.

The options listed below are relevant to profiling with ***pxp***.

4.6.1. **−a** *— Include the Bodies of All Routines in the Profile*

To make the profile more compact, ***pxp*** does not normally print the bodies of routines that were not executed. This option forces all routine bodies to be printed.

4.6.2. **−d** *— Suppress Declaration Parts from a Profile*

Normally a profile includes declaration parts. Specifying **−d** on the command line suppresses declaration parts.

4.6.3. **−e** *— Eliminate #include Directives*

Normally, ***pxp*** preserves `#include` directives in the output when reformatting a program, as though they were comments. Specifying **−e** causes the contents of the specified files to be reformatted into the output stream instead. This is an easy way to eliminate `#include` directives, for example, before transporting a program.

4.6.4. `-f` *— Fully Parenthesize Expressions*

Normally ***pxp*** prints expressions with the minimum number of parentheses necessary to preserve the structure of the input. This option causes ***pxp*** to fully parenthesize expressions. Thus, the statement that normally prints as

```
        d := a + b mod c / e
```

prints as

```
        d := a + ((b mod c) / e)
```

when the `-f` option is specified on the command line.

4.6.5. `-j` *— Left-Justify all Procedures and Functions*

Normally, each procedure and function body is indented to reflect its static nesting depth. This option prevents this nesting and can be used if the indented output would be too wide.

4.6.6. `-t` *— Print a Table Summarizing Procedure and Function Calls*

The `-t` option causes ***pxp*** to print a table summarizing the number of calls to each `procedure` and `function` in the program. It may be specified in combination with the `-z` option, or separately.

4.6.7. `-z` *— Enable and Control the Profile*

The `-z` profile option is very similar to the `-i` listing control option of ***pi***. If `-z` is specified on the command line, then all arguments up to the source file argument (which ends in *.p*) are taken to be the names of procedures and functions or `include` files that are to be profiled. If this list is null, then the whole file is profiled. A typical command for extracting a profile of part of a large program would be

```
        hostname% pxp -z parser.i test compiler.p
```

This specifies that profiles of the routines in the file ***parser.i*** and the routine `test` are to be made.

4.7. Formatting programs using *pxp*

The program ***pxp*** can be used to reformat programs by using a command of the form

```
        hostname% pxp dirty.p > clean.p
```

Note that since the shell creates the output file `clean.p` before ***pxp*** executes, `clean.p` and `dirty.p` must not be the same file.

pxp automatically paragraphs the program. It performs housekeeping chores such as comment alignment, and treating blank lines (lines containing exactly one blank or lines containing only a formfeed character) as though they were comments, preserving their vertical spacing effect in the output. ***pxp*** processes four kinds of comments:

- Left-marginal comments beginning in the first column of the input line are placed in the first column of an output line.
- Aligned comments preceded by no input tokens on the input line are aligned in the output with the running program text.
- Trailing comments preceded in the input line by a token are placed with no more than two spaces separating the token from the comment.
- Right-marginal comments, preceded in the input line by a token from which they are separated by at least three spaces or a tab, are aligned down the right margin of the output. They are aligned to the first tab stop after the 40th column from the current "left margin".

Consider the following program:

```
hostname% cat comments.p
{ This is a left marginal comment. }
program hello(output);
var i : integer; {This is a trailing comment}
j : integer;    {This is a right marginal comment}
k : array [ 1..10] of array [1..10] of integer; {Marginal, but past the margin}
{
  An aligned, multi-line comment
  which explains what this program is
  all about
}
begin
i := 1; {Trailing i comment}
{A left marginal comment}
 {An aligned comment}
j := 1;          {Right marginal comment}
k[1] := 1;
writeln(i, j, k[1])
end.
```

When formatted by *pxp* the following output is produced:

```
hostname% pxp comments.p
{ This is a left marginal comment. }

program hello(output);
var
    i: integer; {This is a trailing comment}
    j: integer;                             {This is a right marginal comment}
    k: array [1..10] of array [1..10] of integer;{Marginal, but past the margin}
{
  An aligned, multi-line comment
  which explains what this program is
  all about
}
begin
    i := 1; {Trailing i comment}
{A left marginal comment}
    {An aligned comment}
    j := 1;                                 {Right marginal comment}
    k[1] := 1;
    writeln(i, j, k[1])
end.
hostname%
```

The following formatting-related options are currently available in *pxp*. The options **−f** and **−j** described in the previous section may also be of interest.

4.7.1. −s — Strip Comments

The **−s** option causes *pxp* to remove all comments from the input text.

4.7.2. −_ — Underline Keywords

A command line argument of the form −_, as in

```
hostname% pxp -_ dirty.p
```

causes *pxp* to underline all keywords in the output for enhanced readability.

4.7.3. −[23456789] — Specify Indenting Unit

The normal unit that *pxp* uses to indent a structure statement level is four spaces. By giving an argument of the form −*d*, with *d* a digit, $2 \leq d \leq 9$, you can specify that *d* spaces are to be used per level instead.

4.8. *pxref*

The cross-reference program *pxref* can be used to make cross-referenced listings of Pascal programs. To produce a cross reference of the program in the file '*foo.p*' you can execute the command:

```
hostname% pxref foo.p
```

The cross reference is unfortunately not block-structured. Full details on *pxref* are given in the *pxref*(1) manual page.

4.9. Multi-file programs

A text inclusion facility is available in Berkeley Pascal. This facility allows the interpolation of source text from other files into the source stream of the translator. It can be used to divide large programs into more manageable pieces to facilitate editing, listing, and maintaining them. The inclusion facility is also used in *pc* for sharing common declarations among separately-compiled modules. See the following section for information about compiling modules separately with *pc*.

The `include` facility is similar to that of the UNIX C compiler. To use it, place the character '#' in the first position of a line immediately followed by the word `include`, and then a filename enclosed in single '' or double '"' quotation marks. The filename may be followed by a semicolon if you wish to treat this as a pseudo-Pascal statement. The filenames of included files must end in '`.i`'. An example of the use of included files in a main program is

```
program compiler(input, output, obj);

#include "globals.i"
#include "scanner.i"
#include "parser.i"
#include "semantics.i"

begin
    { main program }
end.
```

When the `include` pseudo-statement is encountered in the input, the lines from the included file are inserted into the input stream. For the purposes of translation and run-time diagnostics and statement numbers in the listings and post-mortem tracebacks, the lines in the included file are numbered starting from 1. Nested includes may be up to 10 levels deep.

See the description of the `-i` option of *pi* in the "Options Common to *pi*, *pc*, and *pix*" section found in this chapter; this can be used to control listing when `include` files are present.

When a nontrivial line is encountered in the source text after an `include` finishes, the 'popped' filename is printed, in the same manner as above.

For the purposes of error diagnostics when not making a listing, the filename is printed before each diagnostic if the current filename has changed since the last one was printed.

4.10. Separate Compilation with *pc*

A separate compilation facility is provided in the Berkeley Pascal compiler, *pc*. This facility allows programs to be divided into a number of files that are compiled individually and linked together later. This is especially useful for large programs, where small changes would otherwise require time-consuming recompilation of the entire program.

Normally, *pc* expects to be given entire Pascal programs. However, if you give the **−c** option on the command line, *pc* accept a sequence of definitions and declarations, and compiles them into a `.o` file that can be linked with a Pascal program at a later time. In order to have procedures and functions available across separately compiled files, they must be declared with the `external` directive. This directive is similar to the directive `forward` in that it must precede the resolution of the function or procedure, and formal parameters and function result types must be specified in the `external` declaration but may not be specified in the resolution.

Type checking is performed across separately compiled files during loading. Since Pascal type definitions define unique types, any types that are shared between separately-compiled files must have the same definitions.

This problem is solved using a facility similar to the `include` facility discussed above. Definitions can be placed in files having the extension `.h` and the files included by separately-compiled files. Each definition from a `.h` file defines a unique type, and all uses of a definition from the same `.h` file define the same type.

Similarly, the facility is extended to allow the definition of `consts` and the declaration of `labels`, `vars`, and `external functions` and `procedures`. Thus procedures and functions that are used between separately-compiled files must be declared `external`, and must be so declared in a `.h` file included by any file that calls or resolves the function or procedure. Conversely, functions and procedures declared `external` can only be so declared in `.h` files. These files can only be included at the outermost level and define or declare global objects. Note that since only `external function` and `procedure` declarations (and not resolutions) are allowed in `.h` files, statically nested functions and procedures can't be declared `external`.

An example of the use of included `.h` files in a program is:

```
program compiler(input, output, obj);

#include "globals.h"
#include "scanner.h"
#include "parser.h"
#include "semantics.h"

begin
    { main program }
end.
```

The main program might include the definitions and declarations of all the global `labels`, `consts`, `types`, and `vars` from the file `globals.h`, and the `external function` and `procedure` declarations for each of the separately-compiled files for the scanner, parser, and semantics. The header file `scanner.h` would contain declarations of the form:

```
type
    token = record
        { token fields }
    end;

function scan(var inputfile: text): token;
    external;
```

Then the scanner might be in a separately-compiled file containing

```
#include "globals.h"
#include "scanner.h"

function scan;
begin
        { scanner code }
end;
```

which includes the same global definitions and declarations and resolves the scanner functions and procedures declared `external` in the file `scanner.h`.

Chapter 5

Calling Pascal From Other Languages

This section describes the Pascal calling sequence used when other languages call Pascal routines. The following topics are discussed:

- Argument list layout
- Value parameters
- Conformant array parameters
- Procedures and functions as parameters

5.1. Argument List Layout

The argument list consists of up to four separate sections, in the following order:

1. Storage for declared arguments. These are either values or pointers to values, and in any case, correspond directly to explicitly declared formal parameters in the Pascal procedure or function declaration.
2. Storage for auxiliary arguments associated with conformant array parameters.
3. Storage for auxiliary arguments associated with procedures or functions passed as parameters.
4. Storage for an auxiliary argument required if the called routine is declared within another procedure. This can only arise in C or FORTRAN if a nested Pascal procedure is passed as an argument.

The parameter list is organized like this primarily in order to make it easy to call other languages from Pascal. In general, auxiliary items for Pascal-specific requirements (such as range checking and static scoping) have been moved outside the "primary" argument list.

In C notation, the calling sequence observed by *pc* is described as follows:

declared procedures and scalar-valued functions:

```
    p(...args...[,cap bounds][,pf slinks][,slink])
```

structure-valued functions:

```
    (temp = p(...args...[,cap bounds][,pf slinks][,slink]),&temp)
```

formal procedures and scalar-valued functions:

```
    (*p)(...args...[,cap bounds][,pf slinks],slink)
```

formal structure-valued functions:

```
    (temp = (*p)(...args...[,cap bounds][,pf slinks],slink), &temp)
```

where:
`cap bounds` are bounds pairs for conformant array parameters
`pf slinks` are static links for procedure/function parameters
`slink` is the static link of the called procedure/function
`temp` is storage allocated by the caller for the result of a structure-valued function

5.2. Value Parameters

In general, Pascal expects all value parameters except conformant array parameters to be passed directly on the stack, widening to a full word representation if necessary. From C, there are two places where this causes trouble: scalars of type `shortreal`, and arrays of any fixed type.

5.2.1. *Type* `shortreal`

Parameters of type `shortreal` are assumed to have been passed in single precision; note that this differs from C, which always converts `float` arguments to `double` before pushing them on the stack.

If a Pascal procedure with a `shortreal` value parameter must be called from C, use the following device:

For the caller (C), use:

```
extern foo();    /* procedure foo(x:shortreal); */

union {
        int intval;
        float fval;
} u;
...
u.fval = <expression of type 'float'>
foo(u.intval);
```

For the callee (Pascal), use:

```
procedure foo(x: shortreal);
begin
...
end;
```

5.2.2. *Fixed Array Types*

C does not pass arrays by value, but does pass structures by value. An array can be passed by value to Pascal by enclosing the array declaration in a dummy structure. For example, consider the following Pascal routine:

```
procedure foo(name: alfa);
begin
        ... do something with name...
end;
```

where `alfa` is defined by

```
alfa = packed array[1..10] of char;
```

The routine `foo` may be called by using the auxiliary declaration

```
typedef  struct {
        char cbuf[10];
} alfa;

alfa digits;
strncpy(digits, "0123456789", sizeof(digits));
foo(digits);
```

Since this interface is neither efficient nor general, it should be avoided whenever possible. A more general interface is described in the next section.

5.2.3. *Value Conformant Array Parameters*

Value conformant array parameters are handled by creating a copy in the caller's environment and passing a pointer to the copy. In addition, the bounds of the array must be passed (this is described in "Argument List Layout" found earlier in this appendix). For example:

The caller (C):

```
extern foo();

char a[] = "this is a string";
...
foo(a, 0, sizeof(a)-1);
```

The callee (Pascal):

```
procedure foo(s: packed array[lb..ub: integer] of char);
begin
        ...
end;
```

From FORTRAN:

FORTRAN passes all arguments by reference. Thus, from FORTRAN it is impossible to call a Pascal routine that expects value parameters.

5.3. Conformant Array Parameters

A conformant array parameter must include bounds and possibly element widths as arguments. These go immediately after the declared argument list. An element width is included for all except the last dimension of a multidimensional array.

Note that since the bounds are passed by value, Pascal routines with conformant array parameters cannot be called from FORTRAN.

If the called routine knows the element width at compile time, the pair

```
(low bound, high bound: integer)
```

is passed. For C, the low bound is always 0.

If the called routine does not know the element width at compile time, (i.e., for all dimensions but the last dimension of a multidimensional conformant array) a triple

```
(low bound, high bound, element width: integer)
```

must be passed. The element width is computed as

```
(ub - lb + 1) * w
```

where (`lb`, `ub`, `w`) are the bounds and element width of the next lower dimension of the array. Note that this definition is recursive.

Finally, note that bounds information may be shared by several conformant array parameters; this is a consequence of their declaration structure. For example, only one bounds pair is passed for the declaration

```
function innerproduct(
        var x,y: array[lb..ub: integer] of real): real;
external;
```

This could be used from C as follows:

```
#define N 100
double vector1[N], vector2[N];
extern double innerproduct(
        /* double x[],y[]; int lb, ub; */
);
double ip;
ip = innerproduct(vector1, vector2, 0, N-1);
```

5.4. Procedures and Functions as Parameters

A procedure or function passed as an argument is associated with a static link to its lexical parent's activation record. When an outer block procedure or function is passed as an argument, Pascal passes a null pointer in the position normally occupied by the passed routine's static link. So that procedures and functions can be passed to other languages as arguments, the static links for all procedure or function arguments are placed after the end of the conformant array bounds pairs (if any) so that procedures and functions may be passed to other languages as arguments.

Routines in other languages may be passed to Pascal; a dummy argument must be passed in the position normally occupied by the passed routine's static link. If the passed routine is not a Pascal routine, the argument is used only as a place holder.

Chapter 6

The Pascal--C Interface

This appendix gives information for constructing interfaces between Pascal and C routines. It contains information that is necessary for calling existing C library routines from Pascal, as well as for writing Pascal-callable routines in C. However, it is not intended to serve as a tutorial on either subject. Familiarity with both C and Pascal is assumed.

6.1. Order of Declaration of Arguments

The order that arguments are declared is the same in Pascal and C. Certain forms of arguments in Pascal (i.e., procedures, functions, and conformant arrays) cause the compiler to pass additional information after the declared argument list; however, in most cases, external C routines need not be aware of this additional information. See Chapter 5 for further details.

6.2. Value Parameters vs. Reference Parameters

In C, all parameters except arrays are passed by value. Pascal `var` (reference) parameters are handled in C by declaring the formal parameter to be a pointer type. Thus the following Pascal declaration:

```
procedure  incr(var n: integer);
external c;
```

corresponds to the C function

```
incr(n)
        int *n;
{
        *n += 1;
}
```

Pascal allows structured types (records, arrays, and sets) to be passed by value. In C, this is true only of structures and unions. If an array of fixed type is to be passed by value to C, the called routine should declare the formal parameter as a structure. For example:

The caller (Pascal):

```
type
    intarray = array[0..9] of integer;
...
    procedure foo(arr: intarray);
    external c;
```

The callee (C):

```
typedef struct {
    int a[10];
} intarray;

foo(arr)
        intarray arr;
{
        ...
}
```

This type of interface should be avoided if possible, since it is neither general nor efficient.

6.3. Conformant Array Parameters

The conformant array parameter feature of ISO Standard Pascal provides a means of passing arrays of different dimensions to a single routine. For a general description of this feature, see Cooper[1]. Conformant array parameters can be passed to C programs; the argument seen by a C program is a pointer to the array.

Pascal passes the bounds of the array at the end of the argument list; C routines can choose to ignore the bounds if some other convention is followed (e.g., an explicit length parameter or a terminating value). For example:

The caller (Pascal):

```
{
| search returns index in [0..len-1] if value is found
| in a[]; otherwise it returns -1.  Note that actual array
| must have lower bound of 0.
}

function search(
    var a: array[lb..ub:integer] of integer;
    len: integer;
    value: integer): integer;
external c;
```

The callee (C):

```
/*
 * return index in [0..len-1] if value is found
 * in a, else return -1
 */
int
search(a,len,value)
        int a[];
        int len;
        int value;
        /* int lb,ub; NOTUSED */
{
        ...
}
```

Conformant array parameters can be passed by value; if this is done, a copy of the array is made in the caller's environment and the address of the copy is placed in the argument list. This property is useful for dealing with character strings (see "Character String Types" later in this Appendix for further details).

6.4. Procedures and Functions as Parameters

Pascal procedures and functions can be passed as parameters to external C routines. The argument seen by C is a pointer to the text of the passed routine. For example:

The caller (Pascal):

```
type element = record ... end;

procedure qsort(
    var elist: array[lb..ub:integer] of element;
    nelements: integer;
    elementsize: integer;
    function compare(var el1, el2: element): integer );
external c;

function compare(var x,y: element): integer;
begin
    ...
end;
```

The callee (C):

```
typedef struct {
        ...
} element;

qsort(elist,nelements,elementsize,compare)
        element elist[];
        int nelements;
        int elementsize;
        int (*compare)();
{
        ...
}
```

The Pascal compiler appends an extra argument to the argument list, which is significant if the actual procedure is nested; consequently, nested Pascal routines should not be passed as parameters to C or FORTRAN. The compiler issues a warning if this is attempted.

6.5. Compatible Types in Pascal and C

Sizes and alignments of types common to both Pascal and C are listed in the table below:

Pascal type	C type	Size	Alignment
shortreal	float	4 bytes	2 bytes
real	double	8 bytes	2 bytes
longreal	double	8 bytes	2 bytes
integer	int	4 bytes	2 bytes
-32768..32767	short	2 bytes	2 bytes
-128..127	char	1 byte	1 byte
boolean	char	1 byte	1 byte
char	char	1 byte	1 byte
record	struct/union	2 bytes	2 bytes
array	array	2 bytes	2 bytes

In most cases, C arrays and structures describe the same objects as their Pascal equivalents, provided that the components have compatible types and are declared in the same order. Exceptions are noted in the next section.

6.6. Incompatible Types in Pascal and C

This section describes types that differ between Pascal and C. In some cases, the differences are minor; in others, a type has no equivalent in the other language, and can be reproduced only with difficulty.

6.6.1. C Bit Fields

The Pascal compiler ignores the word 'packed', so the minimum field width is 1 byte. Consequently, C bit fields have no equivalent in Pascal, and should be avoided in structures shared by C and Pascal code.

6.6.2. Enumerated Types

In Pascal, enumerated types are represented internally by sequences of integral values starting with 0. Storage is allocated for a variable of an enumerated type as if the type were a subrange of integer. For example, an enumerated type of fewer than 128 elements is treated as 0..127, which according to the rules above, is equivalent to a `char` in C.

In C, enumerated types are allocated a full word and can take on arbitrary integer values.

6.6.3. Character String Types

In Pascal, strings are values of character array type. String assignments and comparisons involving string constants imply blank-padding of the constant to the length of the longer operand. C does not support string assignments, except through library functions (see *string*). By convention, strings in C end with a null byte.

C routines with character string parameters expect a string to be passed by address and be terminated by a null byte. To meet these requirements, the formal parameter should be declared in Pascal as a value conformant `array of char`. Note that null termination is only guaranteed when the actual parameter is a string constant, and that this guarantee is not required by the ISO Pascal Standard.

As an example, the following Pascal program lists the files in the current directory by using the C library routine *system*(3), which executes a string as a shell command:

```
program usesystem;
    procedure system(
       cmd: packed array[lb..ub: integer] of char);
    external c;
begin
    system('/bin/ls -l');
end.
```

Some common constructs in C rely on the fact that strings in C denote static variable storage; the user is cautioned to avoid such idioms in Pascal, especially when calling C library routines. For example, typical usage of *mktemp*(3) in Pascal would be as follows:

```
tmp := mktemp('/tmp/foo.xxxxxxxx');      {WRONG}
```

This is incorrect, since `mktemp()` modifies its argument. A correct solution is to use the C library routine `strncpy()` (see *string*) to copy the string constant to a declared char array variable.

```
procedure strncpy(
   var dest: packed array[l1..u1:integer] of char;
   srce: packed array[l2..u2:integer] of char;
   length: integer);
       external c;

procedure mktemp(
   var dest: packed array[lb..ub:integer] of char);
external c;

       var pathname: packed array[1..40] of char;

strncpy(pathname, '/tmp/foo.xxxxxxxx', sizeof(pathname));
mktemp(pathname);
```

6.6.4. *Pascal Set Types*

In Pascal, a set is implemented as a bit vector, which may be thought of as a C byte array. Direct access to individual elements of a set is highly machine dependent and should be avoided. Note that the implementation may change in a future release.

In the Sun implementation, bits are numbered within a byte from least significant to most significant. For example, the bits in a variable of type `set of 0..31` would be ordered:

set+0:	7, 6, 5, 4, 3, 2, 1, 0
set+1:	15,14,13,12,11,10, 9, 8
set+2:	23,22,21,20,19,18,17,16
set+3:	31,30,29,28,27,26,25,24

In C, a set could be described as a byte array beginning at an even address. The *n*th element in a set [lower...upper] can be tested as follows:

```
#define BITMASK         07
#define BITNUMSIZE      03
register indx;
upper -= lower; /* normalize upper bound */
if ((indx = n - lower) < 0 || indx > upper) {
        /* n is outside the range [lower..upper] */
}
if (setptr[indx >> BITNUMSIZE] & (1 << (indx & BITMASK))) {
        /* n is in [lower..upper] */
}
/* n is not in [lower..upper] */
```

6.6.5. *Pascal Variant Records*

C equivalents of variant records can usually be constructed by the following somewhat awkward correspondence:

Pascal:

```
        record
            <fixed part fields>
            case <tag field> of
            <tag value list(1)>: ( <variant field list(1)> );
                   ...
            <tag value list(n)>: ( <variant field list(n)> );
        end;
```

C:

```
        struct {
            <fixed part fields>
            <tag field>
            union {
                struct { <variant field list(1)> } <name(1)>;
                    ...
                struct { <variant field list(2)> } <name(n)>;
            } <name>;
        };
```

The correspondence fails if the variant part begins at an odd address, which occurs if none of the variants requires word alignment. The problem is that in C, each variant must be represented by a nested structure, which always begins at an even address. In Pascal this restriction is not observed because a variant does not begin a new record. For example:

```
    var x : record
        case tag: char of
        'a': (ch1, ch2: char);
        'b': (flag: boolean);
    end;
```

does not correspond to a C structure, since the substructure of the 'a' variant is not word aligned. However, one can force the variant part of this record to be aligned by adding another variant, for example,

```
    var x : record
                case tag: char of
                'a': (ch1, ch2: char);
                'b': (flag: boolean);
                'K': (ALIGN: integer);
            end;
```

The corresponding C structure is then

```
struct {
        char tag;
        union {
                struct {
                        char ch1, ch2;
                }a_var;
                struct {
                        char flag;
                }b_var;
                struct {
                        int ALIGN;
                }c_var;
        }var_part;
} x;
```

Chapter 7

The Pascal - FORTRAN Interface

This section describes the interface for calling FORTRAN from Pascal. It describes parameter passing conventions, and the mapping between types in FORTRAN and their equivalents in Pascal.

7.1. Order of Declaration of Arguments

The order of declaration of arguments is the same in Pascal and FORTRAN.

7.2. Value Parameters vs. Reference Parameters

In FORTRAN, all parameters are passed by reference, including constants and function results. In general, all constants and temporary values are handled by creating a copy in the caller's environment and passing the copy by reference. The Pascal compiler follows this convention for routines declared with the `external fortran` directive. For example:

The caller (Pascal):

```
function hypot(x,y: real): real;
external fortran;
...
z := hypot(3, 4);
assert(z = 5.0);
```

The callee (FORTRAN):

```
double precision function hypot(x,y)
double precision x,y
hypot = sqrt(x**2 + y**2)
return
end
```

7.3. Conformant Array Parameters

External FORTRAN routines can be declared to accept one-dimensional arrays of different sizes by using conformant array parameters. The calling sequence passes the array bounds at the end of the argument list. Unfortunately, the bounds are not accessible from FORTRAN. In general, the caller must supply an explicit length parameter. For example,

The caller (Pascal):

```
function innerproduct(
    var x,y: array[lb..ub:integer] of real;
    nelements: integer): real;
external fortran;
```

The callee (FORTRAN):

```
double precision function innerproduct(x,y,n)
double precision x,y
integer n
dimension x(n), y(n)
end
```

Multidimensional arrays can cause problems if passed from Pascal to FORTRAN; see the section on "Multidimensional Arrays" later in this appendix for details.

7.4. Procedures and Functions as Parameters

Pascal procedures and functions can be passed as parameters to external FORTRAN routines, subject to the following restrictions:

- All formal parameters of the passed routine must be `var` parameters since the source language of a compiled routine is not recorded in its runtime representation.
- The actual routine passed must be declared at the outer block level.
- All formal parameters of the passed routine must have types with compatible equivalents in FORTRAN.

The argument that FORTRAN sees should be declared with an `external` statement. For example:

The caller(Pascal):

```
function apply(
    function f(var xx:real): real;
    var x: real): real;
external fortran;
```

The callee (FORTRAN):

```
double precision function apply(f,x)
external f
double precision f,x
apply = f(x)
return
end
```

7.5. Compatible Types in Pascal and FORTRAN

Size and alignments of types common to both Pascal and FORTRAN are listed in the table below:

Pascal type	FORTRAN Type	Size	Alignment
shortreal	REAL	4 bytes	2 bytes
real	DOUBLE PRECISION	8 bytes	2 bytes
longreal	DOUBLE PRECISION	8 bytes	2 bytes
integer	INTEGER*4	4 bytes	2 bytes
-32768..32767	INTEGER*2	2 bytes	2 bytes
-128..127	CHARACTER	1 byte	1 byte
boolean	CHARACTER	1 byte	1 byte
char	CHARACTER	1 byte	1 byte
array	array (*)	—	2 bytes

(*) Only one-dimensional arrays are compatible in Pascal and FORTRAN.

7.6. Incompatible Types in Pascal and FORTRAN

7.6.1. Pascal Boolean vs. FORTRAN LOGICAL

In Sun Pascal, Booleans are allocated a single byte, and may reside at odd-byte addresses. In FORTRAN, LOGICAL is defined to be the same size as the default size of INTEGER, which may be 2 or 4 bytes, but is never a single byte and is always word-aligned.

FORTRAN LOGICAL parameters should be declared at the calling site as integers. Boolean values can be passed using the standard function `ord`. For example:

The caller (Pascal):

(WRONG):

```
procedure foo(flag: boolean);    {ERROR}
external fortran;
...
foo(n>0);
```

(RIGHT):

```
procedure foo(flag: integer);
external fortran;
...
foo(ord(n>0));
```

The callee (FORTRAN):

```
subroutine foo(flag)
logical flag
...
```

7.6.2. Multidimensional Arrays

Multidimensional arrays are not compatible in Pascal and FORTRAN. Since Pascal arrays use row-major indexing and FORTRAN arrays use column-major indexing, an array passed in either direction between Pascal and FORTRAN appears to be transposed. For example:

The caller (Pascal):

```
program example(output);

type
    matrix = array [1..5, 1..5] of integer;

var
    a: matrix;
    i, j: integer;

    procedure fort(var a: matrix);
    external fortran;

begin
    for i := 1 to 5 do begin
        for j := 1 to 5 do begin
            a[i,j] := i;
            write(a[i, j]:3);
        end;
        writeln
    end;
    writeln;
    fort(a)
end.
```

The callee (FORTRAN):

```
        subroutine fort(a)
        integer a
        dimension a(5,5)
        integer i,j
        do 10 i = 1,5
            print *,(a(i,j),j = 1,5)
10      continue
        return
        end

output:
-------

        1  1  1  1  1
        2  2  2  2  2
        3  3  3  3  3
        4  4  4  4  4
        5  5  5  5  5

        1  2  3  4  5
        1  2  3  4  5
        1  2  3  4  5
        1  2  3  4  5
        1  2  3  4  5
```

Chapter 8

Sun Extensions to Berkeley Pascal

Sun Microsystems has made many extensions to Berkeley Pascal. These are exclusively language extensions (as opposed to new tools such as *dbx*). In addition, this section discusses the differences between the ISO standard and Sun Pascal.

8.1. Language Extensions Supported by both *pc* and *pi*

8.1.1. Underscores Allowed In Identifiers

The syntax of Pascal identifiers is extended to allow underscores ("_") in all character positions except the first. This improves readability of long identifiers, and for *pc*, allows access to more routines of the Sun libraries than do earlier versions.

8.1.2. Conformant Array Parameters

Level 1 ISO Pascal Standard requires that conformant array parameters be supported. This feature allows a procedure or function to accept arrays with a common element type, but with different bounds. Note that conformant arrays are not truly dynamic — that is, their bounds cannot be altered. They merely provide a mechanism for including subscript bounds information when an array is passed as a formal parameter. For example, the following function computes the real inner product of two real vectors *x* and *y*. *x* and *y* must have the same dimension.

```
function innerproduct (var x,y: array [lb..ub: integer] of real): real;
   var sum: real;
       n:integer;
   begin
     sum := 0.0;
     for n := lb to ub do
        sum := sum + x[n]*y[n];
     innerproduct := sum;
   end;
```

8.1.2.1. Syntax

```
<conformant-array-parameter-specification>
        ::= <value-conformant-array-parameter-specification>
        ::= <variable-conformant-array-parameter-specification>

<value-conformant-array-parameter-specification>
        ::= <identifier-list> ":" <conformant-array-schema>

<variable-conformant-array-parameter-specification>
        ::= "var" <identifier-list> ":" <conformant-array-schema>

<conformant-array-schema>
        ::= <packed conformant-array-schema>
        ::= <unpacked conformant-array-schema>

<packed conformant-array-schema>
        ::= "packed" "array" "[" <index-type-specification> "]"
                "of" <type-identifier>

<unpacked conformant-array-schema>
        ::= "array" "[" <index-type-specification> "]"
                "of" <array-element-type>

<array element-type>
        ::= <type-identifier>
        ::= <conformant-array-schema>

<index-type-specification>
        ::= <bound-identifier> ".." <bound-identifier>
                ":" <type-identifier>
```

A formal conformant array parameter includes read-only bound identifiers as part of its definition. The bound identifiers provide the lower and upper limits of the conformant array parameter's index type. The actual array associated with a conformant array parameter must have the same element type as the conformant array, as well as a compatible index type. When an actual array is passed as a conformant array parameter, its bounds become the bounds of the conformant array parameter. If the formal parameter is a value conformant array parameter, a copy of the actual array is made in the caller's environment and the address of the copy is passed.

A detailed description of conformant array parameters is given in Cooper[1].

8.1.3. Otherwise clause in case statement

Case statements may specify a default action or "otherwise clause", according to the following syntax:

```
<case statement>
        ::= "case" <case selector> "of"
                <case-element> { ";" <case-element> } [";"]
                [<otherwise-clause>]
                "end"

<case selector>
        ::= <expression>

<case-element>
        ::= <case label> { "," <case label> } ":" <statement>

<case-label>
        ::= <constant>

<otherwise-clause>
        ::= "otherwise" <statement> [";"]
```

Note that the reserved word "otherwise" is not a case label, so is not followed by a ":". If specified, it must be at the end of the case statement. For example,

```
program silly (input,output);
var ch:char;
begin
   read(ch);
   case ch of
   '0', '1', '2', '3', '4', '5', '6', '7', '8', '9':
        writeln ('digit');
   otherwise
        writeln ('not a digit')
   end
end.
```

The default action (i.e., the statement immediately following the reserved word "otherwise") is executed if the case selector does not match any of the specified case label values. Without the "otherwise" clause, this situation would result in a run-time error and termination of the program.

8.1.4. `sizeof` *operator*

The `sizeof` operator returns the size of a specified type or variable. If you wish to compute the size of a variant record type, an optional list of variant tag values can be used to specify a particular variant of the given record type, similar to the standard procedures `new` and `dispose`. For example,

```
program showsize (output);
type thing = record
   case boolean of
   true: (n: integer);
   false: (x: real)
   end;
var t: thing;

begin
   writeln (sizeof (t));
   writeln (sizeof (thing));
   writeln (sizeof (thing, true));
   end.
```

The syntax of `sizeof()` is

```
"sizeof" "(" <size_expr_list> ")"
```

where `<size_expr_list> ::= <size_expr> { "," <constant expression> }`
`<size_expr> ::= <type identifier> | <variable>`

`sizeof()` returns the size, in bytes, of a declared type or a (possibly qualified) variable. If an optional list of constant expressions is supplied, the <type identifier> or <variable> must denote a variant record type or an instance of one; the value returned is the size of the variant selected by the list of constant expressions (e.g., the standard procedures `new` and `dispose`).

`sizeof()` is a compile-time function and does not cause code generation other than the generation of the constant value it returns. Since the size of a conformant array parameter is not known until runtime, `sizeof(`*conformant array parameter*`)` is treated as an error. For the one-dimensional conformant array parameter,

```
function size(
    var arr: array[lb..ub: integer] of element
): integer;
```

the size of `arr` may be computed as

```
size := (ub-lb+1) * sizeof(element);
```

Apply the above formula recursively to compute the size of multi-dimensional arrays.

Note: `sizeof` is now a reserved word, unless the **-s** option (compile only standard Pascal) is in effect.

8.1.5. Correct handling of multidimensional array declarations

As specified by the ISO standard, arrays of arrays and multi-dimensional arrays are treated the same. For example,

```
array[1..10] of array[1..6] of real;
```

and

```
array[1..10,1..6] of real;
```

are treated as equivalent, as are `a[i][j]` and `a[i,j]`. In the 4.2BSD versions of *pc* and *pi* substitution of one for the other was considered an error.

8.2. Language extensions supported only by *pc*

8.2.1. Shortreal and Longreal types (pc only)

Example:

```
var x: shortreal;
    y: longreal;
    z: real;    { same as longreal }
```

Description:

pc now supports both single- and double-precision floating point types, which are denoted by the names `shortreal` and `longreal`, respectively. The standard type `real` denotes double-precision floating point, as in earlier versions of Sun Pascal. Note that `real` can be redeclared as either `longreal` or `shortreal`, if desired.

The rules for arithmetic conversions are changed to permit computations involving single precision operands to be done in single precision. The new rules are

- The operators +, -, *

  Let `op` denote a binary arithmetic operator in the set {+,-,*}. Let `x1` and `x2` denote the operands of `op`. Let `t1` and `t2` denote the types of `x1` and `x2`, respectively. The type of (`x1 op x2`) is determined by applying the following rules in the order listed:

  1. If `t1` and `t2` are both subranges of -128..127, the type of the result is -32768..32767.
  2. If `t1` and
     `result is integer.`
     `re both subranges of integer, then the type of the`
  3. If `t1` or `t2` is `longreal`, then the type of the result is `longreal`.
  4. If `t1` and `t2` are either `shortreal` or subranges of -32768..32767 (i.e., representable exactly in 16 bits), then the type of the result is `shortreal`.
  5. Otherwise, the type of the result is `longreal`.

- The integral dividing operators `div, mod`:

  If the operator is one of {`div, mod`}, then the operands are restricted to integral types, and the type of the result is `integer`.

- The operator /:

  If the conversions described above for +, -, * return an integral type, then both operands are converted to `longreal`, and that is the type of the result.

Note: These rules differ from those of C, which automatically forces conversion to double precision for all floating-point arithmetic operations, as well as for floating-point function arguments.

8.2.2. External FORTRAN and C Declarations (pc only)

The `external` directive for procedure and function declarations is extended to allow the optional specification of the source language of a separately compiled procedure or function.

```
<procedure declaration>
        ::= <procedure or function heading> <directive>

<directive>
        ::= "forward"
        ::= "external" [<identifier>] ";"

        where either "fortran" or "c" may be substituted for
        <identifier>.
```

The directives `external fortran` and `external c`, direct *pc* to generate calling sequences compatible with Sun's FORTRAN 77 and C, respectively.

For routines declared `external fortran`, the changes in the calling sequence are as follows:

- For value parameters, the compiler creates a copy of the actual argument's value in the caller's environment, and a pointer to the temporary is passed on the stack. Thus,, you don't need to create (otherwise useless) temporary variables.
- The compiler appends an underscore to the name of the external procedure to conform to a naming convention of the *f77* compiler. Note that names of Pascal procedures called from FORTRAN must supply their own trailing ("_"). This may be done using a `#define` preprocessor declaration to minimize impact on the rest of the program.

Note: Multidimensional Pascal arrays are not compatible with FORTRAN arrays. Since FORTRAN uses column-major ordering, a multidimensional Pascal array passed to FORTRAN may appear to be transposed.

For routines declared `external c`, the only changes in the calling sequence is that value parameters of type `shortreal` are treated as `longreal`.

8.2.3. Bit Operations on Integral Types

```
x := land(y,z);         { bitwise AND           }
x := lor(y,z);          { inclusive OR          }
x := xor(y,z);          { exclusive OR          }
x := lnot(y);           { bitwise NOT           }
x := lsl(y,z);          { logical shift left    }
x := lsr(y,z);          { logical shift right   }
x := asl(y,z);          { arithmetic shift left }
x := asr(y,z);          { arithmetic shift right }
```

These predefined functions provide access to the same bit operations provided by C. Each takes one or two arguments of integral type and returns a result whose type is the larger of the two operand types. The result is computed in-line, producing faster and smaller code than an equivalent external function call.

8.2.4. Preprocessor facilities (pc only)

Preprocessor facilities (e.g., conditional compilation, macros) can be used as in C (see *cpp*). You should be cautioned that comments containing a '#' in column 1 are interpreted by the preprocessor. Also, Sun's C language reserves the symbols "sun", "unix", and "mc68000", which are not reserved in Pascal.

8.2.5. Version identification

The version of *pc* used to compile a given object file can be identified by the following command line:

```
hostname% nm -ap <file> | grep " PC " | head
```

The first line containing the string "PC" indicates the version of the compiler used and the date of its generation, for example:

```
hostname% nm -ap foo.o | grep " PC " | head
00000000 - 00 000d    PC 3.4 (10/4/84)
00000001 - 00 0001    PC foo.p
                         ...
```

8.3. Differences from the ISO Pascal Standard

The following section describes the differences between the ISO Pascal standard and Sun Pascal.

- Operands of binary set operators {*,+,-} are required to have identical types. The standard permits different types as long as the base types are compatible.
- According to the standard, the expression [maxint...-maxint] should be equivalent to [ ]. *pc* and `pi` both refuse to evaluate sets with elements larger than indicated by the definition of type `intset` (predefined as set of 0...127).
- Sun Pascal treats files declared as '`file of char`' the same as files declared as '`text`.' In ISO Pascal the two types are distinct.
- The value of (`m mod n`) is not computed correctly for negative values of m. According to the standard, the divisor must be positive and the result must be negative. The correct result can be obtained by:

```
    result := m mod n;
    if result < 0 then result := result + n;
```

Appendix A

Pascal Language Reference Summary

This appendix is a condensed language reference summary for Berkeley Pascal with Sun extensions. BNF notation is used throughout.

A.1. Programs

<program> ::= *<program heading>* *<declaration list>* *<block>* **.**
  | *<declaration list>*

<program heading> ::= **program** *<identifier>* **(** *<identifier list>* **)** **;**
  | **program** *<identifier>* **;**

<block> ::= **begin** *<statement list>* **end**

A.2. Declarations

<declaration list> ::= *<declaration list>* *<declaration>*
  | *<empty>*

<declaration> ::= *<label declaration>*
  | *<constant declaration>*
  | *<type declaration>*
  | *<variable declaration>*
  | *<procedure declaration>*
  | *<function declaration>*

A.2.1. Label Declarations

<label declaration> ::= **label** *<label list>* **;**

<label list> ::= *<label>*
  | *<label list>* **,** *<label>*

<label> ::= *<unsigned integer>*

A.2.2. Constant Declarations

<constant declaration> ::= **const** *<identifier>* = *<constant>* ;
  | *<constant declaration>* *<identifier>* = *<constant>* ;

A.2.3. Type Declarations

<type declaration> ::= **type** *<identifier>* = *<type>* ;
  | *<type declaration>* *<identifier>* = *<type>* ;

A.2.4. Variable Declarations

<variable declaration> ::= **var** *<identifier list>* : *<type>* ;
  | *<variable declaration>* *<identifier list>* : *<type>* ;

A.2.5. Procedure And Function Declarations

<procedure declaration> ::= *<procedure heading>* **forward** ;
  | *<procedure heading>* **external** *<identifier>* ;
  | *<procedure heading>* **external** ;
  | *<procedure heading>* *<declaration list>* *<block>* ;

<function declaration> ::= *<function heading>* **forward** ;
  | *<function heading>* **external** *<identifier>* ;
  | *<function heading>* **external** ;
  | *<function heading>* *<declaration list>* *<block>* ;

<procedure heading> ::= **procedure** *<identifier>* *<parameters>* ;

<function heading> ::= **function** *<identifier>* *<parameters>* : *<type identifier>* ;

<parameters> ::= (*<parameter list>*)
  | *<empty>*

A.2.6. Formal Parameter Declarations

<parameter list> ::= *<parameter>*
  | *<parameter list>* ; *<parameter>*

<parameter> ::= *<identifier list>* : *<parameter type>*
  | **var** *<identifier list>* : *<parameter type>*
  | **function** *<identifier>* *<parameters>* : *<type identifier>*
  | **procedure** *<identifier>* *<parameters>*

```
<parameter type> ::= <type identifier>
        |  <conformant array schema>
        |  <packed conformant array schema>

<conformant array schema> ::= array [ <index type list> ] of <parameter type>

<packed conformant array schema> ::= packed array [ <index type> ] of <type identifier>

<index type list> ::= <index type>
        |  <index type list> ; <index type>

<index type> ::= <identifier> .. <identifier> : <type identifier>
```

A.3. Constants

```
<constant> ::= <character string>
        |  <constant identifier>
        |  <number>
        |  + <number>
        |  - <number>

<number> ::= <unsigned integer>
        |  <octal constant>
        |  <unsigned real constant>

<constant list> ::= <constant>
        |  <constant list> , <constant>
```

A.4. Types

```
<type> ::= <simple type>
        |  ^ <type identifier>
        |  <structured type>
        |  packed <structured type>

<simple type> ::= <type identifier>
        |  ( <identifier list> )
        |  <constant> .. <constant>

<structured type> ::= array [ <simple type list> ] of <type>
        |  file of <type>
        |  set of <simple type>
        |  record <field list> end

<simple type list> ::= <simple type>
        |  <simple type list> , <simple type>
```

A.5. Record Types

```
<field list> ::= <fixed part> <variant part>

<fixed part> ::= <field>
      |  <fixed part> ; <field>

<field> ::= <empty>
      |  <identifier list> : <type>

<variant part> ::= <empty>
      |  case <type identifier> of <variant list>
      |  case <identifier> : <type identifier> of <variant list>

<variant list> ::= <variant>
      |  <variant list> ; <variant>

<variant> ::= <empty>
      |  <constant list> : ( <field list> )
```

A.6. Statements

```
<statement> ::= <empty>
      |  <unsigned integer> : <statement>
      |  <procedure identifier>
      |  <procedure identifier> ( <actual parameter list> )
      |  <assignment>
      |  begin <statement list> end
      |  case <expression> of <case statement list> end
      |  with <variable list> do <statement>
      |  while <expression> do <statement>
      |  repeat <statement list> until <expression>
      |  for <assignment> to <expression> do <statement>
      |  for <assignment> downto <expression> do <statement>
      |  goto <label>
      |  if <expression> then <statement>
      |  if <expression> then <statement> else <statement>

<assignment> ::= <variable> := <expression>

<statement list> ::= <statement>
      |  <statement list> ; <statement>

<case statement list> ::= <case list element>
      |  <case statement list> ; <case list element>
```

```
<case list element> ::= <constant list> : <statement>
        |   otherwise <statement>
        |   <empty>
```

A.7. Expressions

```
<expression> ::= <simple expression>
        |   <expression> <relational operator> <simple expression>

<simple expression> ::= <signed term>
        |   <simple expression> <adding operator> <signed term>

<signed term> ::= <term>
        |   + <signed term>
        |   - <signed term>

<term> ::= <factor>
        |   <term> <multiplying operator> <factor>

<factor> ::= nil
        |   <character string>
        |   <unsigned integer>
        |   <octal constant>
        |   <unsigned real constant>
        |   <variable>
        |   <function identifier> ( <actual parameter list> )
        |   ( <expression> )
        |   not <factor>
        |   [ <set element list> ]
        |   [ ]
        |   sizeof ( <sizeof argument list> )

<sizeof argument list> ::= <sizeof argument>
        |   <sizeof argument list> , <expression>

<sizeof argument> ::= <type identifier>
        |   <variable>

<set element list> ::= <set element>
        |   <set element list> , <set element>

<set element> ::= <expression>
        |   <expression> .. <expression>
```

A.8. Variables

```
<variable> ::= <identifier>
      |   <qualified variable>

<qualified variable> ::= <array identifier> [ <expression list> ]
      |   <qualified variable> [ <expression list> ]
      |   <record identifier> . <field identifier>
      |   <qualified variable> . <field identifier>
      |   <pointer identifier> ^
      |   <qualified variable> ^
```

A.9. Actual Parameters

```
<actual parameter> ::= <expression>
      |   <expression> : <expression>
      |   <expression> : <expression> : <expression>
      |   <expression> <write base>
      |   <expression> : <expression> <write base>

<expression list> ::= <expression>
      |   <expression list> , <expression>

<actual parameter list> ::= <actual parameter>
      |   <actual parameter list> , <actual parameter>

<write base> ::= oct  |  hex
```

A.10. Operators

```
<relational operator> ::=  =  |  <  |  >  |  <>  |  <=  |  >=  |  in

<adding operator> ::=  +  |-|  or  ¦ ¦

<multiplying operator> ::=  *  |  /  |  div  |  mod  |  and  |  &
```

A.11. Miscellaneous

```
<variable list> ::= <variable>
      |   <variable list> , <variable>

<identifier list> ::= <identifier>
      |   <identifier list> , <identifier>

<empty> ::=
```

A.12. Lexicon

```
<constant identifier> ::= <identifier>

<type identifier> ::= <identifier>

<var identifier> ::= <identifier>

<array identifier> ::= <identifier>

<pointer identifier> ::= <identifier>

<record identifier> ::= <identifier>

<field identifier> ::= <identifier>

<procedure identifier> ::= <identifier>

<function identifier> ::= <identifier>

<identifier> ::= <letter> { <letter> | <digit> | _}

<letter> ::=
          a | b | c | d | e | f | g | h | i | j | k | l | m
        | n | o | p | q | r | s | t | u | v | w | x | y | z
        | A | B | C | D | E | F | G | H | I | J | K | L | M
        | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<unsigned integer> ::= <digit> { <digit> }

<signed integer> ::= <unsigned integer>
      | + <unsigned integer>
      | - <unsigned integer>

<unsigned real constant> ::= <unsigned integer> . <fractional part>
      | <unsigned integer> . <fractional part> e <scale factor>
      | <unsigned integer> . <fractional part> E <scale factor>
      | <unsigned integer> e <scale factor>
      | <unsigned integer> E <scale factor>

<fractional part> ::= <digit> { <digit> }

<scale factor> ::= <signed integer>

<octal constant> ::= <octal digit> { <octal digit> } b
      | <octal digit> { <octal digit> } B

<octal digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
```

<*character string*> ::= ' <*string element*> { <*string element*> } '

<*string element*> ::= <*apostrophe image*> | <*any character except apostrophe or newln*>

<*apostrophe image*> ::= ''

Appendix B

Differences Between Berkeley Pascal and Standard Pascal

The official Pascal standard is "Specification for the Computer Programming Language Pascal" (ISO dp7185).

This section summarizes extensions to the language, discusses the ways in which the undefined specifications were resolved, gives limitations and restrictions of the current implementation, and lists the predefined functions and procedures available in the Berkeley implementation. Sun extensions to the language are listed in Appendix B.

B.1. Extensions to Pascal

This section defines nonstandard language constructs available in Berkeley Pascal. The **−s** standard Pascal option of the translators `pi` and *pc* can be used to detect these extensions in programs that are to be transported.

B.1.1. String Padding

Berkeley Pascal pads constant strings with blanks found in expressions and as value parameters, making them as long as is required. The following is a legal Berkeley Pascal program:

```
program x(output);
var z : packed array [ 1 .. 13 ] of char;
begin
    z := 'red';
    writeln(z)
end;
```

The blanks are added on the right. Thus, the assignment above is equivalent to

```
z := 'red          '
```

which is standard Pascal.

B.1.2. Octal Constants, Octal and Hexadecimal Write

Octal constants may be given as a sequence of octal digits followed by the character 'b' or 'B'. The forms

```
write(a:n oct)
```

and

```
write(a:n hex)
```

cause the internal representation of expression *a* (which must be Boolean, character, integer, pointer, or a user-defined enumerated type) to be written in octal or hexadecimal, respectively.

B.1.3. Assert Statement

An `assert` statement causes a Boolean expression to be evaluated each time the statement is executed. A run-time error results if any of the expressions evaluates to FALSE. The `assert` statement is treated as a comment if run-time tests are disabled. The syntax for `assert` is

```
assert <expr>
```

where *expr* is a Boolean expression.

B.1.4. Enumerated Type Input/Output

Enumerated types can be read and written. On output, the string name associated with the enumerated value is output. If the value is out of range, a run-time error occurs. On input an identifier is read and looked up in a table of names associated with the type of the variable, and the appropriate internal value is assigned to the variable being read. If the name is not found in the table, a run-time error occurs.

B.1.5. Structure-Returning Functions

An extension has been added that allows functions to return arbitrary-sized structures, rather than just scalars as in the standard.

B.1.6. Separate Compilation

The compiler *pc* has been extended to allow separate compilation of programs. Procedures and functions declared at the global level can be compiled separately. Type checking of calls to separately compiled routines is performed at load time to insure that the program as a whole is consistent. See the section "Separate compilation with *pc*," in Chapter 4 for details.

B.2. Implementation Dependent Features

This section describes implementation dependent features of Pascal, which are undefined by the standard.

B.2.1. File Name — File Variable Associations

Each Pascal file variable is associated with a named UNIX file. Except for ***input*** and ***output***, which are exceptions to some of the rules, a name can become associated with a file in any of three ways:

- If a global Pascal file variable appears in the `program` statement then it is associated with the UNIX file of the same name.
- If a Pascal file is reset or rewritten using the extended two-argument form of ***reset*** or ***rewrite*** then the given name is associated.
- If a Pascal file that has never had a UNIX name associated with it is reset or rewritten without specifying a name via the second argument, then a temporary name of the form `tmp.n` is associated with the file. Temporary names start with `tmp.1` and continue by incrementing ***n*** in ASCII order. Temporary files are removed automatically when their scopes are exited.

B.2.2. The Program Statement

The syntax of the `program` statement is (in extended BNF)[10]

```
    program <id> ( <file id> { , <file id > } ) ;
```

The file identifiers (other than `input` and `output`) must be declared as variables of type `file` in the global declaration part.

B.2.3. The Files Input and Output

The formal parameters `input` and `output` are associated with the UNIX standard input and output and have a somewhat special status. The following rules must be noted:

- The program heading must contain the formal parameter `output` if it does any output. If `input` is used (explicitly or implicitly) then it must also be declared here.
- Unlike all other files, the Pascal files `input` and `output` must not be defined in a declaration, since their declaration is automatically done, as in

```
        var input, output: text
```

- The procedure `reset` may be used on `input`. If no UNIX filename has ever been associated with `input`, and no filename is given, then an attempt is made to 'rewind' `input`. If this fails, a run-time error occurs. `rewrite` calls to `output` work as any other file, except that `output` has no associated file initially. This means that a simple

[10] For an explanation of extended BNF notation see Cooper[1], page 2.

```
rewrite(output)
```

associates a temporary name with `output`.

B.2.4. *Details For Files*

If a file other than `input` is read, then reading must be initiated by a call to the procedure `reset`, which causes the Pascal system to attempt to open the associated UNIX file for reading. If this fails, then a run-time error occurs. Writing a file other than `output` must be initiated by a `rewrite` call, which causes the Pascal system to create the associated UNIX file and then to open it for writing only.

B.2.5. *Buffering*

The buffering for `output` is determined by the value of the **−b** option at the end of the `program` statement. If it has its default value 1, then `output` is buffered in blocks of up to 1024 characters, and is flushed whenever a `writeln` occurs and at each reference to the file `input`. If it has the value 0, `output` is unbuffered. Any value of 2 or more gives block buffering without line or input-reference flushing. All other output files are always buffered in blocks of 1024 characters. All output buffers are flushed when the files are closed at scope exit or whenever the procedure `message` is called, and can be flushed using the built-in procedure `flush`.

An important point for an interactive implementation is the definition of 'input↑'. If `input` is a terminal, and the Pascal system reads a character at the beginning of execution to define 'input↑', then no prompt can be printed by the program before the user is required to type some input. For this reason, 'input↑' is not defined by the system until its definition is needed, with reading from a file occurring only when necessary.

B.2.6. *The Character Set*

Seven-bit ASCII is the character set used in UNIX. The standard Pascal symbols `and`, `or`, `not`, <=, >=, <>, and the up arrow '↑' (for pointer qualification) are recognized.[11] Less portable are the synonyms tilde '~' (for `not`), '&' (for `and`), and '¦' (for `or`).

Upper and lower case are considered to be distinct.[12] Keywords and built-in `procedure` and `function` names are composed of all lower-case letters. Thus the identifiers GOTO and GOto are distinct both from each other and from the keyword `goto`. The standard type `boolean` is also available as `Boolean`.

Character strings and constants may be delimited by the character ', or by the character '#'; the latter is sometimes convenient when programs are to be transported. Note that the '#' character has special meaning when it is the first character on a line — see "Multi-file programs" in Chapter 4.

[11] On many terminals and printers, the up arrow is represented as a circumflex '^'. These are not distinct characters, but rather different graphic representations of the same internal codes.

[12] The ISO standard for Pascal considers them to be the same. The **−s** and **−L** options of *pc*, `pi`, *pxp* consider upper- and lower-case letters to be equivalent in identifiers and keywords.

B.2.7. The Standard Types

The standard type `integer` is conceptually defined as

```
type integer = minint .. maxint;
```

`Integer` is implemented with 32-bit two's-complement arithmetic. The predefined constants of type `integer` are

```
const maxint = 2147483647; minint = -2147483648;
```

The standard type `char` is conceptually defined as

```
type char = minchar .. maxchar;
```

The built-in character constants are `minchar` and `maxchar`, `bell` and `tab`; `ord(minchar)` = 0, `ord(maxchar)` = 127.

The types `real` and `longreal` are implemented using 64-bit IEEE floating-point format and the type `shortreal` using 32-bit floating-point format. The floating-point arithmetic for `real` and `longreal` is done in ***round to nearest*** mode, and provides approximately 16 digits of precision with numbers whose magnitudes range from 5.0E-324 to 1.7E308.

Pascal `shortreal` is the same as `real*4` in Fortran and `float` in C. The maximum and minimum representable values are

```
(max) 3.402823e+38
(min) 1.175494e-38
```

Values of type `shortreal` can represent about 7 decimal digits.

B.2.8. Comments

Comments can be delimited by either '{' and '}' or by '(*' and '*)' (as opposed to the standard which does not distinguish between them). If the character '{' appears in a comment delimited by '{' and '}', a warning diagnostic is printed. A similar warning is printed if the sequence '(*' appears in a comment delimited by '(*' and '*)'. The restriction implied by this warning is not part of standard Pascal, but detects many otherwise subtle errors.

You can convert parts of your program to comments without generating an error diagnostic with the following:

```
{ This is a comment enclosing a piece of program
a := functioncall;        (* comment within comment *)
procedurecall;
lhs := rhs;               (* another comment*)
}
```

By using one kind of comment exclusively in your program you can use the other delimiters when you need to comment out parts of your program. In this way, you also allow the translator to help by detecting statements accidently placed within comments.

If a comment does not terminate before the end of the input file, the translator points to the beginning of the comment, indicating that the comment is not terminated. In this case, processing terminates immediately. See the discussion of "QUIT" in Chapter 2.

B.2.9. Option Control

Translator options can be controlled in two distinct ways: on the command line, and in comments. Several options may appear on the command line invoking the translator. These options are given as one or more strings of letters preceded by the character '–' and cause the nondefault setting of each given option to be used. This method of options communication is expected to predominate for UNIX. Thus the command

```
        hostname% pi -l -s foo.p
```

translates the file `foo.p` with the listing option enabled (it's normally off) and only with standard Pascal features available.

If you require more control over the portions of the program where options are enabled then option control in comments can and should be used. Place a '$' as the first character of the comment followed by a comma-separated list of directives. Thus, the following is an equivalent to the command line example given above:

```
        {$l+,s+ listing on, standard Pascal}
```

as the first line of the program. The **–l** option is more appropriately specified on the command line, since in an interactive environment it is unlikely to want a listing of the program each time it is translated.

Most directives consist of a letter designating the option, followed either by a '+' to turn the option on, or by a '–' to turn the option off. The only exception is the **–b** option which takes a single digit instead of a '+' or '–'.

B.2.10. Notes on the Listings

The first page of a listing includes a banner line indicating the version number and generation date of *pi* or *pc*. It also includes the UNIX pathname supplied for the source file and the date of last modification of that file.

Within the body of the listing, lines are numbered consecutively and correspond to the line numbers used by the vi editor. Currently, two special kinds of lines may be used to format the listing: a line consisting of a formfeed character (Control-L), which causes a page eject in the listing, and a line with no characters, which causes the line number to be suppressed in the listing (creating a blank line). These lines correspond to 'eject' and 'space' macros found in many assemblers. Nonprinting characters are printed as the character '?' in the listing.[13]

B.2.11. The Standard Procedure Write

If no minimum field length parameter is specified for a ***write***, the following default values are assumed:[14]

integer	10
real	22

[13] The character generated by a Control-I indents to the next tab stop. Tab stops are set every 8 columns in UNIX. Thus, tabs provide a quick way of indenting in the program.

[14] The default values for the Sun-provided types *shortreal* and *longreal* are 14 and 22, respectively,

Boolean	length of 'true' or 'false'
char	1
string	length of the string
oct	11
hex	8

The end of each line in a text file should be indicated explicitly by `writeln(f)`, where `writeln(output)` may be written simply as `writeln`. The built-in function `page(f)` puts a single ASCII formfeed character on the output file.

B.3. Restrictions and Limitations

B.3.1. Files

Files cannot be members of files or dynamically-allocated structures.

B.3.2. Arrays, Sets, and Strings

For `pi` only, the calculations involving array subscripts and set elements are done with 16-bit arithmetic. This restricts the types with which arrays and sets may be defined. The lower bound of such a range must be greater than or equal to −32768, and the upper bound less than or equal to 32767. Strings may have any length from 1 to 65535 characters, and sets may contain no more than 65535 elements. The range of elements in a set expression is bound by the definition of ***intset*** in the current scope. Refer to the section on "Additional Predefined Types" later in this chapter for more information.

In ***pc***, arrays may be any size that fits into the process address space.

B.3.3. Line and Symbol Length

There is no intrinsic limit on the length of identifiers. Identifiers are considered to be distinct if they differ in any position over their entire length. The limit on the maximum input line length is currently 1024 characters.

B.3.4. Procedure and Function Nesting and Program Size

A maximum 20 levels of procedure and function nesting are allowed. There is no fundamental, translator-defined limit on the size of the program that can be translated. The ultimate limit is supplied by the address space. If you encounter the 'ran out of memory' diagnostic, the program can still be translated if smaller procedures are used, since a lot of space is freed by the translator at the completion of each procedure or function in the current implementation.

In ***pi***, there is an implementation-defined limit of 65536 bytes per variable, but no limit on the number of variables.

B.3.5. Overflow

There is currently no checking for overflow on arithmetic operations at runtime.

B.4. Added Types, Operators, Procedures and Functions

B.4.1. Additional Predefined Types

The type `alfa` is predefined as

```
    type alfa = packed array [ 1..10 ] of char
```

The type `intset` is predefined as

```
    type intset = set of 0..127
```

In most cases, the context of an expression involving a constant set allows the translator to determine the type of the set, even though the constant set itself may not uniquely determine this type. In the cases where it is not possible to determine the type of the set from local context, the expression type defaults to a set over the entire base type unless the base type is integer[15]. In the latter case, the type defaults to the current binding of `intset`, which must be "type set of (a subrange of) integer" at that point.

B.4.2. Additional Predefined Operators

The relationals '<' and '>' of proper set inclusion are available.

B.4.3. Nonstandard Procedures

`argv(i,a)`	assigns the (possibly truncated or blank-padded) *i'th* argument of the invocation of the current UNIX process to the variable *a*, where *i* is an integer and *a* is a string variable. The range of *i* is *0* to *argc−1*.
`date(a)`	assigns the current date to the `alfa` variable *a* in the format *dd mmm yy*, where *mmm* is the first three characters of the month, (e.g., 'Apr').
`flush(f)`	writes the output buffered for Pascal file *f* into the associated UNIX file.
`halt`	terminates the execution of the program with a control flow traceback.
`linelimit(f,x)`	causes the program to be abnormally terminated if more than *x* lines are written on file *f*, where *f* is a textfile and *x* an integer expression. If *x* is less than 0 then no limit is imposed.
`message(x,...)`	causes the parameters (which have the same format as the built-in procedure

15 The translators make a special case of the construct 'if ... in [...]' and enforces only the more lax restriction on 16-bit arithmetic given above in this case.

	`write`), to be written unbuffered on `stderr`, which is usually the user's terminal.
`null`	a procedure of no arguments which does absolutely nothing. It is useful as a place holder, and is generated by *pxp* in place of the invisible empty statement.
`remove(a)`	causes the UNIX file whose name is given by the string *a*, with trailing blanks eliminated, to be removed.
`reset(f,a)`	causes the file *a* (where *a* is a string with blanks trimmed) to be associated with *f*, and performs a *reset* on *f*.
`rewrite(f,a)`	is analogous to 'reset' above.
`stlimit(i)`	sets the statement limit to be *i* statements, where *i* is an integer. Specifying the **−p** option to *pc* disables statement limit counting.
`time(a)`	causes the current time in the form ' hh:mm:ss ' to be assigned to the `alfa` variable *a*.

B.4.4. Nonstandard Functions

`argc`	returns the count of arguments provided when the Pascal program was invoked. *Argc* is always at least 1.
`card(x)`	returns the cardinality of the set *x*, that is, the number of elements in the set.
`clock`	returns an integer representing the system time (in milliseconds) used by this process.
`expo(x)`	yields the integer-valued exponent in the floating-point representation of *x*; `expo(x) = entier(log2(abs(x)))`.
`random(x)`	invokes a linear congruential random number generator, where *x* is a real parameter that is evaluated but otherwise ignored. Successive seeds are generated as `(seed*a + c) mod m` and the new random number is a normalization of the seed to the range 0.0 to 1.0; `a` is 62605, `c` is 113218009, and `m` is 536870912. The initial seed is 7774755.
`seed(i)`	sets the random number generator seed to *i* and returns the previous seed, where *i* is an integer. Thus, `seed(seed(i))` has no effect except to yield value *i*.
`sysclock`	returns the number of milliseconds of system time used by this process. `sysclock` is an integer function with no arguments.
`undefined(x)`	a Boolean function. Its argument is a real number and it always returns false.
`wallclock`	returns the time in seconds since 00:00:00 GMT January 1, 1970. `wallclock` is an integer function with no arguments.
`sizeof(x)`	see Chapter 8 for extensions specific to the Sun implementation.

Appendix C

Bibliography

[1] Cooper, Doug. *Standard Pascal User Reference Manual*, W.W. Norton and Co., 1983

[2] *Specification for Computer Programming Language Pascal*, BS 6192, British Standards Institute, 1982

[3] Jensen, K., and N. Wirth. *PASCAL User Manual and Report*, Springer-Verlag, 1974.

Appendix D

Pascal Manual Pages

NAME

pc – Pascal compiler

SYNOPSIS

pc [**–c**] [**–g**] [**–o** *output*] [**–O**] [**–b**] [**–C**] [**–fsky**] [**–H**] [**–i** *name* ...] [**–l**] [**–lpfc**] [**–L**] [**–P**] [**–s**] [**–S**] [**–z**] *filename.p* ...

DESCRIPTION

Pc is the Sun Pascal compiler. If given an argument file ending with *.p, pc* compiles the file and leaves the result in an executable file called *a.out* by default.

A program may be separated into more than one *.p* file. *Pc* will compile a number of *.p* files into object files (with the extension *.o* in place of *.p*). Object files may then be loaded into an executable *a.out* file. Exactly one object file must supply a **program** statement to successfully create an executable *a.out* file. The rest of the files must consist only of declarations which logically nest within the program. References to objects shared between separately compiled files are allowed if the objects are declared in **include**d header files, whose names must end with *.h*. Header files may only be included at the outermost level, and thus declare only globally available objects. To allow external **function**s and **procedure**s to be declared, an **external** directive has been added, whose use is similar to the **forward** directive but restricted to appear only in *.h* files. **Function** and **procedure** bodies may not appear in *.h* files. A binding phase of the compiler checks that declarations are used consistently, to enforce the type checking rules of Pascal.

Object files created by other language processors may be loaded together with object files created by *pc*. The **function**s and **procedure**s they define must have been declared in *.h* files included by all the *.p* files which call those routines.

Pascal's calling conventions are the same as in C, with **var** parameters passed by address and other parameters passed by value.

Both *pc* and *pi*(1) support ISO Level 1 Standard Pascal, including conformant array parameters. Deviations from the ISO Standard are noted under **BUGS** below.

See the *Pascal User's Manual* for details.

OPTIONS

See *ld*(1) for load-time options.

–c Suppress loading and produce *.o* file(s) from source file(s).

–g Produce additional symbol table information for the symbolic debugger *dbx*(1).

–o *name*
Name the final output file *name* instead of *a.out.*

–O Optimize the object code.

–b Buffer the file *output* in units of disk blocks, rather than lines.

–C Compile code to perform subscript and subrange checks, verify **assert** statements, and initialize all variables to zero as in *pi.* Note that pointers are not checked. This option differs significantly from the **-C** option of the *cc* compiler.

–fsky Generate code which assumes the presence of a SKY floating-point processor board. Programs compiled with this option can only be run in systems that have a SKY board installed. Programs compiled without the **–fsky** option will use the SKY board from library routines, but won't run as fast as they would if the **–fsky** option were used. If any part of a program is compiled using the **–fsky** option, you must also use this option when linking with the **pc** command, since different startup routines are used to initialize the SKY board.

–H Compile code to perform range checks on pointers into the heap.

–i *name*

Produce a listing for the specified procedures, functions and **include** files.

−l Make a program listing during translation.

−lpfc Load common startup code for programs containing mixed Pascal and FORTRAN object files. Such programs should also be loaded with the FORTRAN libraries (see **files** below).

−L Map upper case letters in keywords and identifiers to lower case.

−P Use partial evaluation semantics for the boolean operators **and** and **or**. For these operators only, left-to-right evaluation is guaranteed, and the second operand is evaluated only if necessary to determine the result.

−s Accept standard Pascal only; nonstandard constructs cause warning diagnostics.

−S Compile the named program, and leave the assembly language output on the corresponding file suffixed '.s'. No '.o' is created.

−z Allow execution profiling with *pxp* by generating statement counters, and arranging for the creation of the profile data file *pmon.out* when the resulting object is executed.

Other arguments are taken to be loader option arguments or libraries of *pc* compatible routines. Certain flags can also be controlled in comments within the program, as described in the *Pascal User's Manual* in the Sun *Pascal* Manual.

FILES

file.p	Pascal source files
/lib/cpp	macro preprocessor
/usr/lib/pc0	compiler
/lib/f1	code generator
/usr/lib/pc2	inline expander of library calls
/lib/c2	peephole optimizer
/usr/lib/pc3	separate compilation consistency checker
/usr/lib/pc3.2strings	text of the error messages
/usr/lib/how_pc	basic usage explanation
/usr/lib/libpc.a	intrinsic functions and I/O library
/usr/lib/libpfc.a	startup code for combined Pascal and FORTRAN programs
/usr/lib/libF77.a	FORTRAN intrinsics library
/usr/lib/libI77.a	FORTRAN I/O library
/usr/lib/libU77.a	FORTRAN<=>Unix interface library
/usr/lib/libm.a	math library
/lib/libc.a	standard library, see *intro*(3)

SEE ALSO

The *Pascal User's Manual* in the Sun *Pascal* Manual.
pi(1), pxp(1), pxref(1)

DIAGNOSTICS

For a basic explanation do

```
tutorial% pc
```

In the diagnostic output of the translator, lines containing syntax errors are listed with a flag indicating the point of error. Diagnostic messages indicate the action which the recovery mechanism took in order to be able to continue parsing. Some diagnostics indicate only that the input is 'malformed.' This occurs if the recovery can find no simple correction to make the input syntactically valid.

Semantic error diagnostics indicate a line in the source text near the point of error. Some errors evoke more than one diagnostic to help pinpoint the error; the follow-up messages begin with an ellipsis '...'.

The first character of each error message indicates its class:

E	Fatal error; no code will be generated.
e	Nonfatal error.
w	Warning – a potential problem.
s	Nonstandard Pascal construct warning.

If a severe error occurs which inhibits further processing, the translator will give a diagnostic and then 'QUIT'.

Names whose definitions conflict with library definitions draw a warning. The library definition will be replaced by the one supplied in the Pascal program. Note that this can have unpleasant sideeffects.

BUGS

The keyword **packed** is recognized but has no effect. The ISO standard requires packed and unpacked structures to be distinguished for portability reasons.

Binary set operators are required to have operands with identical types; the ISO standard allows different types, as long as the underlying base types are compatible.

The **–z** flag doesn't work for separately compiled files.

Because the **–s** option is usurped by the compiler, it is not possible to pass the strip option to the loader. Thus programs which are to be stripped, must be run through *strip*(1) after they are compiled.

NAME

pi – Pascal interpreter code translator

SYNOPSIS

pi [**–b**] [**–l**] [**–L**] [**–n**] [**–o** *name*] [**–p**] [**–s**] [**–t**] [**–u**] [**–w**] [**–z**]
[**–i** name ...] name.p

DESCRIPTION

Pi translates the program in the file *name.p* leaving interpreter code in the file *obj* in the current directory. The interpreter code can be executed using *px*. *Pix* performs the functions of *pi* and *px* for 'load and go' Pascal.

Both *pi* and *pc*(1) support ISO Level 1 Standard Pascal, including conformant array parameters. Deviations from the ISO Standard are noted under **BUGS** below.

OPTIONS

The following flags are interpreted by *pi;* the associated options can also be controlled in comments within the program; see the *Pascal User's Manual* in the *Sun Fortran and Pascal* Manual for details.

–b Buffer the file *output* in units of disk blocks, rather than lines.

–i *name*
Enable the listing for any specified procedures, functions, and **include** files.

–l Make a program listing during translation.

–L Map all identifiers and keywords to lower case.

–n Begin each listed **include** file on a new page with a banner line.

–o *name*
Name the final output file *name* instead of *a.out.*

–p Suppress the post-mortem control flow backtrace if an error occurs; suppress statement limit counting.

–s Accept standard Pascal only; non-standard constructs cause warning diagnostics.

–t Suppress runtime tests of subrange variables and treat **assert** statements as comments.

–u Card image mode; only the first 72 characters of input lines are used.

–w Suppress warning diagnostics.

–z Allow execution profiling with *pxp* by generating statement counters, and arranging for the creation of the profile data file *pmon.out* when the resulting object is executed.

FILES

file.p	input file
file.i	**include** file(s)
/usr/lib/pi3.*strings	text of the error messages
/usr/lib/how_pi*	basic usage explanation
obj	interpreter code output

SEE ALSO

Sun Fortran and Pascal Manual
pix(1), px(1), pxp(1), pxref(1)

DIAGNOSTICS

For a basic explanation do

```
tutorial% pi
```

In the diagnostic output of the translator, lines containing syntax errors are listed with a flag indicating the point of error. Diagnostic messages indicate the action which the recovery mechanism took in order to be able to continue parsing. Some diagnostics indicate only that the input is 'malformed.' This occurs if the recovery can find no simple correction to make the input syntactically valid.

Semantic error diagnostics indicate a line in the source text near the point of error. Some errors evoke more than one diagnostic to help pinpoint the error; the follow-up messages begin with an ellipsis '...'.

The first character of each error message indicates its class:

E	Fatal error; no code will be generated.
e	Non-fatal error.
w	Warning – a potential problem.
s	Non-standard Pascal construct warning.

If a severe error occurs which inhibits further processing, the translator will give a diagnostic and then 'QUIT'.

BUGS

The keyword **packed** is recognized but has no effect. The ISO standard requires packed and unpacked structures to be distinguished for portability reasons.

Binary set operators are required to have operands with identical types; the ISO standard allows different types, as long as the underlying base types are compatible.

For clarity, semantic errors should be flagged at an appropriate place in the source text, and multiple instances of the 'same' semantic error should be summarized at the end of a **procedure** or **function** rather than evoking many diagnostics.

When **include** files are present, diagnostics relating to the last procedure in one file may appear after the beginning of the listing of the next.

NAME

pix – Pascal translator and interpreter

SYNOPSIS

pix [options] [**–i** name ...] name.p [argument ...]

DESCRIPTION

Pix is a 'load and go' version of Pascal which combines the functions of the translator *pi* and the interpreter *px*. *Pix* uses *pi* to translate the program in the file *name.p* and, if there were no fatal errors during translation, calls *px* to execute the resulting interpretive code with the specified arguments. A temporary file is used for the object code; the file *obj* is neither created nor destroyed.

Options are as described under pi(1).

FILES

/usr/ucb/pi	Pascal translator
/usr/ucb/px	Pascal interpreter
/tmp/pix*	temporary
/usr/lib/how_pix	basic explanation

SEE ALSO

The *Pascal User's Manual* in the *Pascal for the Sun Workstation* Manual.
pi(1), px(1)

DIAGNOSTICS

For a basic explanation do

```
tutorial% pix
```

NAME

pmerge − pascal file merger

SYNOPSIS

pmerge name.p ...

DESCRIPTION

Pmerge assembles the named Pascal files into a single standard Pascal program. The resulting program is listed on the standard output. It is intended to be used to merge a collection of separately compiled modules so that they can be run through **pi**, or exported to other sites.

FILES

/usr/tmp/MG*	default temporary files

SEE ALSO

pc(1), pi(1),
Auxiliary documentation *Pascal User's Manual* in the Sun *Fortran and Pascal* Manual.

BUGS

Very minimal error checking is done, so incorrect programs will produce unpredictable results. Block comments should be placed after the keyword to which they refer or they are likely to end up in bizarre places.

NAME

px – Pascal interpreter

SYNOPSIS

px [obj [argument ...]]

DESCRIPTION

Px interprets the abstract machine code generated by *pi.* The first argument is the file to be interpreted, and defaults to *obj*; remaining arguments are available to the Pascal program using the built-ins *argv* and *argc*. *Px* is also invoked by *pix* when running 'load and go'.

If the program terminates abnormally an error message and a control flow backtrace are printed. The number of statements executed and total execution time are printed after normal termination. The **p** option of *pi* suppresses all of this except the message indicating the cause of abnormal termination.

FILES

obj	default object file
pmon.out	profile data file

SEE ALSO

The *Pascal User's Manual* in the Sun *Pascal* Manual.
pi(1), pix(1)

DIAGNOSTICS

Most run-time error messages are self-explanatory. Some of the more unusual ones are:

Reference to an inactive file
: A file other than *input* or *output* was used before a call to *reset* or *rewrite.*

Statement count limit exceeded
: The limit of 500,000 executed statements (which prevents excessive looping or recursion) has been exceeded.

Bad data found on integer read
Bad data found on real read
: Usually, non-numeric input was found for a number. For reals, Pascal requires digits before and after the decimal point so that numbers like '.1' or '21.' evoke the second diagnostic.

panic: *Some message*
: Indicates a internal inconsistency detected in *px* probably due to a Pascal system bug.

BUGS

Post-mortem traceback is not limited; infinite recursion leads to almost infinite traceback.

NAME

pxp – Pascal execution profiler

SYNOPSIS

pxp [**–acdefjLnstuw_**] [**–23456789**] [**–z** [name ...]] name.p

DESCRIPTION

Pxp can be used to obtain execution profiles of Pascal programs or as a pretty-printer. To produce an execution profile all that is necessary is to translate the program specifying the **z** option to *pc*, *pi*, or *pix*, execute the program, and then type the command

```
tutorial% pxp –z name.p
```

Pxp generates a reformatted listing if none of the **c**, **t**, or **z** options are specified; thus

```
tutorial% pxp old.p > new.p
```

places a pretty-printed version of the program in *old.p* in the file *new.p*.

OPTIONS

The use of the following options of *pxp* is discussed in the *Pascal User's Manual* in the Sun *Pascal* Manual.

–a Print the bodies of all procedures and functions in the profile; even those which were never executed.

–c Extract profile data from the file *core*.

–d Include declaration parts in a profile.

–e Eliminate **include** directives when reformatting a file; the **include** is replaced by the reformatted contents of the specified file.

–f Fully parenthesize expressions.

–j Left justify all procedures and functions.

–L Map all identifiers and keywords to lower case.

–n Eject a new page as each file is included; in profiles, print a blank line at the top of the page.

–s Strip comments from the input text.

–t Print a table summarizing **procedure** and **function** call counts.

–u Card image mode; only the first 72 characters of input lines are used.

–w Suppress warning diagnostics.

–z Generate an execution profile. If no *names* are given the profile is of the entire program. If a list of *names* is given, then only the specified **procedures** or **functions** and the contents of the specified **include** files will appear in the profile.

–_ Underline keywords.

–*d* Use *d* spaces (where *d* is a digit, $2 \leq d \leq 9$) as the basic indenting unit. The default is 4.

FILES

name.p	input file
name.i	include file(s)
name.h	include file(s)
pmon.out	profile data
core	profile data source for **–c** option
/usr/lib/how_pxp	information on basic usage

SEE ALSO

The *Pascal User's Manual* in the Sun *Pascal* Manual.
pc(1), pi(1), px(1)

DIAGNOSTICS

For a basic explanation do

```
    tutorial% pxp
```

Error diagnostics include 'No profile data in file' with the **c** option if the **z** option was not enabled to *pi;* 'Not a Pascal system core file' if the core is not from a *px* execution; 'Program and count data do not correspond' if the program was changed after compilation, before profiling; or if the wrong program is specified.

BUGS

Does not place multiple statements per line.

Procedures and functions as parameters are printed without nested parameter lists, as in the obsolete Jensen and Wirth syntax.

NAME

pxref – Pascal cross-reference program

SYNOPSIS

pxref [–] name

DESCRIPTION

Pxref makes a line numbered listing and a cross-reference of identifier usage for the program in *name*. The optional '–' argument suppresses the listing. The keywords **goto** and **label** are treated as identifiers for the purpose of the cross-reference. **Include** directives are not processed, but cause the placement of an entry indexed by '#include' in the cross-reference.

SEE ALSO

The *Pascal User's Manual* in the Sun *Fortran and Pascal* Manual.

BUGS

Identifiers are trimmed to 10 characters.

READER COMMENT SHEET

Dear Customer,
We who work here at Sun Microsystems wish to provide the best possible documentation for our products. To this end, we solicit your comments on this manual. We would appreciate your telling us about errors in the content of the manual, and about any material which you feel should be there but isn't.

Typographical Errors:
Please list typographical errors by page number and actual text of the error.

Technical Errors:
Please list errors of fact by page number and actual text of the error.

Content:
Please list errors of fact by page number and actual text of the error.

Content:

Did this guide meet your needs? If not, please indicate what you think should be added or deleted in order to do so. Please comment on any material which you feel should be present but is not. Is there material which is in other manuals, but would be more convenient if it were in this manual?

Layout and Style:

Did you find the organization of this guide useful? If not, how would you rearrange things? Do you find the style of this manual pleasing or irritating? What would you like to see different?