Berkeley Pascal User's Manual
Version 3.1 – April 1986

William N. Joy‡, Susan L. Graham, Charles B. Haley‡,
Marshall Kirk McKusick, and Peter B. Kessler‡

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, California 94720

ABSTRACT

Berkeley Pascal is designed for interactive instructional use and runs on the PDP/11 and VAX/11 computers. Interpretive code is produced, providing fast translation at the expense of slower execution speed. There is also a fully compatible compiler for the VAX/11. An execution profiler and Wirth's cross reference program are also available with the system.

The system supports full Pascal. The language accepted is 'standard' Pascal, and a small number of extensions. There is an option to suppress the extensions. The extensions include a separate compilation facility and the ability to link to object modules produced from other source languages.

The *User's Manual* gives a list of sources relating to the UNIX† system, the Pascal language, and the Berkeley Pascal system. Basic usage examples are provided for the Pascal components *pi*, *px*, *pix*, *pc*, and *pxp*. Errors commonly encountered in these programs are discussed. Details are given of special considerations due to the interactive implementation. A number of examples are provided including many dealing with input/output. An appendix supplements Wirth's *Pascal Report* to form the full definition of the Berkeley implementation of the language.

Introduction

The Berkeley Pascal *User's Manual* consists of five major sections and an appendix. In section 1 we give sources of information about UNIX, about the programming language Pascal, and about the Berkeley implementation of the language. Section 2 introduces the Berkeley implementation and provides a number of tutorial examples. Section 3 discusses the error diagnostics produced by the translators *pc* and *pi*, and the runtime interpreter *px*. Section 4 describes input/output with special attention given to features of the interactive implementation and to features unique to UNIX. Section 5 gives details on the components of the system and explanation of all relevant options. The *User's Manual* concludes with an appendix to Wirth's *Pascal Report* with which it forms a precise definition of the implementation.

---

‡Author's current addresses: William Joy: Sun Microsystems, 2550 Garcia Ave., Mountain View, CA 94043; Charles Haley: S & B Associates, 1110 Centennial Ave., Piscataway, NJ 08854; Peter Kessler: Xerox Research Park, Palo Alto, CA
† UNIX is a Trademark of Bell Laboratories.

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. Earlier 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.

1. Sources of information

This section lists the resources available for information about general features of UNIX, text editing, the Pascal language, and the Berkeley Pascal implementation, concluding with a list of references. The available documents include both so-called standard documents – those distributed with all UNIX system – and documents (such as this one) written at Berkeley.

1.1. Where to get documentation

Current documentation for most of the UNIX system is available ‘‘on line’’ at your terminal. Details on getting such documentation interactively are given in section 1.3.

1.2. Documentation describing UNIX

The following documents are those recommended as tutorial and reference material about the UNIX system. We give the documents with the introductory and tutorial materials first, the reference materials last.

UNIX For Beginners – Second Edition

This document is the basic tutorial for UNIX available with the standard system.

Communicating with UNIX

This is also a basic tutorial on the system and assumes no previous familiarity with computers; it was written at Berkeley.

An introduction to the C shell

This document introduces *csh,* the shell in common use at Berkeley, and provides a good deal of general description about the way in which the system functions. It provides a useful glossary of terms used in discussing the system.

UNIX Programmer’s Manual

This manual is the major source of details on the components of the UNIX system. It consists of an Introduction, a permuted index, and eight command sections. Section 1 consists of descriptions of most of the ‘‘commands’’ of UNIX. Most of the other sections have limited relevance to the user of Berkeley Pascal, being of interest mainly to system programmers.

UNIX documentation often refers the reader to sections of the manual. Such a reference consists of a command name and a section number or name. An example of such a reference would be: *ed* (1). Here *ed* is a command name – the standard UNIX text editor, and ‘(1)’ indicates that its documentation is in section 1 of the manual.

The pieces of the Berkeley Pascal system are *pi* (1), *px* (1), the combined Pascal translator and interpretive executor *pix* (1), the Pascal compiler *pc* (1), the Pascal execution profiler *pxp* (1), and the Pascal cross-reference generator *pxref* (1).

It is possible to obtain a copy of a manual section by using the *man* (1) command. To get the Pascal documentation just described one could issue the command:

% **man pi**

to the shell. The user input here is shown in **bold face**; the '% ', which was printed by the shell as a prompt, is not. Similarly the command:

% **man man**

asks the *man* command to describe itself.

1.3. Text editing documents

The following documents introduce the various UNIX text editors. Most Berkeley users use a version of the text editor *ex;* either *edit,* which is a version of *ex* for new and casual users, *ex* itself, or *vi* (visual) which focuses on the display editing portion of *ex.*

A Tutorial Introduction to the UNIX Text Editor

This document, written by Brian Kernighan of Bell Laboratories, is a tutorial for the standard UNIX text editor *ed.* It introduces you to the basics of text editing, and provides enough information to meet day-to-day editing needs, for *ed* users.

Edit: A tutorial

This introduces the use of *edit,* an editor similar to *ed* which provides a more hospitable environment for beginning users.

Ex/edit Command Summary

This summarizes the features of the editors *ex* and *edit* in a concise form. If you have used a line oriented editor before this summary alone may be enough to get you started.

Ex Reference Manual – Version 3.7

A complete reference on the features of *ex* and *edit.*

An Introduction to Display Editing with Vi

Vi is a display oriented text editor. It can be used on most any CRT terminal, and uses the screen as a window into the file you are editing. Changes you make to the file are reflected in what you see. This manual serves both as an introduction to editing with *vi* and a reference manual.

Vi Quick Reference

This reference card is a handy quick guide to *vi;* you should get one when you get the introduction to *vi.*

1.4. Pascal documents – The language

This section describes the documents on the Pascal language which are likely to be most useful to the Berkeley Pascal user. Complete references for these documents are given in section 1.7.

Pascal User Manual

By Kathleen Jensen and Niklaus Wirth, the *User Manual* provides a tutorial introduction to the features of the language Pascal, and serves as an excellent quick-reference to the language. The reader with no familiarity with Algol-like languages may prefer one of the Pascal text books listed below, as they provide more examples and explanation. Particularly important here are pages 116-118 which define the syntax of the language. Sections 13 and 14 and Appendix F pertain only to the 6000-3.4 implementation of Pascal.

Pascal Report

By Niklaus Wirth, this document is bound with the *User Manual.* It is the guiding reference for implementors and the fundamental definition of the language. Some programmers find this report too concise to be of practical use, preferring the *User Manual* as a reference.

Books on Pascal

Several good books which teach Pascal or use it as a medium are available. The books by Wirth *Systematic Programming* and *Algorithms + Data Structures = Programs* use Pascal as a vehicle for teaching programming and data structure concepts respectively. They are both recommended. Other books on Pascal are listed in the references below.

1.5. Pascal documents – The Berkeley Implementation

This section describes the documentation which is available describing the Berkeley implementation of Pascal.

User's Manual

The document you are reading is the *User's Manual* for Berkeley Pascal. We often refer the reader to the Jensen-Wirth *User Manual* mentioned above, a different document with a similar name.

Manual sections

The sections relating to Pascal in the *UNIX Programmer's Manual* are *pix* (1), *pi* (1), *pc* (1), *px* (1), *pxp* (1), and *pxref* (1). These sections give a description of each program, summarize the available options, indicate files used by the program, give basic information on the diagnostics produced and include a list of known bugs.

Implementation notes

For those interested in the internal organization of the Berkeley Pascal system there are a series of *Implementation Notes* describing these details. The *Berkeley Pascal PXP Implementation Notes* describe the Pascal interpreter *px* ; and the *Berkeley Pascal PX Implementation Notes* describe the structure of the execution profiler *pxp*.

1.6. References

UNIX Documents

Communicating With UNIX
Computer Center
University of California, Berkeley
January, 1978.

Ricki Blau and James Joyce
Edit: a tutorial
UNIX User's Supplementary Documents (USD), 14
University of California, Berkeley, CA. 94720
April, 1986.

Ex/edit Command Summary
Computer Center
University of California, Berkeley
August, 1978.

William Joy

Ex Reference Manual – Version 3.7
UNIX User's Supplementary Documents (USD), 16
University of California, Berkeley, CA. 94720
April, 1986.

William Joy
An Introduction to Display Editing with Vi
UNIX User's Supplementary Documents (USD), 15
University of California, Berkeley, CA. 94720
April, 1986.

William Joy
An Introduction to the C shell (Revised)
UNIX User's Supplementary Documents (USD), 4
University of California, Berkeley, CA. 94720
April, 1986.

Brian W. Kernighan
UNIX for Beginners – Second Edition
UNIX User's Supplementary Documents (USD), 1
University of California, Berkeley, CA. 94720
April, 1986.

Brian W. Kernighan
A Tutorial Introduction to the UNIX Text Editor
UNIX User's Supplementary Documents (USD), 12
University of California, Berkeley, CA. 94720
April, 1986.

Dennis M. Ritchie and Ken Thompson
The UNIX Time Sharing System
Reprinted from Communications of the ACM July 1974 in
UNIX Programmer's Supplementary Documents, Volume 2 (PS2), 1
University of California, Berkeley, CA. 94720
April, 1986.

Pascal Language Documents

Cooper and Clancy
Oh! Pascal!, 2nd Edition
W. W. Norton & Company, Inc.
500 Fifth Ave., NY, NY. 10110
1985, 475 pp.

Cooper
Standard Pascal User Reference Manual
W. W. Norton & Company, Inc.
500 Fifth Ave., NY, NY. 10110
1983, 176 pp.

Kathleen Jensen and Niklaus Wirth
Pascal – User Manual and Report

Springer-Verlag, New York.
1975, 167 pp.

Niklaus Wirth
Algorithms + Data structures = Programs
Prentice-Hall, New York.
1976, 366 pp.

Berkeley Pascal documents

The following documents are available from the Computer Center Library at the University of California, Berkeley.

William N. Joy
Berkeley Pascal PX Implementation Notes
Version 1.1, April 1979.
(Vax-11 Version 2.0 By Kirk McKusick, December, 1979)

William N. Joy
Berkeley Pascal PXP Implementation Notes
Version 1.1, April 1979.

2. Basic UNIX Pascal

The following sections explain the basics of using Berkeley Pascal. In examples here we use the text editor *ex* (1). Users of the text editor *ed* should have little trouble following these examples, as *ex* is similar to *ed*. We use *ex* because it allows us to make clearer examples.† The new UNIX user will find it helpful to read one of the text editor documents described in section 1.4 before continuing with this section.

2.1. A first program

To prepare a program for Berkeley Pascal we first need to have an account on UNIX and to 'login' to the system on this account. These procedures are described in the documents *Communicating with UNIX* and *UNIX for Beginners*.

Once we are logged in we need to choose a name for our program; let us call it 'first' as this is the first example. We must also choose a name for the file in which the program will be stored. The Berkeley Pascal system requires that programs reside in files which have names ending with the sequence '.p' so we will call our file 'first.p'.

A sample editing session to create this file would begin:

```
% ex first.p
"first.p" [New file]
:
```

We didn't expect the file to exist, so the error diagnostic doesn't bother us. The editor now knows the name of the file we are creating. The ':' prompt indicates that it is ready for command input. We can add the text for our program using the 'append' command as follows.

```
:append
program first(output)
begin
	writeln(´Hello, world!´)
```

† Users with CRT terminals should find the editor *vi* more pleasant to use; we do not show its use here because its display oriented nature makes it difficult to illustrate.

```
end.
.
:
```

The line containing the single '**.**' character here indicated the end of the appended text. The ':' prompt indicates that *ex* is ready for another command. As the editor operates in a temporary work space we must now store the contents of this work space in the file 'first.p' so we can use the Pascal translator and executor *pix* on it.

```
:write
"first.p" [New file] 4 lines, 59 characters
:quit
%
```

We wrote out the file from the edit buffer here with the 'write' command, and *ex* indicated the number of lines and characters written. We then quit the editor, and now have a prompt from the shell.‡

We are ready to try to translate and execute our program.

```
% pix first.p
Mon Apr 14 19:00 1986  first.p:
    2  begin
e   ---↑--- Inserted ´;´
Execution begins...
Hello, world!
Execution terminated.

1 statements executed in 0.02 seconds cpu time.
%
```

The translator first printed a syntax error diagnostic. The number 2 here indicates that the rest of the line is an image of the second line of our program. The translator is saying that it expected to find a ';' before the keyword **begin** on this line. If we look at the Pascal syntax charts in the Jensen-Wirth *User Manual*, or at some of the sample programs therein, we will see that we have omitted the terminating ';' of the **program** statement on the first line of our program.

One other thing to notice about the error diagnostic is the letter 'e' at the beginning. It stands for 'error', indicating that our input was not legal Pascal. The fact that it is an 'e' rather than an 'E' indicates that the translator managed to recover from this error well enough that generation of code and execution could take place. Execution is possible whenever no fatal 'E' errors occur during translation. The other classes of diagnostics are 'w' warnings, which do not necessarily indicate errors in the program, but point out inconsistencies which are likely to be due to program bugs, and 's' standard-Pascal violations.†

After completing the translation of the program to interpretive code, the Pascal system indicates that execution of the translated program began. The output from the execution of the program then appeared. At program termination, the Pascal runtime system indicated the number of statements executed, and the amount of cpu time used, with the resolution of the latter being 1/60'th of a second.

Let us now fix the error in the program and translate it to a permanent object code file *obj* using *pi*. The program *pi* translates Pascal programs but stores the object code instead of executing it‡.

---

‡ Our examples here assume you are using *csh*.

†The standard Pascal warnings occur only when the associated **s** translator option is enabled. The **s** option is discussed in sections 5.1 and A.6 below. Warning diagnostics are discussed at the end of section 3.2, the associated **w** option is described in section 5.2.

‡This script indicates some other useful approaches to debugging Pascal programs. As in *ed* we can shorten commands in *ex* to an initial prefix of the command name as we did with the *substitute* command here. We have also used the '!' shell escape command here to execute other commands with a shell without leaving the editor.

```
% ex first.p
"first.p" 4 lines, 59 characters
:1 print
program first(output)
:s/$/;
program first(output);
:write
"first.p" 4 lines, 60 characters
:quit
% pi first.p
%
```

If we now use the UNIX *ls* list files command we can see what files we have:

```
% ls
first.p
obj
%
```

The file ‘obj’ here contains the Pascal interpreter code. We can execute this by typing:

```
% px obj
Hello, world!

1 statements executed in 0.01 seconds cpu time.
%
```

Alternatively, the command:

```
% obj
```

will have the same effect. Some examples of different ways to execute the program follow.

```
% px
Hello, world!

1 statements executed in 0.01 seconds cpu time.
% pi −p first.p
% px obj
Hello, world!
% pix −p first.p
Hello, world!
%
```

Note that *px* will assume that ‘obj’ is the file we wish to execute if we don’t tell it otherwise. The last two translations use the **−p** no-post-mortem option to eliminate execution statistics and ‘Execution begins’ and ‘Execution terminated’ messages. See section 5.2 for more details. If we now look at the files in our directory we will see:

```
% ls
first.p
obj
%
```

We can give our object program a name other than ‘obj’ by using the move command *mv* (1). Thus to name our program ‘hello’:

```
% mv obj hello
% hello
Hello, world!
% ls
first.p
hello
%
```

Finally we can get rid of the Pascal object code by using the *rm* (1) remove file command, e.g.:

```
% rm hello
% ls
first.p
%
```

For small programs which are being developed *pix* tends to be more convenient to use than *pi* and *px*. Except for absence of the *obj* file after a *pix* run, a *pix* command is equivalent to a *pi* command followed by a *px* command. For larger programs, where a number of runs testing different parts of the program are to be made, *pi* is useful as this *obj* file can be executed any desired number of times.

2.2. A larger program

Suppose that we have used the editor to put a larger program in the file ‘bigger.p’. We can list this program with line numbers by using the program *cat*-n i.e.:

```
% cat –n bigger.p
     1  (*
     2   * Graphic representation of a function
     3   *    f(x) = exp(–x) * sin(2 * pi * x)
     4   *)
     5  program graph1(output);
     6  const
     7          d = 0.0625;   (* 1/16, 16 lines for interval [x, x+1] *)
     8          s = 32;       (* 32 character width for interval [x, x+1]
     9          h = 34;       (* Character position of x–axis *)
    10          c = 6.28138;  (* 2 * pi *)
    11          lim = 32;
    12  var
    13          x, y: real;
    14          i, n: integer;
    15  begin
    16          for i := 0 to lim begin
    17                  x := d / i;
    18                  y := exp(–x9 * sin(i * x);
    19                  n := Round(s * y) + h;
    20                  repeat
    21                          write(´ ´);
    22                          n := n – 1
    23                  writeln(´*´)
    24  end.
%
```

This program is similar to program 4.9 on page 30 of the Jensen-Wirth *User Manual*. A number of problems have been introduced into this example for pedagogical reasons.

If we attempt to translate and execute the program using *pix* we get the following response:

```
% pix bigger.p
Mon Apr 14 19:00 1986  bigger.p:
    9         h = 34;      (∗ Character position of x−axis ∗)
w -----------------------↑ ---- (∗ in a (∗ ... ∗) comment
   16         for i := 0 to lim begin
e  ----------------------------↑ ---- Inserted keyword do
   18                 y := exp(−x9 ∗ sin(i ∗ x);
E  ---------------------------↑ ---- Undefined variable
e  -------------------------------------------↑ ---- Inserted ´)´
   19                 n := Round(s ∗ y) + h;
E  ---------------------↑ ---- Undefined function
E  ----------------------------------------↑ ---- Undefined variable
   23                 writeln(´∗´)
e  ----------------↑ ---- Inserted ´;´
   24  end.
E  ----↑ ---- Expected keyword until
E  ---------↑ ---- Malformed declaration
E  ---------↑ ---- Unexpected end−of−file − QUIT
Execution suppressed due to compilation errors
%
```

Since there were fatal ‘E’ errors in our program, no code was generated and execution was necessarily suppressed. One thing which would be useful at this point is a listing of the program with the error messages. We can get this by using the command:

% **pi −l bigger.p**

There is no point in using *pix* here, since we know there are fatal errors in the program. This command will produce the output at our terminal. If we are at a terminal which does not produce a hard copy we may wish to print this listing off-line on a line printer. We can do this with the command:

% **pi −l bigger.p | lpr**

In the next few sections we will illustrate various aspects of the Berkeley Pascal system by correcting this program.

2.3. Correcting the first errors

Most of the errors which occurred in this program were *syntactic* errors, those in the format and structure of the program rather than its content. Syntax errors are flagged by printing the offending line, and then a line which flags the location at which an error was detected. The flag line also gives an explanation stating either a possible cause of the error, a simple action which can be taken to recover from the error so as to be able to continue the analysis, a symbol which was expected at the point of error, or an indication that the input was ‘malformed’. In the last case, the recovery may skip ahead in the input to a point where analysis of the program can continue.

In this example, the first error diagnostic indicates that the translator detected a comment within a comment. While this is not considered an error in ‘standard’ Pascal, it usually corresponds to an error in the program which is being translated. In this case, we have accidentally omitted the trailing ‘∗)’ of the comment on line 8. We can begin an editor session to correct this problem by doing:

```
% ex bigger.p
"bigger.p" 24 lines, 512 characters
:8s/$/ ∗)
        s = 32;       (∗ 32 character width for interval [x, x+1] ∗)
:
```

The second diagnostic, given after line 16, indicates that the keyword **do** was expected before the keyword **begin** in the **for** statement. If we examine the *statement* syntax chart on page 118 of the Jensen-Wirth *User Manual* we will discover that **do** is a necessary part of the **for** statement. Similarly, we could have referred to section C.3 of the Jensen-Wirth *User Manual* to learn about the **for** statement and gotten the same information there. It is often useful to refer to these syntax charts and to the relevant sections of this book.

We can correct this problem by first scanning for the keyword **for** in the file and then substituting the keyword **do** to appear in front of the keyword **begin** there. Thus:

```
:/for
        for i := 0 to lim begin
:s/begin/do &
        for i := 0 to lim do begin
:
```

The next error in the program is easy to pinpoint. On line 18, we didn't hit the shift key and got a '9' instead of a ')'. The translator diagnosed that 'x9' was an undefined variable and, later, that a ')' was missing in the statement. It should be stressed that *pi* is not suggesting that you should insert a ')' before the ';'. It is only indicating that making this change will help it to be able to continue analyzing the program so as to be able to diagnose further errors. You must then determine the true cause of the error and make the appropriate correction to the source text.

This error also illustrates the fact that one error in the input may lead to multiple error diagnostics. *Pi* attempts to give only one diagnostic for each error, but single errors in the input sometimes appear to be more than one error. It is also the case that *pi* may not detect an error when it occurs, but may detect it later in the input. This would have happened in this example if we had typed 'x' instead of 'x9'.

The translator next detected, on line 19, that the function *Round* and the variable *h* were undefined. It does not know about *Round* because Berkeley Pascal normally distinguishes between upper and lower case.† On UNIX lower-case is preferred‡, and all keywords and built-in **procedure** and **function** names are composed of lower-case letters, just as they are in the Jensen-Wirth *Pascal Report*. Thus we need to use the function *round* here. As far as *h* is concerned, we can see why it is undefined if we look back to line 9 and note that its definition was lost in the non-terminated comment. This diagnostic need not, therefore, concern us.

The next error which occurred in the program caused the translator to insert a ';' before the statement calling *writeln* on line 23. If we examine the program around the point of error we will see that the actual error is that the keyword **until** and an associated expression have been omitted here. Note that the diagnostic from the translator does not indicate the actual error, and is somewhat misleading. The translator made the correction which seemed to be most plausible. As the omission of a ';' character is a common mistake, the translator chose to indicate this as a possible fix here. It later detected that the keyword **until** was missing, but not until it saw the keyword **end** on line 24. The combination of these diagnostics indicate to us the true problem.

The final syntactic error message indicates that the translator needed an **end** keyword to match the **begin** at line 15. Since the **end** at line 24 is supposed to match this **begin**, we can infer that another **begin** must have been mismatched, and have matched this **end**. Thus we see that we need an **end** to match the **begin** at line 16, and to appear before the final **end**. We can make these corrections:

```
:/x9/s//x)
             y := exp(−x) ∗ sin(i ∗ x);
:+s/Round/round
             n := round(s ∗ y) + h;
:/write
```

†In ``standard´´ Pascal no distinction is made based on case.

‡One good reason for using lower–case is that it is easier to type.

```
                write(´ ´);
:/
        writeln(´*´)
:insert
        until n = 0;
.
:$
end.
:insert
    end
.
:
```

At the end of each **procedure** or **function** and the end of the **program** the translator summarizes references to undefined variables and improper usages of variables. It also gives warnings about potential errors. In our program, the summary errors do not indicate any further problems but the warning that *c* is unused is somewhat suspicious. Examining the program we see that the constant was intended to be used in the expression which is an argument to *sin*, so we can correct this expression, and translate the program. We have now made a correction for each diagnosed error in our program.

```
:?i ?s//c /
          y := exp(−x) * sin(c * x);
:write
"bigger.p" 26 lines, 538 characters
:quit
% pi bigger.p
%
```

It should be noted that the translator suppresses warning diagnostics for a particular **procedure**, **function** or the main **program** when it finds severe syntax errors in that part of the source text. This is to prevent possibly confusing and incorrect warning diagnostics from being produced. Thus these warning diagnostics may not appear in a program with bad syntax errors until these errors are corrected.

We are now ready to execute our program for the first time. We will do so in the next section after giving a listing of the corrected program for reference purposes.

```
% cat −n bigger.p
     1  (∗
     2   ∗ Graphic representation of a function
     3   ∗    f(x) = exp(−x) ∗ sin(2 ∗ pi ∗ x)
     4   ∗)
     5  program graph1(output);
     6  const
     7          d = 0.0625;   (∗ 1/16, 16 lines for interval [x, x+1] ∗)
     8          s = 32;       (∗ 32 character width for interval [x, x+1] ∗)
     9          h = 34;       (∗ Character position of x−axis ∗)
    10          c = 6.28138;  (∗ 2 ∗ pi ∗)
    11          lim = 32;
    12  var
    13          x, y: real;
    14          i, n: integer;
    15  begin
    16          for i := 0 to lim do begin
    17                  x := d / i;
    18                  y := exp(−x) ∗ sin(c ∗ x);
    19                  n := round(s ∗ y) + h;
    20                  repeat
    21                          write(´ ´);
    22                          n := n − 1
    23                  until n = 0;
    24                  writeln(´∗´)
    25          end
    26  end.
%
```

2.4. Executing the second example

We are now ready to execute the second example. The following output was produced by our first run.

```
% px
Execution begins...

Real division by zero

    Error in "graph1"+2 near line 17.
Execution terminated abnormally.

2 statements executed in 0.01 seconds cpu time.
%
```

Here the interpreter is presenting us with a runtime error diagnostic. It detected a ‘division by zero’ at line 17. Examining line 17, we see that we have written the statement ‘x := d / i’ instead of ‘x := d ∗ i’. We can correct this and rerun the program:

```
% ex bigger.p
"bigger.p" 26 lines, 538 characters
:17
```

```
        x := d / i
:s´/´*
        x := d * i
:write
"bigger.p" 26 lines, 538 characters
:q
% pix bigger.p
Execution begins...
                                  *
                                              *
                                                      *
                                                           *
                                                           *
                                                        *
                                                  *
                                          *
                                  *
                            *
                      *
                    *
                    *
                      *
                          *
                              *
                                  *
                                      *
                                         *
                                           *
                                           *
                                          *
                                        *
                                     *
                                  *
                               *
                              *
                             *
                            *
                             *
                               *
                                *
                                  *
Execution terminated.

2550 statements executed in 0.67 seconds cpu time.
%
```

This appears to be the output we wanted. We could now save the output in a file if we wished by using the shell to redirect the output:

```
% px > graph
```

We can use *cat* (1) to see the contents of the file graph. We can also make a listing of the graph on the line printer without putting it into a file, e.g.

```
% px | lpr
```

```
Execution begins...
Execution terminated.

2550 statements executed in 0.68 seconds cpu time.
%
```

Note here that the statistics lines came out on our terminal. The statistics line comes out on the diagnostic output (unit 2.) There are two ways to get rid of the statistics line. We can redirect the statistics message to the printer using the syntax '| &' to the shell rather than '|', i.e.:

% **px | & lpr**
%

or we can translate the program with the **p** option disabled on the command line as we did above. This will disable all post-mortem dumping including the statistics line, thus:

% **pi −p bigger.p**
% **px | lpr**
%

This option also disables the statement limit which normally guards against infinite looping. You should not use it until your program is debugged. Also if **p** is specified and an error occurs, you will not get run time diagnostic information to help you determine what the problem is.

2.5. 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 (one with no characters on it) corresponds to a 'space' macro in an assembler, leaving a completely blank line without a line number. A line containing only a control-l (form-feed) character will cause a page eject in the listing with the corresponding line number suppressed. This corresponds to an 'eject' pseudo-instruction. See also section 5.2 for details on the **n** and **i** options of *pi*.

2.6. 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 which was abnormally terminated due to excessive looping or recursion or by a program fault, the counts can facilitate location of the error. Zero counts mark portions of the program which were not executed; during the early debugging stages they should prompt new test data or a re-examination 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.

An example

A prime number is a number which is divisible only by itself and the number one. The program *primes*, written by Niklaus Wirth, determines the first few prime numbers. In translating the program we have specified the **z** option to *pix*. This option causes the translator to generate counters and count instructions sufficient in number to determine the number of times each statement in the program was executed.† When execution of the program completes, either normally or abnormally, this count data is written to the file *pmon.out* in the current directory.‡ It is then possible to prepare an execution profile by giving *pxp* the name of the file associated with this data, as was done in the following example.

---

†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.

‡*Pmon.out* has a name similar to *mon.out* the monitor file produced by the profiling facility of the C compiler *cc* (1). See *prof* (1) for a discussion of the C compiler profiling facilities.

```
% pix –l –z primes.p
Berkeley Pascal PI –– Version 3.1 (9/7/85)

Mon Apr 14 19:00 1986  primes.p

program primes(output);
const n = 50; n1 = 7; (*n1 = sqrt(n)*)
var i,k,x,inc,lim,square,l: integer;
      prim: boolean;
      p,v: array[1..n1] of integer;
begin
     write(2:6, 3:6); l := 2;
     x := 1; inc := 4; lim := 1; square := 9;
     for i := 3 to n do
     begin (*find next prime*)
          repeat x := x + inc; inc := 6–inc;
               if square <= x then
                    begin lim := lim+1;
                         v[lim] := square; square := sqr(p[lim+1])
                    end ;
               k := 2; prim := true;
               while prim and (k<lim) do
               begin k := k+1;
                    if v[k] < x then v[k] := v[k] + 2*p[k];
                    prim := x <> v[k]
               end
          until prim;
          if i <= n1 then p[i] := x;
          write(x:6); l := l+1;
          if l = 10 then
               begin writeln; l := 0
               end
     end ;
     writeln;
end .
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   173
   179   181   191   193   197   199   211   223   227   229

Execution terminated.

1404  statements executed in 0.37 seconds cpu time.
%
```

Discussion

The header lines of the outputs of *pix* and *pxp* in this example indicate the version of the translator and execution profiler in use at the time this example 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 at which the profile data was gathered.

```
% pxp –z primes.p
```

```
Berkeley Pascal PXP -- Version 2.13 (4/2/84)

Mon Apr 14 19:00 1986  primes.p

Profiled Sun Sep 27 17:22 1987

     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.
%
```

To determine the number of times a statement was executed, one looks to the left of the statement and finds the corresponding vertical bar '|'. If this vertical bar is labelled with a count then that count gives the number of times the statement was executed. If the bar is not labelled, we look up in the listing to find the first ‘|’ which directly above the original one which has a count and that is 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**.

More information on *pxp* can be found in its manual section *pxp* (1) and in sections 5.4, 5.5 and 5.10.

3. Error diagnostics

This section of the *User's Manual* discusses the error diagnostics of the programs *pi*, *pc* and *px*. *Pix* is a simple but useful program which invokes *pi* and *px* to do all the real processing. See its manual section *pix* (1) and section 5.2 below for more details. All the diagnostics given by *pi* will also be given by *pc*.

3.1. Translator syntax errors

A few comments on the general nature of the syntax errors usually made by Pascal programmers and the recovery mechanisms of the current translator may help in using the system.

Illegal characters

Characters such as ‘$’, ‘!’, and ‘@’ are not part of the language Pascal. If they are found in the source program, and are not part of a constant string, a constant character, or a comment, they are considered to be ‘illegal characters’. This can happen if you leave off an opening string quote ‘´’. Note that the character ‘"’, although used in English to quote strings, is not used to quote strings in Pascal. Most non-printing characters in your input are also illegal except in character constants and character strings. Except for the tab and form feed characters, which are used to ease formatting of the program, non-printing characters in the input file print as the character ‘?’ so that they will show in your listing.

String errors

There is no character string of length 0 in Pascal. Consequently the input ‘´´’ is not acceptable. Similarly, encountering an end-of-line after an opening string quote ‘´’ without encountering the matching closing quote yields the diagnostic ‘‘Unmatched ´ for string’’. It is permissible to use the character ‘#’ instead of ‘´’ to delimit character and constant strings for portability reasons. For this reason, a spuriously placed ‘#’ sometimes causes the diagnostic about unbalanced quotes. Similarly, a ‘#’ in column one is used when preparing programs which are to be kept in multiple files. See section 5.11 for details.

Comments in a comment, non-terminated comments

As we saw above, these errors are usually caused by leaving off a comment delimiter. You can convert parts of your program to comments without generating this diagnostic since there are two different kinds of comments – those delimited by ‘{’ and ‘}’, and those delimited by ‘(∗’ and ‘∗)’. Thus consider:

```
{ 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 will also allow the translator to help by detecting statements accidentally placed within comments.

If a comment does not terminate before the end of the input file, the translator will point to the

†If you wish to transport your program, especially to the 6000-3.4 implementation, you should use the character sequence ‘(∗’ to delimit comments. For transportation over the *rcslink* to Pascal 6000-3.4, the character ‘#’ should be used to delimit characters and constant strings.

beginning of the comment, indicating that the comment is not terminated. In this case processing will terminate immediately. See the discussion of ''QUIT'' below.

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:

```
Mon Apr 14 19:00 1986  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 same constructs are also illegal as input to the Pascal interpreter *px*.

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 considers a set of simple corrections to see whether they will allow the analysis to continue. These corrections involve replacing an input token with a different token, inserting a token, or replacing an input token with a different token. Most of these changes will not cause fatal syntax errors. The exception is the insertion of or replacement with a symbol such as an identifier or a number; in this case the recovery makes no attempt to determine *which* identifier or *what* number should be inserted, hence these are considered fatal syntax errors.

Consider the following example.

```
% pix −l synerr.p
Berkeley Pascal PI -- Version 3.1 (9/7/85)

Mon Apr 14 19:00 1986  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.
%
```

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 which it does not, the translator diagnostic may not indicate this, as the translator is unlikely to recognize the construct you supply.

Undefined or improper identifiers

If an identifier is encountered in the input but is undefined, the error recovery will replace it with an identifier of the appropriate class. Further references to this identifier will be 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, e.g. 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 will be produced and an identifier of the appropriate type inserted. Further incorrect references to this identifier will be flagged only if they involve incorrect use in a different way, with all incorrect uses being summarized in the same way as undefined variable uses are.

Expected symbols, malformed constructs

If none of the above mentioned corrections appear reasonable, the error recovery will examine the input to the left of the point of error to see if there is only one symbol which can follow this input. If this is the case, the recovery will print 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 that the input is ‘‘malformed’’. If necessary, the translator may then skip forward in the input to a place where analysis can continue. This process may cause some errors in the text to be missed.

Consider the following example:

```
% pix –l synerr2.p
Berkeley Pascal PI –– Version 3.1 (9/7/85)

Mon Apr 14 19:00 1986  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 lines 6
  E − b undefined on line 4
  E − c undefined on line 5 6
Execution suppressed due to compilation errors
%
```

Here we misspelled *output* and gave a FORTRAN style variable declaration which the translator diagnosed as a ‘Malformed declaration’. When, on line 6, we used ‘(’ and ‘)’ for subscripting (as in FORTRAN) rather than the ‘[’ and ‘]’ which are used in Pascal, the translator noted that *a* was not defined as a **procedure**. This occurred because **procedure** and **function** argument lists are delimited by parentheses in Pascal. As it is not permissible to assign to procedure calls the translator diagnosed a malformed statement at the point of assignment.

Expected and unexpected end-of-file, ‘‘QUIT’’

If the translator finds a complete program, but there is more non-comment text in the input file, then it will indicate that an end-of-file was expected. This situation may occur after a bracketing error, or if too many **end**s are present in the input. The message may appear after the recovery says that it ‘‘Expected `.´ ’’ since ‘.’ is the symbol that terminates a program.

If severe errors in the input prohibit further processing the translator may produce a diagnostic followed by ‘‘QUIT’’. One example of this was given above – a non-terminated comment; another example is a line which is longer than 160 characters. Consider also the following example.

```
% pix –l mism.p
Berkeley Pascal PI –– Version 3.1 (9/7/85)

Mon Apr 14 19:00 1986  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
E  ------------------↑ ---- Unexpected end–of–file – QUIT
%
```

3.2. Translator semantic errors

The extremely large number of semantic diagnostic messages which the translator produces make it unreasonable to discuss each message or group of messages in detail. The messages are, however, very informative. We will here explain the typical formats and the terminology used in the error messages so that you will be able to make sense out of them. In any case in which a diagnostic is not completely comprehensible you can refer to the *User Manual* by Jensen and Wirth for examples.

Format of the error diagnostics

As we saw in the example program above, the error diagnostics from the Pascal translator include the number of a line 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 is occasionally the number of a line containing a bracketing keyword like **end** or **until**. In this case, 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 to some users. This is the only notable exception, however, to reasonable associations.

Incompatible types

Since Pascal is a strongly typed language, many semantic errors manifest themselves as type errors. These are called ‘type clashes’ by the translator. The types allowed for various operators in the language are summarized on page 108 of the Jensen-Wirth *User Manual*. It is important to know that the Pascal translator, in its diagnostics, distinguishes between the following type ‘classes’:

array	Boolean	char	file	integer
pointer	real	record	scalar	string

These words are plugged into a great number of error messages. Thus, if you tried to assign an *integer* value to a *char* variable you would receive a diagnostic like the following:

```
Mon Apr 14 19:00 1986  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 will be given each time.

Scalar

The only class whose meaning is not self-explanatory is ‘scalar’. Scalar has a precise meaning in the Jensen-Wirth *User Manual* 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 would be correct, in the context of the *User Manual* to refer to an integer variable as a *scalar* variable *pi* prefers the more specific identification.

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 similar to:

```
Mon Apr 14 19:00 1986  sin1.p:
E 12 – sin takes exactly one argument
```

is given. If the type of the argument is wrong, a message like

```
Mon Apr 14 19:00 1986  sin2.p:
E 12 – sin´s argument must be integer or real, not char
```

is produced. A few functions and procedures implemented in Pascal 6000-3.4 are diagnosed as unimplemented in Berkeley Pascal, notably those related to **segmented** files.

Can’t read and write scalars, etc.

The messages which state that scalar (user-defined) types cannot be written to and from files are often mysterious. It is in fact the case that 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 which allows enumerated types to be read and written, however if the program is to be portable, you will have to write your own routines to perform these functions. Standard Pascal only allows the reading of characters, integers and real numbers from text files. You cannot read strings or Booleans. It is possible to make a

file of color

but the representation is binary rather than string.

Expression diagnostics

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

```
% pi –l expr.p
Berkeley Pascal PI –– Version 3.1 (9/7/85)

Mon Apr 14 19:00 1986  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
%
```

This example is admittedly far-fetched, but illustrates that the error messages are sufficiently clear to allow easy determination of the problem in the expressions.

Type equivalence

Several diagnostics produced by the Pascal translator complain about ‘non-equivalent types’. In general, Berkeley Pascal considers variables to have the same type only if they were declared with the same constructed type or with the same 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
```

thus produces the diagnostics:

```
Mon Apr 14 19:00 1986  typequ.p:
E 7 − Type clash: non−identical pointer types
  ... Type of expression clashed with type of variable in assignment
```

Thus 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, in practice, to declare any type which will be used in this way.

Unreachable statements

Berkeley 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!´)
```

Goto's into structured statements

The translator detects and complains about **goto** statements which transfer control into structured statements (**for**, **while**, etc.) 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.

Unused variables, never set variables

Although *pi* always clears variables to 0 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:

1) Use of a variable which is never assigned a value.
2) A variable which is declared but never used, distinguishing between those variables for which values are computed but which are never used, and those completely unused.

In fact, these diagnostics are applied to all declared items. Thus a **const** or a **procedure** which is declared but never used is flagged. The **w** option of *pi* may be used to suppress these warnings; see sections 5.1 and

5.2.

3.3. Translator panics, i/o errors

Panics

One class of error which rarely occurs, but which 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 will be quickly and perhaps ungracefully terminated. You should contact a teaching assistant or a member of the system staff, after saving a copy of your program for later inspection. If you were making changes to an existing program when the problem occurred, you may be able to work around the problem by ascertaining which change caused the *snark* and making a different change or correcting an error in the program. A small number of panics are possible in *px*. All panics should be reported to a teaching assistant or systems staff so that they can be fixed.

Out of memory

The only other error which will abort translation when no errors are detected is running out of memory. All tables in the translator, with the exception of the parse stack, are dynamically allocated, and can grow to take up the full available process space of 64000 bytes on the PDP-11. On the VAX-11, table sizes are extremely generous and very large (25000) line programs have been easily accommodated. For the PDP–11, it is generally true that the size of the largest translatable program is directly related to **procedure** and **function** size. A number of non-trivial Pascal programs, including some with more than 2000 lines and 2500 statements have been translated and interpreted using Berkeley Pascal on PDP-11's. Notable among these are the Pascal-S interpreter, a large set of programs for automated generation of code generators, and a general context-free parsing program which has been used to parse sentences with a grammar for a superset of English. In general, very large programs should be translated using *pc* and the separate compilation facility.

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 may yet be able to translate your program if you break this one **procedure** or **function** into several routines.

I/O errors

Other errors which you may encounter when running *pi* relate to input-output. If *pi* cannot open the file you specify, or if the file is empty, you will be so informed.

3.4. Run-time errors

We saw, in our second example, a run-time error. We here give the general description of run-time errors. The more unusual interpreter error messages are explained briefly in the manual section for *px* (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 being inaccessible to the user.

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 loop, or running out of memory‡. The interpreter will

‡The checks for running out of memory are not foolproof and there is a chance that the interpreter will fault, producing a

produce a backtrace 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 we have accidentally declared the constant *n1* to be 6, instead of 7 on line 2 of the program primes as given in section 2.6 above. If we run this program we get the following response.

```
% 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 "primes"+8 near line 14.
Execution terminated abnormally.

941  statements executed in 0.25 seconds cpu time.
%
```

Here 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.

Interrupts

If the program is interrupted while executing and the **p** option was not specified, then a backtrace will be printed.† The file *pmon.out* of profile information will be written if the program was translated with the **z** option enabled to *pi* or *pix*.

I/O interaction errors

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

4. Input/output

This section describes features of the Pascal input/output environment, with special consideration of the features peculiar to an interactive implementation.

4.1. Introduction

Our first sample programs, in section 2, used the file *output*. We gave examples there of redirecting the output to a file and to the line printer using the shell. Similarly, we can read the input from a file or another program. Consider the following Pascal program which is similar to the program *cat* (1).

```
% pix –l kat.p <primes
Berkeley Pascal PI –– Version 3.1 (9/7/85)
```

---

core image when it runs out of memory. This situation occurs very rarely.

∗ On the VAX−11, each variable is restricted to allocate at most 65000 bytes of storage (this is a PDP−11ism that has survived to the VAX.)

†Occasionally, the Pascal system will be in an inconsistent state when this occurs, e.g. when an interrupt terminates a **procedure** or **function** entry or exit. In this case, the backtrace will only contain the current line. A reverse call order list of procedures will not be given.

```
Mon Apr 14 19:00 1986  kat.p

program kat(input, output);
var
        ch: char;
begin
        while not eof do begin
                while not eoln do begin
                        read(ch);
                        write(ch)
                end;
                readln;
                writeln
        end
end { kat }.
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   173
   179   181   191   193   197   199   211   223   227   229

Execution terminated.

925  statements executed in 0.25 seconds cpu time.
%
```

Here we have used the shell's syntax to redirect the program input from a file in *primes* in which we had placed the output of our prime number program of section 2.6. It is also possible to 'pipe' input to this program much as we piped input to the line printer daemon *lpr* (1) before. Thus, the same output as above would be produced by

% **cat primes | pix –l kat.p**

All of these examples use the shell to control the input and output from files. One very simple way to associate Pascal files with named UNIX files is to place the file name in the **program** statement. For example, suppose we have previously created the file *data.* We then use it as input to another version of a listing program.

```
% cat data
line one.
line two.
line three is the end.
% pix –l copydata.p
Berkeley Pascal PI –– Version 3.1 (9/7/85)

Mon Apr 14 19:00 1986  copydata.p

program copydata(data, output);
var
        ch: 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 { copydata }.
Execution begins...
line one.
line two.
line three is the end.
Execution terminated.

134  statements executed in 0.04 seconds cpu time.
%
```

By mentioning the file *data* in the **program** statement, we have indicated that we wish it to correspond to the UNIX file *data*. Then, when we ‘reset(data)’, the Pascal system opens our file ‘data’ for reading. More sophisticated, but less portable, examples of using UNIX files will be given in sections 4.5 and 4.6. There is a portability problem even with this simple example. Some Pascal systems attach meaning to the ordering of the file in the **program** statement file list. Berkeley Pascal does not do so.

4.2. 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. Setting *eof* or *eoln* actually corresponds to an implicit read in which the input is inspected, but no input is ‘‘used up’’. In fact, there is no way the system can know whether the input is at the end-of-file or the end-of-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 run-time action by the user. However, if the input is from a terminal, then the input is what the user types.† If the system were to do an initial read automatically at the beginning of program execution, and if the input were 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 or printing a herald.

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 or 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, e.g. 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 will be necessary for it to attempt to read from the input.

Thus 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 be appear to be a correct program for requesting, reading and echoing numbers.

†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.

Notice, however, that the **while** loop asks whether *eof* is true *before* the request is printed. This will force the Pascal system to decide whether the input is at the end-of-file. The Pascal system will give no messages; it will simply wait 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
```

The user must still type a line before the **while** test is completed, but the prompt will ask for it. This example, however, is still not correct. To understand why, it is first necessary to know, as we will discuss below, that there is a blank character at the end of each line in a Pascal text file. The *read* procedure, when reading integers or real numbers, is defined so that, if there are only blanks left in the file, it will return a zero value and set the end-of-file condition. If, however, there is a number remaining in the file, the end-of-file condition will not be set even if it is the last number, as *read* never reads the blanks after the number, and there is always at least one blank. Thus the modified code will still put out a spurious

```
that was a 0
```

at the end of a session with it when the end-of-file is reached. The simplest way to correct the problem in this example is to use the procedure *readln* instead of *read* here. In general, unless we test the end-of-file condition both before and after calls to *read* or *readln*, there will be inputs for which our program will attempt to read past end-of-file.

4.3. More about eoln

To have a good understanding of when *eoln* will be 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.† 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 new-line character, the normal UNIX line separator. When the read routine gets the new-line character, it replaces that character by a blank (causing every line to end with a blank) and sets *eoln* to true. *Eoln* will be true as soon as we read the last character of the line and **before** we read the blank character corresponding to the end of line. Thus it is almost always a mistake to write a program which deals with input in the following way:

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

as this will almost surely have 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

†In Pascal terms, ‘read(ch)’ corresponds to ‘ch := inputˆ; get(input)’

discards this blank. The next character available from *read* will therefore be the first character of the next line, if one exists.

4.4. Output buffering

A final point about Pascal input-output must be noted here. This 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 which has been generated for the file *output* whenever

1) A *writeln* occurs, or
2) The program reads from the terminal, or
3) 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 integers will not print until the *writeln* occurs. The delay can be somewhat disconcerting, and you should be aware that it will occur. By setting the **b** option to 0 before the **program** statement by inserting a comment of the form

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

we can cause *output* to be completely unbuffered, with a corresponding horrendous degradation in program efficiency. Option control in comments is discussed in section 5.

4.5. 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. As we saw in section 2.9, by mentioning the file in the **program** statement, we could 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 will generate a temporary name of the form 'tmp.x' for some character 'x', and associate this UNIX file name name with the Pascal file. The first such generated name will be 'tmp.1' and the names continue by incrementing their last character through the ASCII set. The advantage of using such temporary files is that they are automatically *remove*d 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 we can give that name in the *reset* or *rewrite* call, e.g. we could have associated the Pascal file *data* with the file 'primes' in our example in section 3.1 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 is serves as an aid to program documentation. The second parameter to *reset* and *rewrite* may 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 run time. Full details on file name/file variable associations are given in section A.3.

4.6. 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

% **obj /etc/motd /usr/dict/words hello**

will invoke the program in the file *obj* with *argc* having a value of 4. The zeroth element accessed by *argv* will be ‘obj’, the first ‘/etc/motd’, etc.

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* will 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 file names.

We are now ready to give a Berkeley Pascal program ‘kat’, based on that given in section 3.1 above, which can be used with the same syntax as the UNIX system program *cat* (1).

```
% cat 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 }.
%
```

Note that the *reset* call to the file *input* here, which is necessary for a clear program, may be disallowed 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 we can do

```
% pi kat.p
% mv obj kat
% kat primes
     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


930  statements executed in 0.29 seconds cpu time.
% 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!)

287 statements executed in 0.08 seconds cpu time.
%
```

Thus we see that, if it is given arguments, 'kat' will, like *cat,* copy each one in turn. If no arguments are given, it copies from the standard input. Thus it will work as it did before, with

```
% kat < primes
```

now equivalent to

```
% kat primes
```

although the mechanisms are quite different in the two cases. Note that if 'kat' is given a bad file name, for example:

```
% kat xxxxqqq

Could not open xxxxqqq: No such file or directory

        Error in "kat"+5 near line 11.

4 statements executed in 0.02 seconds cpu time.
%
```

it will give a diagnostic and a post-mortem control flow backtrace for debugging. If we were going to use 'kat', we might want to translate it differently, e.g.:

```
% pi −pb kat.p
```

% **mv obj kat**

Here we have disabled the post-mortem statistics printing, so as not to get the statistics or the full traceback on error. The **b** option will cause the system to block buffer the input/output so that the program will run more efficiently on large files. We could have also specified the **t** option to turn off runtime tests if that was felt to be a speed hindrance to the program. Thus we can try the last examples again:

```
% kat xxxxqqq

Could not open xxxxqqq: No such file or directory

        Error in "kat"
% kat primes
     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

%
```

The interested reader may wish to try writing a program which accepts command line arguments like *pi* does, using *argc* and *argv* to process them.

5. Details on the components of the system

5.1. Options

The programs *pi*, *pc*, and *pxp* take a number of options.† There is a standard UNIX convention for passing options to programs on the command line, and this convention is followed by the Berkeley Pascal system programs. As we saw in the examples above, option related arguments consisted 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 it inverts the default setting of that option. Thus

% **pi –l foo.p**

enables the listing option **l**, since it defaults off, while

% **pi –t foo.p**

disables the run time tests option **t**, since it defaults on.

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

{$l–}

Here we see that the opening comment delimiter (which could also be a '(*') is immediately followed by the character '$'. After this '$', which signals the start of the option list, we can place a sequence of letters and option controls, separated by ',' characters‡. The most basic actions for options are to set

†As *pix* uses *pi* to translate Pascal programs, it takes the options of *pi* also. We refer to them here, however, as *pi* options.
‡This format was chosen because it is used by Pascal 6000-3.4. In general the options common to both implementations are controlled in the same way so that comment control in options is mostly portable. It is recommended, however, that only one control be put per comment for maximum portability, as the Pascal 6000-3.4 implementation will ignore controls after the first one which it does not recognize.

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.

5.2. Options common to Pi, Pc, and Pix

The following options are common to both the compiler and the interpreter. With each option we give its default setting, the setting it would have if it appeared on the command line, and a sample command using the option. Most options are on/off valued, with the **b** option taking a single digit value.

Buffering of the file output – b

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 section 5 below. Mentioning **b** on the command line, e.g.

% **pi –b assembler.p**

causes standard output to be block buffered, where a block is some system–defined number of characters. The **b** option may also be controlled in comments. It, unique among the Berkeley Pascal options, takes a single digit value rather than an on or off setting. A value of 0, e.g.

{$b0}

causes the file *output* to be unbuffered. Any value 2 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.

Include file listing – i

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

% **pix –i scanner.i compiler.p**

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

% **pix –i scanner compiler.p**

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

Make a listing – l

The **l** option enables a listing of the program. The **l** option defaults off. When specified on the command line, it causes 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.

---

†**Include** files are discussed in section 5.9.

Standard Pascal only − s

The **s** option causes many of the features of the UNIX implementation which are not found in standard Pascal to be diagnosed as ‘s’ warning errors. This option defaults off and is enabled when mentioned on the command line. Some of the features which are diagnosed are: non-standard **procedure**s and **function**s, extensions to the **procedure** *write*, and the 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, thus

% **pi −s isitstd.p**

will produce warnings unless the program meets the standard.

Runtime tests − t and C

These options control the generation of tests that subrange variable values are within bounds at run time. *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.‡

Suppress warning diagnostics − w

The **w** option, which defaults on, 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

% **pi −w tryme.p**

suppresses these usually useful diagnostics.

Generate counters for a pxp execution profile − z

The **z** option, which defaults off, enables the production of execution profiles. By specifying **z** on the command line, i.e.

% **pi −z foo.p**

or by enabling it in a comment before the **program** statement causes *pi* and *pc* to insert operations in the interpreter code to count the number of times each statement was executed. An example of using *pxp* was given in section 2.6; its options are described in section 5.6. Note that the **z** option cannot be used on separately compiled programs.

5.3. Options available in Pi

Post-mortem dump − p

The **p** option defaults on, and causes the runtime system to initiate a post-mortem backtrace when an error occurs. It also cause *px* to 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 give 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 backtrace on error, **p** must be off at the end of the **program** statement. Thus, the Pascal cross-reference program was translated with

% **pi −pbt pxref.p**

‡See section A.1 for a description of **assert** statements.

5.4. Options available in Px

The first argument to *px* is the name of the file containing the program to be interpreted. If no arguments are 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 section 4.6.

Px may 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

% **obj primes**

will be converted to read

% **px obj primes**

5.5. Options available in Pc

Generate assembly language – S

The program is compiled and the assembly language output is left in file appended .s. Thus

% **pc –S foo.p**

creates a file *foo.s*. No executable file is created.

Symbolic Debugger Information – g

The **g** option causes the compiler to generate information needed by *sdb*(1) the symbolic debugger. For a complete description of *sdb* see Volume 2c of the UNIX Reference Manual.

Redirect the output file – o

The *name* argument after the **-o** is used as the name of the output file instead of *a.out*. Its typical use is to name the compiled program using the root of the file name. Thus:

% **pc –o myprog myprog.p**

causes the compiled program to be called *myprog*.

Generate counters for a *prof* execution profile – p

The compiler produces code which 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 used by *pxp*. This results in less degradation in execution, at somewhat of a loss in accuracy. See *prof*(1) for a more complete description.

Run the object code optimizer – O

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

5.6. Options available in Pxp

Pxp takes, on its command line, a list of options followed by the program file name, which must end in ‘.p’ as it must for *pi*, *pc*, and *pix*. *Pxp* will produce an execution profile if any of the **z**, **t** or **c** options is specified on the command line. If none of these options 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 following options are relevant to profiling with *pxp*:

Include the bodies of all routines in the profile − a

Pxp normally suppresses printing the bodies of routines which were never executed, to make the profile more compact. This option forces all routine bodies to be printed.

Suppress declaration parts from a profile − d

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

Eliminate include directives − e

Normally, *pxp* preserves **include** directives to 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, e.g. before transporting a program.

Fully parenthesize expressions − f

Normally *pxp* prints expressions with the minimal parenthesization necessary to preserve the structure of the input. This option causes *pxp* to fully parenthesize expressions. Thus the statement which prints as

    d := a + b mod c / e

with minimal parenthesization, the default, will print as

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

with the **f** option specified on the command line.

Left justify all procedures and functions − j

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.

Print a table summarizing procedure and function calls − t

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.

Enable and control the profile − z

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 **procedure**s and **function**s or **include** files which are to be profiled. If this list is null, then the whole file is to be profiled. A typical command for extracting a profile of part of a large program would be

% **pxp −z test parser.i compiler.p**

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

5.7. Formatting programs using pxp

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

% **pxp dirty.p > clean.p**

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

Pxp automatically paragraphs the program, performing housekeeping chores such as comment alignment, and treating blank lines, lines containing exactly one blank and lines containing only a form-feed character as though they were comments, preserving their vertical spacing effect in the output. *Pxp* distinguishes between four kinds of comments:

1) Left marginal comments, which begin in the first column of the input line and are placed in the first column of an output line.
2) Aligned comments, which are preceded by no input tokens on the input line. These are aligned in the output with the running program text.
3) Trailing comments, which are preceded in the input line by a token with no more than two spaces separating the token from the comment.
4) Right marginal comments, which are preceded in the input line by a token from which they are separated by at least three spaces or a tab. These are aligned down the right margin of the output, currently to the first tab stop after the 40th column from the current ‘‘left margin’’.

Consider the following program.

```
% 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.

```
% 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.
%
```

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.

Strip comments −s

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

Underline keywords − _

A command line argument of the form −_ as in

% **pxp −_ dirty.p**

can be used to cause *pxp* to underline all keywords in the output for enhanced readability.

Specify indenting unit − [23456789]

The normal unit which *pxp* uses to indent a structure statement level is 4 spaces. By giving an argument of the form −*d* with *d* a digit, $2 \le d \le 9$ you can specify that *d* spaces are to be used per level instead.

5.8. Pxref

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

% **pxref foo.p**

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

5.9. Multi-file programs

A text inclusion facility is available with 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 for ease in editing, listing, and maintenance.

The **include** facility is based on that of the UNIX C compiler. To trigger it you can place the character ‘#’ in the first portion of a line and then, after an arbitrary number of blanks or tabs, the word ‘include’ followed by a filename enclosed in single ‘´’ or double ‘"’ quotation marks. The file name 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 would be:

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

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

begin
    { main program }
end.
```

At the point the **include** pseudo-statement is encountered in the input, the lines from the included file are interpolated into the input stream. For the purposes of translation and runtime diagnostics and statement numbers in the listings and post-mortem backtraces, the lines in the included file are numbered from

1. Nested includes are possible up to 10 deep.

See the descriptions of the **i** option of *pi* in section 5.2 above; this can be used to control listing when **include** files are present.

When a non-trivial 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 will be printed before each diagnostic if the current filename has changed since the last filename was printed.

5.10. Separate Compilation with Pc

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

Normally, *pc* expects to be given entire Pascal programs. However, if given the –**c** option on the command line, it will accept a sequence of definitions and declarations, and compile them into a **.o** file, to be linked with a Pascal program at a later time. In order that procedures and functions be available across separately compiled files, they must be declared with the directive **external**. 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 at the **external** declaration and may not be specified at the resolution.

Type checking is performed across separately compiled files. Since Pascal type defintions define unique types, any types which are shared between separately compiled files must be the same definition. This seemingly impossible problem is solved using a facility similar to the **include** facility discussed above. Definitions may be placed in files with 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 **const**s and the declaration of **label**s, **var**s, and **external function**s and **procedure**s. Thus **procedure**s and **function**s which are used between separately compiled files must be declared **external**, and must be so declared in a **.h** file included by any file which calls or resolves the **function** or **procedure**. Conversely, **function**s and **procedure**s declared **external** may only be so declared in **.h** files. These files may be included only at the outermost level, and thus define or declare global objects. Note that since only **external function** and **procedure** declarations (and not resolutions) are allowed in **.h** files, statically nested **function**s and **procedure**s can not be declared **external**.

An example of the use of included **.h** files in a program would be:

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

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

begin
   { main program }
end.
```

This might include in the main program the definitions and declarations of all the global **label**s, **const**s, **type**s **var**s 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.

A. Appendix to Wirth's Pascal Report

This section is an appendix to the definition of the Pascal language in Niklaus Wirth's *Pascal Report* and, with that Report, precisely defines the Berkeley implementation. This appendix includes a summary of extensions to the language, gives the ways in which the undefined specifications were resolved, gives limitations and restrictions of the current implementation, and lists the added functions and procedures available. It concludes with a list of differences with the commonly available Pascal 6000–3.4 implementation, and some comments on standard and portable Pascal.

A.1. Extensions to the language Pascal

This section defines non-standard 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 which are to be transported.

String padding

Berkeley Pascal will pad constant strings with blanks in expressions and as value parameters to make 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 padded blanks are added on the right. Thus the assignment above is equivalent to:

```
z := ´red          ´
```

which is standard Pascal.

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.

Assert statement

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

assert <expr>

Enumerated type input-output

Enumerated types may be read and written. On output the string name associated with the enumerated value is output. If the value is out of range, a runtime 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 runtime error occurs.

Structure returning functions

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

Separate compilation

The compiler *pc* has been extended to allow separate compilation of programs. Procedures and functions declared at the global level may 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 section 5.10 for details.

A.2. Resolution of the undefined specifications

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:

1) If a global Pascal file variable appears in the **program** statement then it is associated with UNIX file of the same name.
2) If a file was reset or rewritten using the extended two-argument form of *reset* or *rewrite* then the given name is associated.
3) If a file which has never had UNIX name associated is reset or rewritten without specifying a name via the second argument, then a temporary name of the form ‘tmp.x’ is associated with the file. Temporary names start with ‘tmp.1’ and continue by incrementing the last character in the USASCII ordering. Temporary files are removed automatically when their scope is exited.

The program statement

The syntax of the **program** statement is:

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

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

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:

1) The program heading **must** contains the formal parameter *output.* If *input* is used, explicitly or implicitly, then it must also be declared here.

2) Unlike all other files, the Pascal files *input* and *output* must not be defined in a declaration, as their declaration is automatically:

   var input, output: text

3) The procedure *reset* may be used on *input.* If no UNIX file name has ever been associated with *input,* and no file name is given, then an attempt will be made to ‘rewind’ *input.* If this fails, a run time error will occur. *Rewrite* calls to output act as for any other file, except that *output* initially has no associated file. This means that a simple

   rewrite(output)

   associates a temporary name with *output.*

Details for files

If a file other than *input* is to be 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 runtime error occurs. Writing of a file other than *output* must be initiated by a *rewrite* call, which causes the Pascal system to create the associated UNIX file and to then open the file for writing only.

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 512 characters, 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 512 characters. All output buffers are flushed when the files are closed at scope exit, 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 teletype, and the Pascal system reads a character at the beginning of execution to define ‘input↑’, then no prompt could 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, reading from a file occurring only when necessary.

The character set

Seven bit USASCII is the character set used on UNIX. The standard Pascal symbols ‘and’, ’or’, ’not’, ’<=’, ’>=’, ’<>’, and the uparrow ‘↑’ (for pointer qualification) are recognized.† Less portable are the synonyms tilde ‘˜’ for **not**, ‘&’ for **and**, and ‘|’ for **or**.

Upper and lower case are considered to be distinct. 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

†On many terminals and printers, the up arrow is represented as a circumflex ‘ˆ’. These are not distinct characters, but rather different graphic representations of the same internal codes.
The proposed standard for Pascal considers them to be the same.

meaning when it is the first character on a line − see *Multi-file programs* below.

The standard types

The standard type *integer* is conceptually defined as

type integer = minint .. maxint;

Integer is implemented with 32 bit twos complement arithmetic. Predefined constants of type *integer* are:

const maxint = 2147483647; minint = −2147483648;

The standard type *char* is conceptually defined as

type char = minchar .. maxchar;

Built-in character constants are 'minchar' and 'maxchar', 'bell' and 'tab'; ord(minchar) = 0, ord(maxchar) = 127.

The type *real* is implemented using 64 bit floating point arithmetic. The floating point arithmetic is done in 'rounded' mode, and provides approximately 17 digits of precision with numbers as small as 10 to the negative 38th power and as large as 10 to the 38th power.

Comments

Comments can be delimited by either '{' and '}' or by '(*' and '*)'. If the character '{' appears in a comment delimited by '{' and '}', a warning diagnostic is printed. A similar warning will be 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.

Option control

Options of the translators may be controlled in two distinct ways. A number of 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 default setting of each given option to be changed. This method of communication of options is expected to predominate for UNIX. Thus the command

% **pi –l –s foo.p**

translates the file foo.p with the listing option enabled (as it normally is off), and with only standard Pascal features available.

If more control over the portions of the program where options are enabled is required, then option control in comments can and should be used. The format for option control in comments is identical to that used in Pascal 6000–3.4. One places the character '$' as the first character of the comment and follows it by a comma separated list of directives. Thus an equivalent to the command line example given above would be:

{$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 it is extremely unlikely in an interactive environment that one wants a listing of the program each time it is translated.

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 **b** option takes a single digit instead of a '+' or '–'.

Notes on the listings

The first page of a listing includes a banner line indicating the version and date of generation of *pi* or *pc*. It also includes the UNIX path name 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 for the editor. Currently, two special kinds of lines may be used to format the listing: a line consisting of a form-feed 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 truly blank line. These lines thus correspond to 'eject' and 'space' macros found in many assemblers. Non-printing characters are printed as the character '?' in the listing.†

The standard procedure write

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

integer	10
real	22
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 explicitly indicated by 'writeln(f)', where 'writeln(output)' may be written simply as 'writeln'. For UNIX, the built-in function 'page(f)' puts a single ASCII form-feed character on the output file. For programs which are to be transported the filter *pcc* can be used to interpret carriage control, as UNIX does not normally do so.

A.3. Restrictions and limitations

Files

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

Arrays, sets and strings

The calculations involving array subscripts and set elements are done with 16 bit arithmetic. This restricts the types over 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 32768. In particular, strings may have any length from 1 to 65535 characters, and sets may contain no more than 65535 elements.

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 single position over their entire length. There is a limit, however, on the maximum input line length. This limit is quite generous however, currently exceeding 160 characters.

Procedure and function nesting and program size

At most 20 levels of **procedure** and **function** nesting are allowed. There is no fundamental, translator defined limit on the size of the program which can be translated. The ultimate limit is supplied by the hardware and thus, on the PDP-11, by the 16 bit address space. If one runs up against the 'ran out of memory' diagnostic the program may yet translate if smaller procedures are used, as a lot of space is freed by the translator at the completion of each **procedure** or **function** in the current implementation.

---

†The character generated by a control-i indents to the next 'tab stop'. Tab stops are set every 8 columns in UNIX. Tabs thus provide a quick way of indenting in the program.

On the VAX-11, there is an implementation defined limit of 65536 bytes per variable. There is no limit on the number of variables.

Overflow

There is currently no checking for overflow on arithmetic operations at run-time on the PDP-11. Overflow checking is performed on the VAX-11 by the hardware.

A.4. Added types, operators, procedures and functions

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†. 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.

Note that if *intset* is redefined via:

type intset = **set of** 0..58;

then the default integer set is the implicit *intset* of Pascal 6000–3.4

Additional predefined operators

The relationals ‘<’ and ‘>’ of proper set inclusion are available. With *a* and *b* sets, note that

(**not** (*a* < *b*)) <> (*a* >= *b*)

As an example consider the sets *a* = [0,2] and *b* = [1]. The only relation true between these sets is ‘<>’.

Non-standard procedures

argv(i,a)	where *i* is an integer and *a* is a string variable assigns the (possibly truncated or blank padded) *i* ’th argument of the invocation of the current UNIX process to the variable *a*. The range of valid *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, i.e. ‘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 backtrace.
linelimit(f,x)‡	with *f* a textfile and *x* an integer expression causes the program to be abnormally terminated if more than *x* lines are written on file *f*. If *x* is less than 0 then no limit is imposed.
message(x,...)	causes the parameters, which have the format of those to the built-in **procedure** *write,* to be written unbuffered on the diagnostic unit 2, almost

†The current translator makes a special case of the construct ‘if ... in [...]’ and enforces only the more lax restriction on 16 bit arithmetic given above in this case.
‡Currently ignored by pdp-11 *px*.

	always 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)	where *a* is a string causes the UNIX file whose name is *a,* with trailing blanks eliminated, to be removed.
reset(f,a)	where *a* is a string causes the file whose name is *a* (with blanks trimmed) to be associated with *f* in addition to the normal function of *reset.*
rewrite(f,a)	is analogous to 'reset' above.
stlimit(i)	where *i* is an integer sets the statement limit to be *i* statements. 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.*

Non-standard functions

argc	returns the count of arguments when the Pascal program was invoked. *Argc* is always at least 1.
card(x)	returns the cardinality of the set *x,* i.e. the number of elements contained in the set.
clock	returns an integer which is the number of central processor milliseconds of user time used by this process.
expo(x)	yields the integer valued exponent of the floating-point representation of *x*; expo(*x*) = entier(log2(abs(*x*))).
random(x)	where *x* is a real parameter, evaluated but otherwise ignored, invokes a linear congruential random number generator. 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)	where *i* is an integer sets the random number generator seed to *i* and returns the previous seed. Thus seed(seed(i)) has no effect except to yield value *i.*
sysclock	an integer function of no arguments returns the number of central processor milliseconds of system time used by this process.
undefined(x)	a Boolean function. Its argument is a real number and it always returns false.
wallclock	an integer function of no arguments returns the time in seconds since 00:00:00 GMT January 1, 1970.

A.5. Remarks on standard and portable Pascal

It is occasionally desirable to prepare Pascal programs which will be acceptable at other Pascal installations. While certain system dependencies are bound to creep in, judicious design and programming practice can usually eliminate most of the non-portable usages. Wirth's *Pascal Report* concludes with a standard for implementation and program exchange.

In particular, the following differences may cause trouble when attempting to transport programs between this implementation and Pascal 6000–3.4. Using the **s** translator option may serve to indicate many problem areas.†

†The **s** option does not, however, check that identifiers differ in the first 8 characters. *Pi* and *pc* also do not check the semantics of **packed**.

Features not available in Berkeley Pascal

Segmented files and associated functions and procedures.

The function *trunc* with two arguments.

Arrays whose indices exceed the capacity of 16 bit arithmetic.

Features available in Berkeley Pascal but not in Pascal 6000-3.4

The procedures *reset* and *rewrite* with file names.

The functions *argc, seed, sysclock,* and *wallclock.*

The procedures *argv, flush,* and *remove.*

Message with arguments other than character strings.

Write with keyword **hex**.

The **assert** statement.

Reading and writing of enumerated types.

Allowing functions to return structures.

Separate compilation of programs.

Comparison of records.

Other problem areas

Sets and strings are more general in Berkeley Pascal; see the restrictions given in the Jensen-Wirth *User Manual* for details on the 6000–3.4 restrictions.

The character set differences may cause problems, especially the use of the function *chr,* characters as arguments to *ord,* and comparisons of characters, since the character set ordering differs between the two machines.

The Pascal 6000–3.4 compiler uses a less strict notion of type equivalence. In Berkeley Pascal, types are considered identical only if they are represented by the same type identifier. Thus, in particular, unnamed types are unique to the variables/fields declared with them.

Pascal 6000–3.4 doesn't recognize our option flags, so it is wise to put the control of Berkeley Pascal options to the end of option lists or, better yet, restrict the option list length to one.

For Pascal 6000–3.4 the ordering of files in the program statement has significance. It is desirable to place *input* and *output* as the first two files in the **program** statement.

Acknowledgments

The financial support of William Joy and Susan Graham by the National Science Foundation under grants MCS74-07644-A04, MCS78-07291, and MCS80-05144, and the William Joy by an IBM Graduate Fellowship are gratefully acknowledged.