P R O P A S C A L

U S E R M A N U A L

Version zz 2.1 for Z80 with CP/M

April 1983

37 Gwendolen Avenue
London SW15 6EP England

COPYRIGHT

DISCLAIMER

While every effort is made to ensure accuracy, Prospero Software cannot be held responsible for errors or omissions, and reserve the right to revise this document without notice.

TRADEMARKS

Acknowledgement is made for references in this manual to CP/M (a trademark of Digital Research of California), to CDOS (a trademark of Cromemco Inc. of California), to Macro-80 (a trademark of Microsoft of Washington State), and to Z80 (a trademark of Zilog Corporation of California).

PASCAL

Pascal is a programming language originated by Niklaus Wirth and colleagues in Zurich during the early 1970's. Since then it has achieved worldwide recognition, and been implemented on a wide variety of computers. It reflects Wirth's belief that the organisation of data is an aspect of programming as important as the definition of the processing to be carried out on that data, and indeed that the two are inseparable. Pascal provides for definition of record layouts and files, as well as arrays, and includes dynamic storage allocation facilities (the "heap") as well as the more conventional arrangements.

Two factors have probably contributed most to the popular success which Pascal has achieved. Both are essentially practical in nature.

> Efficient programs can be generated without any recourse in the language to hardware-dependent concepts. Pascal programs are in practice more portable than programs written in most other languages.
>
> Many different kinds of application are supported without the language becoming too large to implement on small machines. Clearly this is a vital consideration to micro-computer users.

Finally, Pascal is an orderly language which encourages a systematic approach to program development.

A Standard for Pascal has been prepared under the auspices of the International Standards Organisation (ISO), and copies can be obtained from the British Standards Institution.

PRO PASCAL

Pro Pascal complies with all the requirements of ISO 7185, Level 0 (i.e. excluding conformant array parameters).

There are extensions for character string handling, double precision floating-point arithmetic, random access to files, and for separate compilation of program segments.

The Pro Pascal compiler is a true compiler, generating native machine code for efficient program execution.

FORMAT OF THIS MANUAL

The manual is divided into three parts.

Part I is a guide to the main features of Pascal, intended for the reader having some familiarity with Basic or Fortran. It presents the topics in a "learning" rather than a "reference" sequence, and covers sufficient ground to enable many practical programs to be produced without going into all the possibilities.

Part II forms a detailed reference manual for writing programs in Pro Pascal. It describes all the features of the language, including the extensions and the facilities related to the operating system.

Part III contains the directions for operating the software (compiler, link-editor, etc.), the options available, format of diagnostics, hints on program testing, and suchlike matters. There are also details of hardware requirements and installation procedures.

There are appendices giving the formal syntax, the compile-time and run-time error codes, and the ASCII character set.

It is not possible in the scope of a manual such as this to provide instruction in Pascal for the complete novice. A number of books are available which do this, and the names of a few will be found at the end of Part I.

PART I - INTRODUCTION TO PASCAL

1 EXAMPLE PASCAL PROGRAM

This part of the Pro Pascal user manual is intended to provide readers having some preliminary knowledge of programming (in Basic or Fortran, for instance) with an introduction to the main features of Pascal. The presentation describes the Pro Pascal language, including a few of the extensions which are not part of strict Standard Pascal. The objective has been to provide sufficient information to enable many practical programs to be written.

To introduce the general form and appearance, this section contains a complete example program called "results", which reads the results of a competition, tabulates them with the average score for each entrant, and at the end gives the winner of the competition. The winner is the entrant having the highest average from five or more events.

The input to the program is to be presented in the form of lines, each line starting with a competitor's number (3 digits), followed by his scores in up to eight events (scores in the range 0 to 100). The text of the program, and a small sample tabulation, are given below.

Sample output:

```
105      76   65   47   59   81   69              397  66.17
108      55   58   68   67   42                   290  58.00
110      67   39   72   73   65   71              387  64.50
114      70   78   76   82                        306  76.50
119      69   43   38   46   39                   235  47.00
121      52   47   32   43   48   55   72         349  49.86
122      74   56   65   42   88   81              406  67.67
124      46   63   72   42   59   60              342  57.00
127      50   51   36   48   67                   252  50.40

   Winner is number  122  with average 67.67
```

```
PROGRAM results (input);

  CONST maxevents = 8;        {maximum events for one entrant}
        colwidth = 5;         {column width on tabulation}

  TYPE  competitor = 100..999;      {range of entrants' numbers}
        score = 0..100;       {possible scores for one event}

  VAR   thiscomp, winner: competitor;
        eventscore: score;  totalscore: integer;
        eventcount: 0..maxevents;
        average, winningav: real;
        listing: text;        {output file for tabulation}

  BEGIN
    winningav := 0;
    assign (listing, 'RESULTS.PRN');  rewrite (listing);

        {process input and produce listing}
    WHILE NOT eof (input) DO
      BEGIN      {read competitor number}
        read (thiscomp);
        write (listing, thiscomp:5, ' ':3);

        eventcount := 0;  totalscore := 0;
           {now his scores until end-of-line}
        WHILE NOT eoln (input) DO
          BEGIN
            read (eventscore);
            write (listing, eventscore:colwidth);
            totalscore := totalscore + eventscore;
            eventcount := eventcount + 1;
          END     {of processing one result};

            {space across to totals column}
        IF eventcount < maxevents THEN
          write (listing, ' ': (maxevents-eventcount)*colwidth);
            {calculate & print average}
        IF eventcount = 0 THEN average := 0
        ELSE average := totalscore / eventcount;
        writeln (listing, totalscore:8, average:7:2);
            {is average greater than current winner ?}
        IF (eventcount >= 5) AND (average > winningav) THEN
          BEGIN
            winner := thiscomp;    {best so far}
            winningav := average;
          END;
        readln ;
      END     {of processing one competitor};

            {at end of input, print winner}
    writeln (listing);  writeln (listing);   {blank lines}
    IF winningav > 0 THEN
      writeln (listing, '    Winner is number', winner:5,
                        ' with average', winningav:6:2)
    ELSE writeln (listing, '    No entrant qualified as winner');

  END.
```

Note that the layout of the program text is arranged to help the eye follow the general shape, which consists of some initialisation, a process to be carried out for each entrant, and finally the printing of the winner. The processing of an entrant can be subdivided into the reading of his number, a repeated section dealing with his scores in various events, then the calculation of his average and the comparison of this with the current leader.

A number of aspects of Pascal are shown in this example; all will be covered in later sections, but a few points can usefully be made immediately.

1. Both upper and lower case letters are used to make the text easier to read. The compiler does not make any distinction between the cases.

2. Named constants are used for some parameters of the program, allowing these factors to be amended simply. One such factor is the maximum number of events allowed for (set at 8). Another is the column width in the tabulation, which appears in the write operations (set at 5).

3. The range of values for a competitor's number and a score in one event are given as part of the program in lines 6 and 7. This information enables the compiler to produce a program taking account of the anticipated sizes of numbers, and (optionally) to include automatic checking.

2 GENERAL LAYOUT AND APPEARANCE

2.1 Program skeleton

A Pascal program has a general shape that is determined by the following skeleton:

```
Program heading
LABEL declarations
CONST declarations
TYPE declarations
VAR declarations
PROCEDURE and FUNCTION declarations
Program body
```

The heading and the body must be present. All the others are optional, and may be omitted if not needed, though a program without any variables would be very limited in what it could do.

The program heading consists of the word PROGRAM, followed by the program name. The program body contains the statements which determine the actions of the program. It is a rule of Pascal that objects must be declared before they are used, and the various declarations that come between the heading and the body are the means of doing this. It is worth noting that while any declarations that are not needed can be omitted, the ones that are present must be in the order listed.

2.2 Symbols, words, and constants

The text of a Pascal program is made up of words, special-character symbols, and constants. The symbols are used for "punctuation" (for instance comma, semicolon), to represent operations to be carried out (+, -, *), and to distinguish special constructs. These uses will be introduced as they are needed. The next subsection deals with words. Constants are generally written just as in normal usage:

```
5     10     520     -6     3.4
```

(the last being a "real" or floating-point value). Character-string constants are placed between quote marks, e.g:

```
'I am the greatest.'
```

Pro Pascal also allows integer constants to be written in hexadecimal, as for instance:

    100H   0FFH   5CH

(A leading zero must be present if the constant would otherwise start with a letter.)

2.3 Identifiers and reserved words

Objects (variables, for instance) which are introduced into a program are given "identifiers" by the programmer. An identifier is a name, made up from letters and digits, starting with a letter. Pro Pascal also allows the underscore character "_" to be used within identifiers to improve readability. (This is not part of strict Standard Pascal.) Examples:

    account   last_used   P5   wordlength

An identifier is separated from the next object in the program by any character which is not a letter or digit. Thus a space can be (and often is) used as a separator, and the end of a line similarly. Any number of spaces can precede a component of the program, and may be used to help readability.

The programmer has a great deal of freedom in selecting names for objects. In Pro Pascal there is effectively no limit to the length of a name, though it may be useful to remember that some other Pascal implementations may only differentiate by means of the first eight characters. (A name is terminated by end-of-line, so cannot exceed the length of a line, which is limited to 255 characters.) Some words are however "reserved" and given special significance in the language, and may not be used as names. Some of the commoner reserved words are

```
PROGRAM  CONST  TYPE  VAR
PROCEDURE  FUNCTION  BEGIN  END
IF  THEN  ELSE  CASE  REPEAT  UNTIL
FOR  TO  DO  WHILE  AND  OR  DIV  MOD
```

A complete list is given in part II.

(Reserved words may be thought of as extending the repertory of special characters, though the words are of course chosen to be appropriate and help in understanding the program.)

2.4 Program appearance

For the purpose of obtaining the meaning from the program, the compiler makes no distinction between upper and lower case letters, nor does it give any importance to layout in the sense of what is collected on one line and what is put on the next. The human eye, nevertheless, gets a lot of help in its understanding of the program text from such points of appearance. In this manual, upper case is used for reserved words (PROGRAM, BEGIN, WHILE etc.) and generally lower case for identifiers (lastused, wordlength).

Indenting the left-hand margin is also a great help in conveying the meaning of a program to the human reader. The example in section 1 shows this to some extent, and the suggested approach is described below with the kinds of statement which benefit.

2.5 Comments

A comment can be introduced into the program text anywhere that a space would be allowed. A comment can be delimited either by matching curly brackets {...}, or by the equivalent (*...*) if curly brackets are not available.

3 STATEMENTS

Statements describe the actions of a program, and for this reason are described first. To be complete, however, many statements need variables to act upon. For the purpose of this section, we assume that the variables named have already been declared.

3.1 Expressions

There are many instances in the description of statements where an expression may be used. A simple form of expression can be just a single variable or constant, and many actual expressions even in complicated programs are no more than this. An expression is also, though, the means of specifying arithmetic or logical operations, and for such purposes follows the notation found in many programing languages, with symbols + for add, - for subtract, and * for multiplying. Pascal makes a distinction between integer division giving an integer result, for which the reserved-word symbol DIV is used, and division giving a floating-point result, which is invoked by "/".

Relational operators can be used:

```
= (equal)                <> (not equal)
< (less than)            <= (less or equal)
> (greater than)         >= (greater or equal)
```

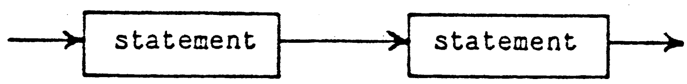
Logical operations are called for by the reserved-word symbols AND, OR and NOT. (Pascal does not allow AND to be used as a masking operation, since that implies implementation-dependent knowledge about the internal representation.) Example expressions:

```
5
intvar
ind + interval
units-10
balance > limit
(a < b) OR (c = 6)
thickness + 4 * (length - height)
```

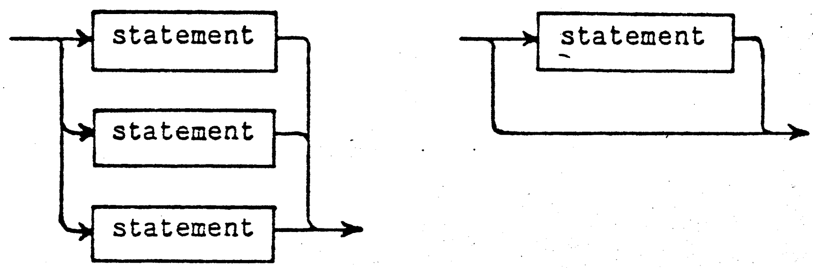
Use of functions within expressions is covered in the description of procedures and functions, and a few other special forms are mentioned as they arise.

3.2 Simple, conditional, and repetitive statements

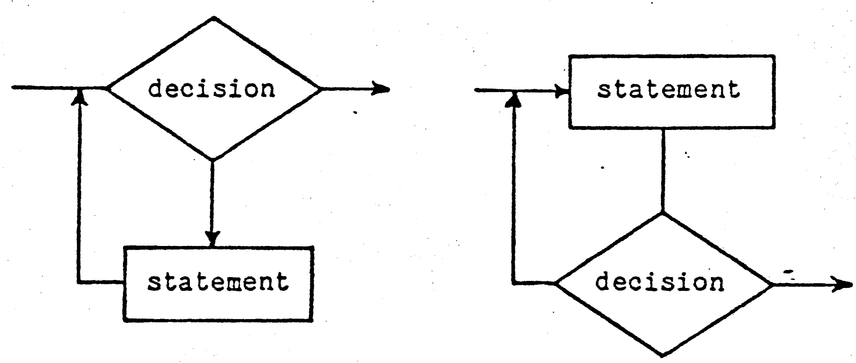
It is useful to categorise statements according to the way the flow of control passes through them. The so-called "simple" statements are obeyed just once:



Conditional statements provide a choice from a number of actions:



Repetitive statements allow for execution of the controlled operation a number of times:



These structures can be put together in any way, since the blocks labelled "statement" can each be of any of the possible forms.

To enable the compiler to process the component statements, the semicolon symbol is used as a separator, for example:

```
statement ; statement ; statement
```

Another important constructional device is the "compound statement", in which a sequence of statements is grouped by BEGIN and END into a single unit:

```
BEGIN  statement ; statement ; .. END
```

This method of grouping is needed when a subsidiary statement of a conditional or repetitive statement is to be a sequence rather than a single statement. For clarity, it is important to lay out a compound statement so that the BEGIN and END can be seen to match:

```
BEGIN
  statement ; {indented}
  statement ;
  .
  .
END
```

3.3 Simple statements

There are just three kinds of simple statement.

(i) An assignment statement gives the value of an expression to a variable:

```
first_time := 2;
next := next + increment;
compound := base + 0.045 * excess;
```

(Note that the compound symbol := is used to mean "set equal to". The symbol = on its own is kept for comparison operations.)

(ii) A procedure statement invokes execution of the procedure - this is described later under "procedures and functions".

(iii) Special instances of procedure calls are the read and write operations, for example:

```
read (input, currentitem);
write (output, 'Average density=', avdensity);
```

(iv) GOTO statement. The control structures of Pascal provide the means of describing the large majority of instances in which a GOTO would be needed in some languages (FORTRAN, for example), and the programmer's aim should be to avoid GOTOs where possible. However, some instances remain where a program is made more contorted and difficult to understand by avoiding GOTO than by using it, for instance:

- to terminate a program from an error or exception routine,
- to exit from a loop at an intermediate point rather than the beginning or end.

3.4 Conditional statements

Pascal has two forms of conditional statement:

(i) The IF statement allows a choice between two alternatives, one of which may be "do nothing":

```
IF index < 10 THEN do_one
ELSE do_two;
IF status > dummy THEN dothis;
```

Notice that do_one must not be terminated by a semicolon (it is in effect terminated by the ELSE).

(ii) The CASE statement allows a choice of one from a number of possibilities. The format is:

```
CASE v OF
  1:  do_one;
  2:  do_two;
  4:  do_four;
END {case}
```

The case constants 1, 2, and 4 must be possible values of the control variable v. The list is terminated by the symbol END.

As shown, any value of v except 1, 2, or 4 is illegal. There is a variation to allow "any others" to be collected together, thus:

```
CASE v OF
  1:  do_one;
  2:  do_two;
  4:  do_four;
  OTHERWISE  do_others;
END {case}
```

Often, the action to be taken on other values is simply "do nothing", as for example:

```
CASE v OF
  1:  do_one;
  2:  do_two;
  4:  do_four;
  OTHERWISE ;
END {case}
```

3.5 Repetitive statements

There are three repetitive statements:

(i) The WHILE statement provides a choice at the beginning:

```
WHILE a < 10 DO process
```

Thus it is possible that "process" may not be entered at all. Process must alter the value of a to terminate the loop.

(ii) The REPEAT statement has its exit at the end, and the controlled statement is obeyed at least once:

```
REPEAT
  process
UNTIL a >= 10
```

(iii) The FOR statement provides a combined loop and count facility:

```
FOR a := 1 TO 10 DO process;
FOR b := 10 DOWNTO 1 DO anotherprocess;
FOR l := 2*n TO twicemax DO yetanother;
```

The variable (a, b, or l) is the "control variable", and the initial and final values are general expressions, of which 1 and 10 are simple examples. In the form e1 TO e2, if e1 is greater than e2 then the loop statement is never obeyed (and similarly in the DOWNTO form if e1 is less than e2). The increment/decrement is always 1.

4 LABELS AND LABEL DECLARATIONS

Labels in Pascal are used only in conjunction with GOTO statements. They take the form of integer values and are "sited" in the program body by appearing in front of a statement, followed by a colon. Each label in a body must have a distinct value, and must be declared in the LABEL declaration group. For example:

```
        LABEL 99;       {error exit}
        .
        .
        IF value > validmax THEN
          BEGIN
            write (output, 'Exceeds maximum value');
            GOTO 99;
          END;
          .
          .
    99: ;
    END {program}.
```

If there is more than one label, the list is presented thus:

```
        LABEL 99, 10, 120;
```

5 CONST DECLARATIONS

The CONST declaration group allows an identifier to be used to represent a constant. There are two main advantages to be gained from doing this:

(a) The name can be chosen to make the program self-documenting;

(b) Multiple references to the same value (e.g. buffer size) can be altered by changing one declaration at the beginning of the program.

Note that (as with other declarations except PROCEDURE and FUNCTION) the word CONST appears just once. Each individual declaration is terminated by a semicolon:

```
CONST columnwidth = 7;
      buffersize  = 128;
      validtext   = 'Valid entry. Date:';
```

6 DATA TYPES AND TYPE DECLARATIONS

It is not possible to separate completely the treatment of types in Pascal from their application to variables, and thus there is in this section some anticipation of the next. It may be found helpful to look ahead to see the overall picture first.

6.1 Data types

The word "type" is used in Pascal with an important and specialist meaning. It describes the structure and attributes of an item of data, not only (as in say Fortran) making the distinction between integer and real values, but also allowing the programmer to define data structures of his own.

(Niklaus Wirth's book "Algorithms + Data Structures = Programs" gives a thorough explanation of the idea of data types, and the title itself shows the importance which he attaches to the subject.)

A variable may only be assigned a value that is appropriate to its type, and similarly there are rules governing the association of types within expressions. These rules are enforced by the compiler, not to be irksome but to ensure that before testing even starts a large proportion of "silly" errors are removed.

6.2 Built-in types

There are five built-in data types that can be used in any program without declaration.

(i) char - the data item is a character, in Pro Pascal (as in many other implementations) one from the ASCII character set.

(ii) integer - the item is an integer. In Pro Pascal the range of integers is nine decimal digits (to be exact: -2147483647 to +2147483647).

(iii) real - the item is a floating-point quantity.

(iv) longreal - an extended-precision floating-point quantity.

(v) boolean - the item is a logical value which may be either false or true. Boolean values often occur "on the fly" as in

```
IF a < b THEN
```

but may also be assigned to appropriate variables.

6.3 User-defined types

6.3.1 TYPE declarations

Any user-defined type may be given a name, and appear in a declaration laid out as follows:

```
        TYPE  typename1 = type1;
              typename2 = type2;
              .
              .
```

Examples will be found in the following sections. Note that a type declaration does not of itself introduce any variables, but just provides a "template" for a data layout.

6.3.2 Enumerated types

The type consists of a list of possible values, set out as for example:

```
        TYPE  dayofweek = (Sunday, Monday, Tuesday,
              Wednesday, Thursday, Friday, Saturday);
```

A variable "day", declared to be of type dayofweek, may take any one of the values Sunday to Saturday, but may not take a value which is not in the list (10, say). If "day" and "today" are both of type dayofweek then

```
        day  := Monday;
        today := day;
        IF today = Tuesday THEN...
```

are all valid, but

```
        IF today = 10 THEN ..
```

is not.

The order of the items in the list may be used, for instance:

```
IF today < Wednesday
```

is true if today is Sunday, Monday, or Tuesday. There are operations "succ" and "pred" to get to the following or preceding value in the list, so that

```
day := Monday;
today := succ (day);
```

leaves "today" with the value Tuesday. Enumerated types may also be used in FOR statements:

```
FOR day := Monday TO Friday DO...
```

and as array indexes (subscripts).

In Pro Pascal, an enumerated type may have at most 256 values.

6.3.3 Subranges

Subrange types are introduced by declarations such as

```
TYPE competitor = 100..999;
     byterange = -128..127;
     weekday = Monday..Friday;
```

They have particular use in defining the range of an array index (and so the size of the array), but in Pro Pascal the compiler also makes use of the information given in a subrange type for deciding the storage needed for individual variables, and also in generating the (optional) extra code for range checking. It is therefore a good general practice to use subrange types wherever appropriate when first writing a program.

6.3.4 Sets

A set declaration is of the form

```
s = SET OF b
```

where b is another type, called the "base" type of the set. The base type must be an ordinal type, by which is meant one having distinct values (not, for instance, the type "real"). The idea of the set is to enable a program to represent economically those members of the base type having some useful common property. For example, the predeclared type char is a valid base type in Pro Pascal, and sets can therefore be constructed which represent all the vowels in the upper-case and lower-case alphabets, or all the characters which cannot be displayed on some particular printing device.

As another example of the way sets can be used, a retailer might have a range of commodity codes 00 to 99, some of which are subject to VAT. Declare

```
TYPE commoditycode = 00..99;
     setofcc = SET OF commoditycode;
VAR VATcodes: setofcc;
```

then in the body of the program, set VATcodes by a statement such as

```
VATcodes := [5,6,8,10..15,22,50..59]
```

and later statements can say for instance

```
IF thiscode IN VATcodes THEN ...
ELSE ...
```

Such a program is clearer and more maintainable than one which has a long sequence of tests to sort out the codes subject to VAT.

The list of values within square brackets is the notation for constructing a set having those members. In the example above all the values are constants, but they may be variables (or indeed any expressions). That example showed a static value given to VATcodes at the beginning of the program. The value of a set variable can of course be changed (as with any other variable), and one might for instance be used as a means of "ticking off" items which arise in an arbitrary or random sequence.

Details of possible operations with sets are given in Part II. In Pro Pascal, the base type of a set may be char, an enumerated type, or a subrange of integer lying within 0 to 2039. The storage allocated for a set variable is determined by the range of the base type.

6.3.5 Arrays

The concept of an array appears in most programming languages (e.g. Fortran and BASIC). In Pascal, the declaration of an array must specify an index type, defining the range of index values, and a component type.

```
TYPE intarray = ARRAY [0..9] OF integer;
     realvect = ARRAY [1..5, -10..10] OF real;
     dayletter = ARRAY [dayofweek] OF char;
```

The last of these declares a data structure which has one character for each value of the enumerated type dayofweek (see 6.3.2 above).

The components of an array may be of any other data type. An array of sets, for example, would be permissible and might be useful. An array of records is allowed, as is an array of files.

An array may sometimes be referenced whole for the purpose of assigning it to another array of the same type. More commonly, the individual elements are referenced by putting an index after the array name:

```
daycode[Friday] := 'F';
IF subvec[j*3+1] > k THEN...
```

The index can be any expression that evaluates to the declared index type.

6.3.6 Strings

In strict Standard Pascal, the term "string-type" is given to types of the form

```
PACKED ARRAY [1..n] OF char
```

Such types are compatible with character string literals of the same length for purposes of assignment and comparison, for example

```
dayname := 'Friday   ';
IF month = 'Apr' THEN ...
```

and similarly with other string-type variables of the same length.

Pro Pascal also implements dynamic-length strings. These string variables have a maximum length given in their declarations, which are of the form "string[n]". During execution of the program, they may take values of any length up to the given maximum. Character string literals can be assigned and compared, provided that the declared length is not exceeded; a PACKED ARRAY string-type variable can also be assigned to a dynamic string, but not vice versa. There is a limit of 255 characters on the declared length.

An important aspect of the use of dynamic strings is the set of procedures and functions which are provided to perform insertion, deletion, and other operations. See 8.6.1 below.

The declaration "string" without a length specification is accepted as an abbreviation for "string[80]".

6.3.7 Records

An array-type describes a uniform collection of elements of the same component type. A record on the other hand is a grouping of pieces of data which are not necessarily related in form. It is a common concept of data processing, and is found for instance in Cobol and PL/1. While the elements of an array are selected by index values, the fields of a record are named.

The component fields may be of any other data type. An array, for example, may be part of a record, as may another record, or even a file.

The form of declaration may be seen in the following example:

```
TYPE makes = (Ford, Bedford, Leyland, AEC, Scammell);
     date  = RECORD
               day: 1..31; month: 1..12;
               year: 1900..1999;
             END {date};
     vehicle = RECORD
                 makercode: makes;
                 registration: string[7];
                 mileage: integer;
                 lastservice: date;
               END {vehicle};
```

If a variable is now declared as

```
        VAR truck: vehicle;
```

the fields are referenced by name as for instance

```
        truck.makercode
        truck.registration[1]
        truck.lastservice.month
```

(since "date" is a record within a record, its individual fields require the further extension).

It might be more useful to have an array of such records, as for instance

```
        VAR trucks: ARRAY [1..50] OF vehicle;
```

in which case an index is needed to choose an individual entry.

```
        trucks[thisone].makercode
        trucks[thatone].mileage
```

To simplify (and make more efficient) the references to record fields, Pascal has a WITH statement. It specifies a particular record, and the field names can then be used on their own.

```
        WITH trucks[thisone] DO
          BEGIN
            mileage := mileage + miles;
            IF lastservice.month < duemonth THEN...
            .
            .
          END
```

The WITH statement can equally be used to specify a single record variable such as "truck", to avoid frequent repetitions of "truck" before the field names.

There are other facilities of records, including a method of describing variants, which are covered in Part II of this manual.

6.3.8 Pointers

Besides the variables considered up to now which are allocated at compile time, Pascal includes a dynamic storage facility known as a "heap". Space can be taken from and returned to the heap at any time during the running of an object program, according to the requirements of each particular execution.

Objects in the heap are addressed through pointers. A pointer is a variable which is associated with a particular data type, and must always point to an object of that type (unless it is currently unused when it should be given the special value NIL). For instance, a pointer to the type "vehicle" in 6.3.7 might be introduced by

```
TYPE ptvehicle = ^ vehicle;
```

and a pointer variable declared

```
VAR ptruck : ptvehicle;
```

To get space in the heap for a vehicle record, and to set ptruck to point to it, the following statement is used

```
new (ptruck)
```

after which the record can be filled in by statements such as

```
ptruck^.makercode := AEC
```

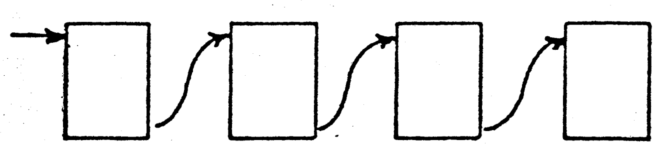
(Note that the up-arrow comes before the name in the type declaration, but after it when making references.)

When processing is complete, the statement

```
dispose (ptruck)
```

returns the space to the dynamic pool.

Of course, this particular example would not be a worthwhile use of the apparatus. The full value of the heap becomes more apparent in situations such as a program which requires two large structures but not both at once. And the full versatility may be gauged from the idea of adding a new field of type ptvehicle to the vehicle record, which allows a chain of records to be built up to any length:



6.3.9 Files

A file in Pascal is a data structure having an indefinite number of components. In practice, files are generally implemented as the means whereby programs can transfer data to or from discs or other external devices. They are best considered as being of two main kinds: text files, and others.

6.3.9.1 Text files

A text file is composed of characters grouped into lines. It is therefore the natural means of communication with the user, through console or listing. There are facilities for automatically converting values between internal and external representations as they are read from or written to text files, and in the case of output the program can control the layout. A text file is declared as, for example:

```
VAR listfile: text;
```

(Text files are equivalent to formatted files in Fortran.)

6.3.9.2 Other files

Files based on other data types can be used, typically for intermediate storage, or transfer of data from one program to another without involving conversion. Returning to the example of "vehicle" as a record type, a file may be declared as

```
VAR fleet: FILE OF vehicle;
```

that is, "fleet" is a series of records describing vehicles. Note, first, that all the components of a file are of the same type. (The use of "variant" records, described in Part II, makes this a less severe restriction than it may seem at first.) Also, just one component is accessible to the program at a time, as though a "window" was moved along the file through which one component can be viewed. In Standard Pascal the window can only move sequentially; Pro Pascal provides, in addition, a random-access facility.

The file components do not have to be records - FILE OF integer, for instance, is perfectly valid. Pro Pascal provides automatic blocking of small components. The only prohibition is a combination of declarations which defines one file within another one.

6.3.9.3 Common concepts

The operations read and write are available with any file. For non-text files, they have the effect of moving one component between the file (at the current position of the "window") and a program variable, for example

```
        read (fleet, truck)
```

copies the current component from the file "fleet" to the variable "truck", and at the same time moves the window to the next component. The basic operation on a text file is similar; for instance, if ch is a variable of type char, the statement

```
        write (listfile, ch)
```

moves the value of ch to the current position in listfile and advances the window. However, there are further possiblities with textfiles that are discussed below.

Another characteristic of all files is the concept of "eof" (or end-of-file). Check for this condition before a read operation by a statement such as

```
        IF eof(fleet) THEN summary
```

(No special steps have to be taken at the end of writing the file; the file-handling software simply notes the last component.)

Before it is addressed by read or write operations, a file must be set into the input or output condition by one of the statements

```
        reset (f);  {for input}
        rewrite (f);  {for output}
```

These have the effect of positioning the file window at the first component. The sequence of operations on a work file might be as follows:

```
        rewrite (work);    {prepare for output}
        write (work,..);   write (work,..); ...
        reset (work);    {back to start & prepare for input}
        read (work,..);    read (  );  ...
```

The standard text files "input" and "output" can be used by any program without having to be declared or any reset or rewrite given.

Any implementation generally has methods of associating Pascal files with specific devices or disc files. The Pro Pascal arrangements are discussed in the "implementation dependent" section of Part II.

6.3.9.4 Special features of text files

Text files have the special property that the basic file components (characters) are held within a substructure of lines. The way this is defined is designed to be independent of the method of determining the end of lines in any particular hardware or operating system.

When writing, the end of each line is indicated by a writeln operation, e.g.

```
writeln (output)
```

For reading, an end-of-line condition similar to the end-of-file condition is introduced, obtained by the operation eoln. This condition is true when the file window is at a point where a writeln was given. The readln operation skips any remaining characters on the current line and positions the window at the first character of the next line (or eof becomes true).

Read and write operations on text files can specify multiple transfers within the same line, e.g.

```
write (output, 'Total-', total)
```

The line termination can be included as well, by using writeln instead of write.

Conversion operations are automatically supplied when reading or writing values in internal representation such as integer or real (though not for user-defined types such as enumerated). On writing, the layout can be controlled by specifying a field width with the value, thus

```
write (output, value:width)
```

where both "value" and "width" are in principle expressions. If the value is of type integer, it will be displayed right-justified in a field of the specified width. When width is omitted, Pro Pascal displays integer values right-justified in a field of 11 characters. If the significant digits of the output are more than the given width, the width is exceeded. As a consequence, a width of 1 gives a left-justified output. Further options are available with real values, as described in Part II.

A special facility of Pro Pascal is the "append" operation, which can be used instead of rewrite when new information is to be added to an existing file.

6.3.9.5 Special features of non-text files

In Standard Pascal, all file operations are sequential. Pro Pascal has additional facilities for random access to non-text files; the operation

```
seek (ntfile, elnumber)
```

positions the file window at the specified element number. Read (or write) operations following take effect from this position. Random read operations can be performed on a file written sequentially, and for this purpose reset should be specified to initialise the file.

The "append" facility described above is also available for non-text files, allowing data to be added to an existing sequential file.

The operation "update" is also provided in Pro Pascal as an alternative to reset and rewrite, indicating that both forms of access are to be used. Update operation is inherently less secure than the sequential file processing of Standard Pascal, and should be used with appropriate safeguards against system malfunctions (regular backups in particular). It is also not intended that update operations be done on an empty file; a sequential initialising process should be carried out first.

7 VARIABLES AND VAR DECLARATIONS

7.1 Variable declarations

Variable declarations instruct the compiler to allocate space in the object program, and to associate with each variable a type (which among other things dictates the size of the item). For example:

```
VAR thiscomp, winner: competitor;
    eventscore: score;
    totalscore: integer;
    listing: text;
```

(from the sample program in section 1). The types integer and text are built-in with predeclared significance (which the programmer can redefine if he wishes), whereas type declarations for competitor and score must already have been encountered.

The type quoted in a variable declaration need not be in the form of an identifier - any of the forms described in the previous section can be written after the colon, for example:

```
VAR linecount: 1..66;
    fleet: FILE OF vehicle;
    answer: (yes, no, dontknow);
```

The variables of one type also do not have to be listed together (as thiscomp and winner), though it is often helpful to do so.

7.2 Reference to variables

The forms of reference to various types of variable have already been shown. To summarise:

A complete ("entire") variable is referenced simply by the variable name.

An array element is selected by an index expression in square brackets - a[e].

A field in a record is selected by the field name separated from the record reference by a period (full stop).

The object pointed at is obtained by following the pointer reference with an up-arrow (^).

Because Pascal allows such combinations as an array of records, or a record having an array as one of its fields, or a record as part of a larger record, the selection of an elementary item may need to be done in stages. In all cases it is a matter of progressive refinement, following a logical path to the required object by use of the four forms shown above.

8 PROCEDURES AND FUNCTIONS

Procedures provide one of the most valuable methods of subdividing a program into manageable pieces, as well as allowing for commoning-up of similar sections of code.

8.1 Blocks

The program skeleton shown in section 2.1 consists of a program heading followed by:

LABEL declarations
CONST declarations
TYPE declarations
VAR declarations
PROCEDURE and FUNCTION declarations
Program body

This collection (from LABEL to body) is called a block, and a procedure declaration is formed from a procedure heading and a block. Since the procedure block can include procedure declarations, it follows that the first procedure can have further procedures inside it (like the big fleas and little fleas).

For many programs, however, it is sufficient to collect the procedures at one level, thus:

```
PROGRAM able;
  TYPE ...
  VAR  ...

  PROCEDURE alpha;
    VAR ...
    BEGIN
      {body of alpha}
    END;

  PROCEDURE beta;
    TYPE ...
    VAR  ...
    BEGIN
      {body of beta}
    END;

  FUNCTION gamma: real;
    BEGIN
      {body of gamma}
    END;

BEGIN
  {program body}
END.
```

Indenting and comments are useful in showing up this structure to the eye.

There is one important constraint to observe. In the example above, statements in the body of beta can use procedure alpha, but without special arrangements the reverse is not true. Often this constraint is not difficult to live with: it is simply necessary to put the more primitive, low-level procedures first.

The above, and much else in this section, applies equally to functions.

8.2 Scope

One of the important characteristics of a block is its opaque quality from outside. Procedure alpha can be used by beta or the program body, but anything declared inside it (a variable, for instance) is invisible and cannot be referred to from outside. On the other hand, the block is "transparent" from inside, and the body of alpha can use types or variables from the main program. The subdivision of the program into watertight compartments makes the whole thing more secure, and allows attention to be given to a reasonable-sized portion of the problem at once.

The term "scope" is used to mean that part of the whole program text over which the declaration of a name applies. It is, generally speaking, the block in which the declaration occurs, and any blocks nested within it. The same name can be re-used in an inner block - though this is not on the whole a good practice, being confusing to humans - in which case the "nearest" declaration is the one which is taken at any reference.

8.3 Declaring and using simple procedures

A procedure such as alpha in 8.1 is very like a miniature program. It can have its own "local" variables, which come into existence when the procedure is used and vanish when control reaches the end of the body and returns to the point of call.

To call alpha, the statement

```
alpha;
```

is used.

8.4 Parameters

Procedures as so far described are a useful means of subdividing programs, but rely on variables of an enclosing block (typically the main block) for communication. Parameters give an important extension to the independence, and hence the structural value, of procedures.

A parameter is a variable of the procedure which is filled in at the time of call. It has the advantage of being local to the procedure, and hence private except at the time the procedure is invoked. For example,

```
PROCEDURE upper (fch: char);
  BEGIN
    IF fch IN ['A'..'Z'] THEN
      writeln (output, 'Upper case');
  END  {upper};
```

Here, upper has a "formal parameter" fch of type char. Each call of upper must supply a character, and upper will display the message 'Upper case' if it is in the range A to Z. The call

```
upper ('X');
```

is obvious, but the supplied value would more usefully be a variable, e.g.

```
read (input, ch);   upper (ch);
```

Incidentally, this shows how the read and write operations are in fact examples of procedures. They are unusual in two ways -

(a) they are used without being declared,

(b) they may have a variable number of parameters.

Some other "standard procedures" are described in section 8.6. User-defined procedures must be declared, and each call must supply the number and types of parameters to match the declaration.

The parameter fch to procedure upper is a "value" parameter - the call can supply any character expression, including a constant. The parameters to the procedure write are of this kind. An alternative form is found in the procedure read, which returns a value to the caller via its parameter. This kind is a VAR parameter, so called because the declaration of such a parameter in a user procedure starts with the word VAR. Within the procedure, the parameter name may appear on the left-hand side of assignments, and the call must supply a variable (which may be an array element or a field in a record) into which the assignment is returned. Before using a VAR parameter to return a value, the called procedure can refer to the current contents of the variable. It is therefore somewhat more versatile, but less safe, than a value parameter.

Here the procedure lower has a VAR parameter of type char:

```
PROCEDURE lower (VAR fch: char);
  BEGIN
    IF fch IN ['A'..'Z'] THEN
      fch:= chr(ord(fch) - ord('A') + ord('a'));
  END {lower};
```

If the variable supplied in the call is an upper-case letter, the procedure replaces it by the lower-case equivalent. (This example uses two further concepts, ord and chr. Pascal does not permit arithmetic operations to be carried out directly on characters, because implementation-dependent assumptions about character codes would then be embedded in programs. For further details, see below and in Part II.)

8.5 Functions

In fact, ord and chr are examples of functions. Other examples are sqrt(x), which returns the square root of the argument x, and the eof predicate mentioned in section 6.3.9 on files. A function is in many respects like a procedure, but differs in that it always returns an answer, and is invoked by quoting the function name where the answer is required, typically within an expression.

A function has a type, which is the type of the answer, and is included in the declaration:

```
FUNCTION lowercase (fch: char): char;
  BEGIN
    IF fch IN ['A'..'Z'] THEN
    lowercase := chr (ord(fch) - ord('A') + ord('a'))
    ELSE lowercase := fch;
  END {lowercase};
```

This example uses the value parameter/function result mechanism to perform the same service as the procedure "lower" in the previous section. The alternative forms of call might be

```
read (input, ch);   lower (ch);
```

and

```
read (input, ch);   ch := lowercase (ch);
```

However, the function can be more versatile in use, as for instance

```
read (input, ch);    write (output, lowercase (ch));
```

8.6 Standard procedures

In Pascal, a number of procedures (known as "standard procedures") are provided as part of the language, and can be used without having to be declared. The file-handling procedures read and write introduced earlier (see 6.3.9) are examples; others are the math functions sqrt, sin, cos, exp, ln, and arctan which can be used within expressions whenever needed.

8.6.1 String-handling procedures

A further category is the group of procedures and functions used for manipulating dynamic-length strings. The principal ones are

length(s)	a function which gives the current length of string s
copy(s,i,n)	a string function which gives n characters from string s starting at character i
delete(sv,i,n)	a procedure to delete n characters from string variable sv starting at character i
insert(s,sv,i)	a procedure to insert string s into string variable sv at position i
concat(s1,s2,...)	a function which gives the "concatenation" of s1, s2, etc.

Others provide for searching within a string for a given substring, and for converting an integer from internal form to decimal. Details will be found in Part II.

Example program using strings:

```
    PROGRAM list (input,output);

          {Copy a textfile from "source" to "output"
           with line numbers.}

    VAR
          source: text;      {source file}
          name: string[10];
          filename: string[14];
          line: string[120];
          linecount: 0..9999;

    BEGIN
          linecount := 0;
          write('Source name - ');  readln(name);
          WHILE (length(name) > 0) AND (name[1] = ' ') DO
            delete(name,1,1);
            {remove any leading spaces}
          filename := concat(name,'.PAS');
          assign(source,filename);  reset(source);

            {now copy from source to output, a line at a time}
          WHILE NOT eof(source) DO
            BEGIN
              linecount := linecount + 1;
              readln(source,line);
              writeln(output, linecount:4, ':  ',line);
            END;
    END.
```

9 GETTING STARTED

The information in the previous sections is sufficient to allow quite advanced Pascal programs to be produced. A few topics (program segmentation, for instance) have been omitted from the sequential presentation to avoid confusion in the early stages. Part II contains a fully detailed description in reference format, and should be consulted if any queries arise, or features not covered in Part I are to be used. This section is devoted to some practical guidance for those who may not be familiar with how a language system such as Pro Pascal is actually used.

9.1 Think ahead

Except for comparatively trivial programs, Pascal cannot really be composed at the keyboard. Plan at least the general shape of a program on paper, with particular reference to any data structures. If there is a significant amount of processing code, consider how it might be given shape and clarity by subdividing it into procedures; even if a procedure is only called from one place, it helps to concentrate the logic and make errors simpler to track down.

Once a program has been given a suitable initial shape, Pascal is very malleable. Statements can easily be added or moved, made conditional or put within a loop. Sections of code can be extracted into procedures, giving the possibility of introducing new local variables and the independence provided by the procedure structure.

9.2 Choice of names

Names in Pascal form an important part of the self-documenting aspect of any source text. Variables called i and j, for instance, give away little of their purpose in their names, and should be avoided, except possibly as very localised loop counts. (There is not even any built-in rule that i should be of type integer, though it would be perverse to use the name for, say, an array of characters.) Meaningful names - vehicle, printentry, scanlist, today, linecount - make all the difference to readability and hence ease of testing and maintenance.

9.3 Compilation and linking

When a source program has been entered into the computer, it must go through two stages before it can be run. The source is compiled, and the output from the compilation is then linked with a selection of routines from the Pascal library to form an executable object program. (This arrangement will be familiar to most users of Fortran.) Directions for operating the compiler and the linker will be found in Part III.

During the compilation process, thorough checks are made that the program obeys the Pascal language rules. Any violations are reported, with the type of error and its position. After correcting the errors, the compilation must be retried. As a result of this (perhaps apparently frustrating) sequence, many small errors are in fact put right in a short period of time. For example, because all objects must be declared before use, any mis-spelled or incorrectly keyed names can be eliminated.

Errors of logic in the program, however, may remain (including possibly the typing mistake which turns one intended name into another legitimate one). These can only be found by linking the compiler's output and trying to execute the result. If the program is a large one, it may be worth inserting a few extra statements such as

```
writeln ('Initialisation complete');
```

which can easily be removed later.

A number of kinds of error are trapped at run time by the routines from the library, and may be located from the information displayed. There are also extra checks which can optionally be included in the object code by the compiler, and may help in the detection of such things as use of variables before any value has been given to them. Details of these aids will be found in Part III.

9.4 Conclusion

Pascal presents many more possibilities than Basic or Fortran, and consequently takes a little longer to learn to use, but the trouble taken is amply repaid. The professional will appreciate for example that procedures can be collected and used again in different programs, or how simply file-processing operations can be programmed, because such things improve productivity. And it is not necessary to be a professional to feel the sense and logic of a well-structured program. It was one of the motives behind the design of Pascal to improve the reliability of software, and it forms a valuable tool in achieving that purpose.

10 FURTHER READING

This User Manual is not intended to be a Pascal primer, or to deal with every aspect of the definition and use of the Pascal programming language. Among the many publications which address these topics, the following - each with its own distinctive approach - are certainly worth investigating.

(1) K. Jensen and N. Wirth
"Pascal User Manual and Report"
Springer-Verlag, 1975

(2) N. Wirth
"Algorithms + Data Structures = Programs"
Prentice-Hall, 1976

(3) J. Welsh and J. Elder
"Introduction to Pascal"
Prentice-Hall, 1979

(4) P. Grogono
"Programming in Pascal"
Addison-Wesley, 1980

(5) L.V. Atkinson
"Pascal Programming"
John Wiley & Sons, 1980

(6) D. Fox and M. Waite
"A Pascal Primer"
Sams, Indianapolis, 1981

(7) I.R. Wilson and A.M. Addyman
"A Practical Introduction to Pascal - with BS 6192"
Macmillan Computer Science Series, 1983

In a rather special category is the definition of ISO Standard Pascal, which is now available.

BS 6192: 1982
"Specification for computer programming language Pascal"
British Standards Institution
(ISBN 0 580 12531 9)

While not easy reading, it is clearly a document of importance to all serious users of the language.

PART II - PRO PASCAL LANGUAGE DEFINITION

1 LEXICAL ASPECTS

Considered from the aspect of its representation on the printed page, rather than with regard to its syntax or meaning, a Pascal program can be viewed as a sequence of lexical "tokens" interspersed with "separators". The forms which these two kinds of lexical entity may take are described in 1.1 and 1.2, respectively.

The length of a source line may not exceed 255 characters.

(The notation used, throughout this manual, for defining the Pascal syntax is described in Appendix A.)

1.1 Tokens

These are of 6 kinds:

```
token = special-symbol ¦ identifier ¦ directive ¦
        label ¦ unsigned-number ¦ character-string
```

1.1.1 Special symbols

The special-symbols are tokens with special fixed meanings.

```
special-symbol = "+" ¦ "-" ¦ "*" ¦ "/" ¦ "=" ¦ "<" ¦ ">" ¦
                 "[" ¦ "]" ¦ "." ¦ "," ¦ ":" ¦ ";" ¦ "^" ¦
                 "(" ¦ ")" ¦ "<>" ¦ "<=" ¦ ">=" ¦ ":=" ¦
                 ".." ¦ word-symbol
word-symbol = "AND" ¦ "ARRAY" ¦ "BEGIN" ¦ "CASE" ¦ "COMMON" ¦
              "CONST" ¦ "DIV" ¦ "DO" ¦ "DOWNTO" ¦ "ELSE" ¦
              "END" ¦ "FILE" ¦ "FOR" ¦ "FUNCTION" ¦ "GOTO" ¦
              "IF" ¦ "IN" ¦ "LABEL" ¦ "MOD" ¦ "NIL" ¦ "NOT" ¦
              "OF" ¦ "OR" ¦ "OTHERWISE" ¦ "PACKED" ¦
              "PROCEDURE" ¦ "PROGRAM" ¦ "RECORD" ¦ "REPEAT" ¦
              "SEGMENT" ¦ "SET" ¦ "THEN" ¦ "TO" ¦ "TYPE" ¦
              "UNTIL" ¦ "VAR" ¦ "WHILE" ¦ "WITH"
```

To allow for them not being available on all keyboards, three of the special-symbols have alternative representations:

symbol	alternative
[	(.
]	.)
^	@

In the spelling of word-symbols, as elsewhere in Pascal (except within character-strings), upper- and lower-case letters may be used interchangeably.

Note that word-symbols are "reserved" words: they are not available to the programmer for use as identifiers.

1.1.2 Identifiers

Identifiers are used to denote constants, types, fields, variables, procedures and functions. They are constructed from letters, digits and underscore characters, starting with a letter:

```
identifier = letter { ( letter | digit | underscore ) }
letter = "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" |
         "i" | "j" | "k" | "l" | "m" | "n" | "o" | "p" |
         "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" |
         "y" | "z"
digit = "0" | "1" | "2" | "3" | "4" |
        "5" | "6" | "7" | "8" | "9"
underscore = "_"
```

Identifiers may be arbitrarily long (but may not extend over more than one line). All characters except underscore are significant in distinguishing among identifiers. No distinction is made between the upper and lower case of a letter.

Examples:

```
prime_number    Z80    UP2down4
```

1.1.3 Directives

```
directive = "FORWARD" | "EXTERNAL"
```

Directives are identifiers with special meanings (see 8.1.3). Because they are not "reserved" words, they may be redefined within the source program (although this would seem an odd thing to do).

1.1.4 Labels

A label is a sequence of decimal digits with a value in the range 0..9999.

```
label = digit-sequence
digit-sequence = digit {digit}
```

A label is uniquely identified by its value, so that 2 and 00002, for example, represent the same label.

1.1.5 Unsigned numbers

These are of three types, integer, real and longreal:

```
unsigned-number =
     unsigned-integer | unsigned-real | unsigned-longreal
```

1.1.5.1 Unsigned integers

These may be in either decimal or hexadecimal notation:

```
unsigned-integer = decimal-integer | hexadecimal-integer
decimal-integer = digit-sequence
hexadecimal-integer = digit {hexdigit} "H"
hexdigit = digit | "A" | "B" | "C" | "D" | "E" | "F"
```

Again, no distinction is made between upper- and lower-case letters.

Whichever of the two representations is used, the value must lie in the range 0..maxint, where maxint = 2147483647.

Examples:

```
1066       OFFH
```

1.1.5.2 Unsigned reals

These must be in fixed- or floating-point decimal notation:

```
unsigned-real =
    decimal-integer "." digit-sequence ["E" scale-factor] |
     decimal-integer "E" scale-factor
scale-factor = [sign] decimal-integer
sign = "+" | "-"
```

E means "times 10 to the power of", and may be in upper or lower case.

Examples:

```
10.0     1e-10     0.314159265E1
```

1.1.5.3 Unsigned longreals

These must be in floating-point decimal notation, and are distinguished from real constants in that the decimal exponent is introduced by "D" rather than "E":

```
    unsigned-longreal =
        decimal-integer ["." digit-sequence] "D" scale-factor
```

D means "times 10 to the power of", and may be in upper or lower case. Longreal constants are held to greater precision than real constants (see 6.1.1.1). Examples:

```
    1D0        0.1234567890123456d-99
```

1.1.6 Character strings

A character-string is a sequence of one or more ASCII characters enclosed between apostrophes. If the string is to contain an apostrophe, this is denoted by an "apostrophe image", which consists of two adjacent apostrophes:

```
    character-string = "'" string-element {string-element} "'"
    string-element = string-character | apostrophe-image
    string-character = ASCII-character
    apostrophe-image = "''"
```

A character-string containing just one string-element is a constant of the standard type char (see 6.1.1.1.5).

A character-string containing n string-elements, with n in the range 2..255, is a constant of the type

```
        PACKED ARRAY [1..n] OF char
```

(see 6.1.2.1).

Examples:

```
    'x'
    'This string has 30 characters.'
    ''' is an apostrophe'
```

1.2 Separators

These are of three kinds:

```
separator = space | end-of-line | comment
space = " "
comment = "{" any-sequence-of-ASCII-characters-and-
              end-of-lines-not-including-right-brace "}"
```

Zero or more separators may occur between any two consecutive tokens. At least one separator must occur between any pair of tokens consisting of word-symbols, directives, identifiers, labels or unsigned-numbers. No separators may occur within tokens.

1.2.1 Comments

To allow for the possibility of left- and right-brace characters not being available, "(*" may be substituted for "{" and/or "*)" may be substituted for "}", in a comment.

For example: (* This is a correctly formed comment.}

If the character immediately after the "{" is "$", then the comment may represent a "compiler directive". The Pro Pascal compiler recognises two such directives: source file insertion, and page throw on listing. (Both are extensions to Standard Pascal.)

1.2.1.1 Source file insertion

If the character after $ is I (or i), the comment is treated as a request to the compiler to include the contents of another source file at the point in the text at which the "comment" occurs. For example:

```
{$I typedefs}
```

causes the inclusion of the source file TYPEDEFS.PAS. Any spaces after the I are ignored, and the remainder of the comment is treated as a CP/M filename. If the filename has no extension, .PAS is supplied.

Inserts may be nested, to a maximum depth of 4.

This facility is disabled if the "Accept only strict Standard Pascal" compile-time option is in force.

1.2.1.2 Page throw on listing

If the character after $ is P (or p), the comment is treated as a request to the compiler to insert a page throw (form-feed) into the listing file at that point (assuming that the L compile-time option is in force). Example:

```
{$P}
```

2 PROGRAMS, SEGMENTS AND BLOCKS

The unit of input to the compiler is a program or a segment. Each has, roughly, the form of a procedure declaration.

```
compilation-unit = program ¦ segment
```

An executable Pascal program is composed, in source terms, of a program together with zero or more segments. Each is separately compiled, and then linked together to form the executable program. Execution commences at the beginning of the statement-part of the program. Control passes (temporarily) to a segment only when a procedure or function in that segment is called (from the main program or from a segment): a segment does not have any statement-part.

2.1 Programs

```
program = program-heading ";" block "."
program-heading =
     "PROGRAM" identifier [ "(" global-parameter-list ")" ]
global-parameter-list = identifier-list
identifier-list = identifier {"," identifier}
```

The identifier following PROGRAM is the program name, and has no further significance within the program. The identifiers in the global-parameter-list may optionally, but are not required to, name any files used within the program. The syntax is accepted, but otherwise ignored, in order to preserve compatibility with other Pascal systems.

The concept of "block" is defined in 2.3.

At the beginning of the statement-part of every program, a call is generated to a module in the library which sets up the environment for the program. This includes operations equivalent to the statements reset(input) and rewrite(output), so these standard files may be used by the program without further preparation.

For an example of a complete program, see Part I, section 1.

2.2 Segments

```
segment = segment-heading ";" segment-declarations
        "BEGIN" "END" "."
segment-heading =
     "SEGMENT" identifier ["(" global-parameter-list ")"]
segment-declarations = constant-definition-part
                       type-definition-part
                       variable-declaration-part
                       procfunc-declaration-part
```

The identifier following SEGMENT is the segment name, but has no further significance within the segment. The syntax and meaning of "global-parameter-list" is as in 2.1.

By referring to the syntax of "block" (see 2.3), it will be observed that at the outermost level of a segment, as opposed to a program, the label-declaration-part is absent and the statement-part is trivial (the empty "compound-statement"). Only the procedures and functions within the segment contain executable statements.

As an example of a complete segment, here is one containing just a single function. After being compiled, it could be added to a library of object modules, and would then be available to any Pascal program which declared it as EXTERNAL (see 8.1.3).

```
SEGMENT min;
FUNCTION min (arg1, arg2: integer): integer;
  BEGIN
    IF arg1 < arg2 THEN min := arg1
    ELSE min := arg2;
  END {min};
BEGIN
END.
```

2.3 Blocks

A block consists of declarations, definitions and statements, and is the main ingredient of a program, a procedure declaration or a function declaration.

```
block = label-declaration-part
        constant-definition-part
        type-definition-part
        variable-declaration-part
        procfunc-declaration-part
        statement-part
```

Since a procedure or function can, in turn, contain declarations of procedures and/or functions local to itself, "block" is an essentially recursive concept.

A label or identifier which is declared in a block has a scope which includes any block textually nested within it, except where it is (temporarily) "masked" by having been redeclared in such an inner block.

The first five ingredients in the above definition of "block" all have the nature of declarations, and are treated in sections 4 thru 8. The last - the statement-part - is the subject of section 3. Its formal definition is:

```
statement-part = compound-statement
```

3 STATEMENTS AND EXPRESSIONS

3.1 Statements

Statements denote the actions to be carried out by a program. They may be classified into two groups: simple statements and structured statements.

A statement may be optionally preceded by a label:

```
statement =
    [label ":"] (simple-statement | structured-statement)
```

3.1.1 Simple statements

Simple statements are those which are not made up of other statements. They are of four kinds:

```
simple-statement = empty-statement | assignment-statement |
                   procedure-statement | goto-statement
```

3.1.1.1 Empty statement

This consists of nothing at all, and causes no action to be performed. Thus, anywhere in the Pascal syntax that a statement can occur, one of the options is to put nothing. A particular example is the labelled empty statement, as in:

```
  BEGIN
    IF error THEN GOTO 999;
    {...}
999:
  END
```

3.1.1.2 Assignment statement

The purpose of the assignment statement is to cause the value of an expression on the "right-hand" side to be assigned to a variable, on the "left-hand" side:

```
assignment-statement =
   (variable-access | function-identifier) ":=" expression
```

The type of the expression must be assignment-compatible (see 6.2.2) with the type of the variable on the left-hand side.

If the variable-access involves array indexing (see 3.2.1.1.2) and/or pointer dereferencing (see 3.2.1.1.4), these actions will be carried out before the right-hand-side expression is evaluated.

If the item on the left-hand side is a function-identifier, the assignment statement determines the value which the function will return, when called. The assignment statement must be within the function block. See also 8.1.2.

Examples:

```
z := a*x - b*y
a[i,j] := 0.0
p^.next := NIL
```

3.1.1.3 Procedure statement

A procedure statement denotes a call of the procedure named in it. A (possibly empty) list of actual parameters are passed, which correspond one-for-one with the formal parameters in the procedure's declaration:

```
procedure-statement =
        procedure-identifier [actual-parameter-list]
```

The actual parameters are evaluated in left-to-right order. Further details will be found in 8.2.3.

Examples:

```
open_customer_file
invert (a, b)
pack (a[row], 4*i, z)
```

3.1.1.4 GOTO statement

A GOTO statement causes control to be transferred to the place in the program text at which the label is defined, i.e. to the statement which is prefixed by the label (see 3.1).

```
goto-statement = "GOTO" label
```

The label may be in the current block or at any textually enclosing level.

Example:

```
GOTO 999
```

3.1.2 Structured statements

Structured statements are those which are composed of other statements. There are four kinds:

```
structured-statement =
     compound-statement ¦ conditional-statement ¦
     repetitive-statement ¦ with-statement
```

3.1.2.1 Compound statement

A compound statement is simply a sequence of statements bracketed by the delimiters BEGIN and END:

```
compound-statement = "BEGIN" statement-sequence "END"
statement-sequence = statement  { ";" statement }
```

The statements are executed in the order in which they are written.

Example:

```
BEGIN
  i := 0;  j := 1;
  k := i + j;
END
```

3.1.2.2 Conditional statements

The two sorts of conditional statement permit the selection of one from several alternative statements.

```
conditional-statement = if-statement ¦ case-statement
```

3.1.2.2.1 IF statement

```
if-statement = "IF" boolean-expression "THEN" statement
               [ "ELSE" statement ]
```

Here, boolean-expression is an expression (see 3.2) which is of type boolean, i.e. has the value either true or false. If, at run time, the value of the expression is true, the statement following THEN is executed and the statement following ELSE (if present) is skipped. If the value of the expression is false, the statement following THEN is skipped and (if there is an ELSE clause) the statement following ELSE is executed.

Since the alternatives are "statement"s, either or both may themselves be IF statements. For example:

```
IF i = 0 THEN
  IF j = i THEN reorder
  ELSE finish
```

Any possible ambiguity is resolved by the rule that an ELSE clause is always matched with the nearest unmatched THEN. The above statement is therefore equivalent to

```
IF i = 0 THEN
  BEGIN
    IF j = i THEN reorder
    ELSE finish
  END
```

as opposed to

```
IF i = 0 THEN
  BEGIN
    IF j = i THEN reorder
  END
ELSE finish
```

3.1.2.2.2 CASE statement

```
case-statement = "CASE" case-index "OF"
                 case-list-element { ";" case-list-element}
                 [ ";" "OTHERWISE" statement ] [ ";" ] "END"
case-index = expression
case-list-element = case-constant-list ":" statement
case-constant-list = case-constant  { "," case-constant }
case-constant = constant
```

Here, case-index is an ordinal-type expression which selects, at run time, which of a number of alternative statements is to be executed. The case-constants must all be distinct from one another, and be compatible with the type of the case-index.

If the value of the case-index matches one of the case-constants, the statement in whose case-constant-list that constant figures is executed (and all other statements are bypassed). If the value of the case-index does not match any of the case-constants, then what happens depends on whether an OTHERWISE clause is present or not; if so, the statement following OTHERWISE is executed (all other statements being bypassed); if not, a run-time error occurs.

Example:

```
CASE flag OF
  0:  interrupt := set;
  1:  interrupt := reset;
OTHERWISE error(134);
END
```

Note that, although they may resemble them (as in this example), case-constants are completely different from labels.

3.1.2.3 Repetitive statements

The three sorts of repetitive statement cause certain statement(s) to be executed repeatedly.

```
    repetitive-statement =
        repeat-statement | while-statement | for-statement
```

3.1.2.3.1 REPEAT statement

```
    repeat-statement =
        "REPEAT" statement-sequence "UNTIL" boolean-expression
    boolean-expression = expression
```

The sequence of statements bracketed by the delimiters REPEAT and UNTIL is repeatedly executed until the value of the boolean expression is true. The expression is evaluated after each execution of the statement-sequence. In particular, therefore, the sequence is always executed at least once.

Example:

```
    REPEAT
      j := 12 * i;
      i := succ(i);
    UNTIL j > i
```

3.1.2.3.2 WHILE statement

```
    while-statement = "WHILE" boolean-expression "DO" statement
```

While the value of the boolean expression is true, the statement is repeatedly executed. The expression is evaluated before each (potential) execution of the statement. In particular, therefore, the statement may not be executed at all.

Example:

```
    WHILE i <= j DO
      BEGIN
        j := 12 * i;
        i := succ(i);
      END
```

Note the difference in behaviour compared with the example in 3.1.2.3.1. If (e.g.) the initial values are i = 1 and j = 0, then the REPEAT loop will be executed precisely once, the WHILE loop not at all.

3.1.2.3.3 FOR statement

```
for-statement =
      "FOR" control-variable ":=" initial-value
      ( "TO" | "DOWNTO" ) final-value "DO" statement
control-variable = entire-variable
initial-value = expression
final-value = expression
entire-variable = variable-identifier
```

The statement after DO is repeatedly executed while a sequence of values is assigned to the control-variable. The latter must be an identifier which has been declared in the block immediately containing the FOR statement. The control-variable must be of ordinal-type, and the initial- and final-value expressions must be assignment-compatible with it.

When the FOR statement

```
FOR v := e1 TO e2 DO body
```

is executed, the sequence of events is as follows. The expressions e1 and e2 are evaluated, and if e1 > e2 then nothing remains to be done; otherwise, e1 is assigned to v, body is performed, v is compared with e2, and, for as long as it is not equal to e2, v is incremented and body is again executed.

When the FOR statement

```
FOR v := e1 DOWNTO e2 DO body
```

is executed, the sequence of events is as follows. The expressions e1 and e2 are evaluated, and if e1 < e2 then nothing remains to be done; otherwise, e1 is assigned to v, body is performed, v is compared with e2, and, for as long as it is not equal to e2, v is decremented and body is again executed.

Example:

```
FOR i := j TO 10 DO proc (i, j)
```

If j has the initial value 9, then this FOR loop has the same effect as the sequence of statements

```
i := 9;
proc (i, j);
i := succ(i);
proc (i, j)
```

If, on the other hand, the initial value of j is 12, the FOR loop simply does nothing.

3.1.2.4 WITH statement

```
with-statement = "WITH" record-variable-list "DO" statement
record-variable-list =
     record-variable  { "," record-variable }
record-variable = variable-access
```

As each record-variable in the list is encountered at compile-time, the compiler brings into scope all the field-identifiers of that record-type so that, for the duration of the with-statement, the fields can be referenced without having to select them by means of the usual "record-variable." prefix.

If selecting the record-variable involves array indexing and/or pointer dereferencing, these operations are performed, once and for all, before the component statement is executed.

Example:

```
WITH customer[custno] DO
  IF balance < 0 THEN
    BEGIN
      sendletter;
      creditworthy := false;
    END
```

Assuming (as always) appropriate type declarations, this WITH statement is equivalent to

```
IF customer[custno].balance < 0 THEN
  BEGIN
    sendletter;
    customer[custno].creditworthy := false;
  END
```

Besides being easier to read, the version using the WITH construct may well be compiled into better code.

3.2 Expressions

Expressions possess a value, at run time, of a particular type, and are composed of operands (such as a simple variable name) and operators (such as +). If an expression involves several different operators, the order in which the operations should be performed is determined by grouping them into four classes. In order of decreasing precedence, these are:

NOT
multiplying operators
adding operators
relational operators

Within any one class, operands are evaluated and operations are performed in left-to-right order. The precedence ordering can be overridden by the use of parentheses, ( ).

These ideas are reflected in the following formal definitions:

```
expression =
   simple-expression [relational-operator simple-expression]
simple-expression = [sign] term  { adding-operator term }
term = factor  { multiplying-operator factor }
relational-operator =
    "=" | "<>" | "<" | ">" | "<=" | ">=" | "IN"
adding-operator = "+" | "-" | "OR"
multiplying-operator = "*" | "/" | "DIV" | "MOD" | "AND"
```

Expressions, simple-expressions, terms and factors will be referred to generically as "operands". The various kinds of operators will be treated in section 3.2.2. The concept of "factor" remains to be defined, and this is the subject of the next section.

3.2.1 Factors

```
factor = variable-access | unsigned-constant |
         function-designator | set-constructor |
         "(" expression ")" | "NOT" factor
```

Of the 6 possible forms which factor can take, the last embodies the fact that, as mentioned in 3.2, NOT is the operator with the highest precedence, and the last-but-one reflects the possibility of overriding the usual operator precedence hierarchy by using parentheses. The first 4 forms will now be described.

3.2.1.1 Variable access

Because the Pascal language includes the concepts of records and pointers, as well as the more usual arrays and files, the selection of the data item to be referenced may involve a quite complicated sequence of operations, involving the symbols [ ], . and ^. For example, with suitable type declarations the construct

```
nextp^.sums[count].result
```

might refer just to a real variable. The following definitions formalise the rules for selecting a variable.

First, introduce a 5-fold subdivision:

```
variable-access = entire-variable | indexed-variable |
                  field-designator | referenced-variable |
                  buffer-variable
```

3.2.1.1.1 Entire variable

```
entire-variable = variable-identifier
variable-identifier = identifier
```

An entire-variable is therefore simply an identifier which denotes a variable declared in a VAR or COMMON declaration or in the formal parameter list of a procedure or function.

3.2.1.1.2 Indexed variable

```
indexed-variable =
    array-variable "["index-expression {"," index-expression}"]" |
    dynamic-string-variable "[" index-expression "]"
array-variable = variable-access
dynamic-string-variable = variable-access
index-expression = expression
```

An array-variable is a variable of array-type (see 6.1.2.1). The index-expression(s) must be assignment-compatible with the corresponding index-type(s) in the definition of the array-type.

Just as, when defining an array (see 6.1.2.1), the declaration

```
trans: ARRAY [1..9] OF ARRAY [char] OF char
```

is equivalent to

```
trans: ARRAY [1..9, char] OF char
```

so, when referencing an element of such a multidimensional array, the form

```
trans[3] [ch]
```

is equivalent to

```
trans[3, ch]
```

The same applies however many indexes the array has.

A dynamic-string-variable is a variable of dynamic-string-type (see 6.1.2). The index-expression must be integer-type, and have a value not greater than the maximum length of the dynamic-string-type, as specified in its declaration.

3.2.1.1.3 Field designator

```
field-designator = record-variable "." field-identifier
record-variable = variable-access
field-identifier = identifier
```

A record-variable is a variable of record-type, and the field-identifier must be one of the fields in the declaration of that record-type (see 6.1.2.2).

Examples:

```
persondetails.salary
nextdate^.time.second
```

3.2.1.1.4 Referenced variable

```
referenced-variable = pointer-variable "^"
pointer-variable = variable-access
```

A pointer-variable is a variable of pointer-type (see 6.1.3). The associated referenced-variable is a variable which must have been created dynamically in the heap by means of a call of the standard procedure new (see 8.3.2). The process of going from a pointer-variable to the referenced-variable by means of the ^ symbol is known as "dereferencing" a pointer.

Examples:

```
score[day]^
thisman^.father^.father^.son
```

In the second example, the relevant declarations would be something like

```
TYPE manptr = ^ manrec;
     manrec = RECORD
                father, son: manptr;
                {...}
              END;
VAR thisman: manptr;
```

3.2.1.1.5 Buffer variable

```
buffer-variable = file-variable "^"
file-variable = variable-access
```

A file-variable is a variable of file-type (see 6.1.2.4). The associated buffer-variable denotes the currently-accessible component of the file.

Example:

```
input^
```

3.2.1.2 Unsigned constant

The second possible form of "factor" (see 3.2.1) is an unsigned constant, of which there are 4 kinds:

```
unsigned-constant = unsigned-number | character-string |
                    constant-identifier | "NIL"
```

The definitions of unsigned-number and character-string are in 1.1.5 and 1.1.6, respectively. A constant-identifier is an identifier which has figured on the left-hand side of a CONST declaration (see section 5). The symbol NIL denotes the nil-value for pointer variables, and is assignment-compatible with all pointer-types.

3.2.1.3 Function designator

The third possible form of "factor" is a function call with a (possibly empty) list of actual parameters:

```
function-designator =
     function-identifier [ actual-parameter-list ]
```

For the definition of actual-parameter-list, see 8.2.3.

Examples:

```
max (yours, mine)
cos (i*x + j*y)
```

3.2.1.4 Set constructor

The final form for "factor" is a set-type value:

```
set-constructor =
     "[" [member-designator { "," member-designator} ] "]"
member-designator = expression [ ".." expression ]
```

The expression(s) must be ordinal-type, and must have ordinal values in the range

```
0  <=  ord(expression)   <=  2039
```

If the set-constructor involves more than one expression, the types of the expressions must be mutually compatible. If the expression(s) have type t, the set-constructor has the implicit type SET OF t.

The member-designator x..y represents the set of all values in the closed interval x to y; if x > y, it denotes no value at all.

[] denotes the empty set, which is assignment-compatible with every set-type.

Examples:

```
[you,me,him]
['A'..'Z', 'a'..'z']
```

3.2.2 Cperators

The operators introduced in 3.2 are best described under four headings: arithmetic, boolean, set and relational.

3.2.2.1 Arithmetic operators

Additional information on the precise effects of arithmetic operators on integer-type operands - in particular, those of subrange type - will be found in section 9.

3.2.2.1.1 +

If it is not preceded by an operand, + is a unary operator. It does not alter the value of the operand following it, which must be of integer-type, real-type or longreal-type.

If placed between operands of integer-type, real-type and/or longreal-type, + represents the usual binary operator of addition. If either operand is longreal, then the result is longreal, else, if either operand is real, then the result is real, else the result is integer-type. (The result is thus integer-type only if both operands are integer-type.)

Note that the symbol + is also used for the quite distinct operation of set union (see 3.2.2.3.1).

3.2.2.1.2 -

If not preceded by an operand, - is the unary operator of negation. It may only be applied to operands of integer-, real- or longreal-type, and produces a result of the same type.

If placed between two operands of integer-, real- and/or longreal-type, - represents the usual binary operation of subtraction. The result type is as described in 3.2.2.1.1.

Note that the symbol - is also used for the distinct operation of set difference (see 3.2.2.3.2).

3.2.2.1.3 *

If placed between two operands of integer-, real- and/or longreal-type, * represents multiplication. The result type is as described in 3.2.2.1.1.

Note that the symbol * is also used for set intersection (see 3.2.2.3.3).

3.2.2.1.4 /

The symbol / represents the operation of real division. The two operands may each be integer-, real- or longreal-type. If either operand is longreal, the result is longreal, otherwise, the result is real, any integer-type operand(s) being "floated" to real- or longreal-type (as appropriate) before the division is performed.

3.2.2.1.5 DIV

DIV is the operation of integer division with truncation. Both operands, and the result, are integer-type.

If i >= 0 and j > 0, then the value of i DIV j is such that

```
i - j  <  (i DIV j) * j  <= i
```

If j = 0, a run-time error occurs. If i and/or j are negative, the value of i DIV j is such that

```
abs(i DIV j)  =  abs(i) DIV abs(j)
```

and the sign of i DIV j is positive if i and j have the same signs and negative otherwise. For example:

```
 7 DIV  3  =  2
-7 DIV  3  = -2
 7 DIV -3  = -2
-7 DIV -3  =  2
```

3.2.2.1.6 MOD

MOD is the operation of taking the value of an integer modulo another - roughly, the remainder after division. Both operands, and the result, are integer-type.

If j <= 0, a run-time error occurs; otherwise, the value of i MOD j is that one out of the sequence of values

```
(i - (k*j)),  where k is any integer
```

which is such that

```
0  <=  i MOD j  <  j
```

For example:

```
 7 MOD  3  =  1
-7 MOD  3  =  2
```

3.2.2.2 Boolean operators

3.2.2.2.1 OR

OR is the logical inclusive "or" operator. Both operands, and the result, are of boolean type (i.e. take the values true or false).

3.2.2.2.2 AND

AND is the logical "and" operator. Both operands, and the result, are boolean.

3.2.2.2.3 NOT

NOT is the unary operator of logical negation. It is applied to an operand of boolean type, and produces the result true when applied to the value false, and vice versa.

3.2.2.3 Set operators

3.2.2.3.1 +

If placed between two operands of set-type (see 6.1.2.3), + stands for the operation of set union. The base types of the two operands must be compatible. The result has the type of the union of the two base types.

As an example, suppose there has been the type declaration

```
weekday = (Monday, Tuesday, Wednesday, Thursday, Friday)
```

then the value of the expression

```
[Monday..Wednesday] + [Thursday]
```

is equal to

```
[Monday..Thursday]
```

3.2.2.3.2 -

If placed between two operands of set-type, - represents the operation of set difference. The base types of the two operands must be compatible, and the result has the type which is the difference of the base types. For example, with the same declaration as in 3.2.2.3.1, the value of the expression

```
[Monday..Wednesday] - [Tuesday]
```

is equal to

```
[Monday; Wednesday]
```

3.2.2.3.3 *

If placed between two operands of set-type, * represents "set intersection". The base types of the operands must be compatible, and the result has the type which is the intersection of the two base types. For example, with the type declaration of 3.2.2.3.1, the value of the expression

```
[Monday..Wednesday] * [Thursday]
```

is the empty set, [].

3.2.2.4 Relational operators

3.2.2.4.1 = and <>

These operators are used to compare, for equality or otherwise, two operands of simple-, dynamic-string-, string-, pointer- or set-type. The result type is boolean (true or false).

The operands are of compatible types, or one operand is integer-type and the other real- or longreal-type - and this applies also to the operators in 3.2.2.4.2 and 3.2.2.4.3.

3.2.2.4.2 < and >

These operators are used to compare two compatible simple-, dynamic-string- or string-type operands. The result is boolean.

When two strings are compared, it is on the basis of their lexicographic ordering according to the ASCII character set (see Appendix D) - and the same applies to the operators in 3.2.2.4.3.

3.2.2.4.3 <= and >=

These operators may be used to compare two compatible simple-, dynamic-string-, string- or set-types. The result is boolean.

If s1 and s2 are two set-type operands, then s1 <= s2 is true if, and only if, the set s1 is a (not necessarily proper) subset of s2; and this expression has the same value as s2 >= s1. For example, with the type declaration as in 3.2.2.3.1, the expression

```
[Monday..Friday] >= [Tuesday]
```

is true.

3.2.2.4.4 IN

IN is used to determine whether an ordinal-type value (the left-hand operand) is a member of a set (the right-hand operand). If it is, the expression has the value true, otherwise, false. In particular, if the ordinal-type operand has an ordinal value outside the range of the base type of the set, then IN yields the value false. The type of the left-hand operand must be compatible with the base-type of the set.

As an example, with the type declaration of 3.2.2.3.1, the expression

```
Tuesday IN [Monday..Friday]
```

is true.

4 LABELS

Labels are unsigned decimal integers in the range 0..9999 (see 1.1.4). Their purpose is to enable the flow of control within a program to be abruptly altered, by GOTO statements.

Labels must be explicitly declared. They may then be defined and referenced.

4.1 Declaration of labels

In the overall layout of a block (see 2.3), the first declaration which may (optionally) be present is

```
    label-declaration-part = [ "LABEL" label {"," label} ";" ]
```

The label-declaration-part must contain all labels that are defined in the statement-part of that block. Conversely, all labels in the label-declaration-part must be defined (see 4.2) in the statement-part of the block.

Example:

```
        LABEL 1,        {for re-start}
              999;      {for error exit}
```

4.2 Definition of labels

A label is defined by being prefixed to a statement, as described in 3.1.

Example:

```
    999:  close(workfile)
```

4.3 Reference to labels

The only way a label can be referenced is in a GOTO statement, as described in 3.1.1.4.

Example:

```
        GOTO 999
```

5 CONST DECLARATIONS

A CONST declaration, which comes (after any label declarations) at the head of a block, is a means of giving names to constants:

```
constant-definition-part = ["CONST" constant-definition ";"
                            {constant-definition ";"} ]
constant-definition = constant-identifier "=" constant
constant-identifier = identifier
constant = [sign] (unsigned-number ¦ constant-identifier) ¦
           character-string
```

If the constant contains a sign (+ or -) with the form constant-identifier, then the constant-identifier must (previously) have been defined to represent an integer-, real- or longreal-type value.

Example:

```
CONST
  pi = 3.14159265358979 3D0;
  minuspi = - pi;
  message = 'Please repeat filename';
```

At a level enclosing the outer level of every program or segment, there is an implicit declaration of the predefined standard constant-identifier maxint. If written explicitly, this declaration would look like:

```
CONST
  maxint = 2147483647;
```

6 TYPE DEFINITIONS

Every variable and value in Pascal possesses a type, which may be one of the predefined standard types (see 6.1.1.1, 6.1.2 and 6.1.2.4) or be one created by the programmer. As well as dictating how much storage a variable occupies, the type determines which operations may be performed upon it and what effect those operations have.

The starting point for the formal definition of "type" is the syntactic object "type-denoter". It figures both in variable declarations, which are treated in section 7, and in type definitions, which are the subject of the present section.

The type definition part is the third (optional) component of a "block" (see 2.3).

```
    type-definition-part = [ "TYPE" type-definition ";"
                                    { type-definition ";"} ]
    type-definition = type-identifier "=" type-denoter
    type-identifier = identifier
```

6.1 Type denoters

```
    type-denoter = simple-type | structured-type | pointer-type
```

If, in a type-definition, the type-denoter is a simple-type, then the type-identifier is classified as a "simple-type-identifier". The concept of "x-type-identifier", for arbitrary "x", is defined in like manner.

The three kinds of type-denoter will be treated individually.

6.1.1 Simple types

```
    simple-type = ordinal-type | real-type | longreal-type
    ordinal-type = enumerated-type | subrange-type |
                   integer-type | boolean-type | char-type |
                   ordinal-type-identifier
```

Ordinal types have values which map onto a subset of the integer ordinal numbers. Real and longreal types have "floating-point" values.

If the item on the right-hand side of the type-definition is "ordinal-type-identifier", the definition simply introduces a synonym for an existing type-identifier.

Enumerated and subrange types will be defined in 6.1.1.2 and 6.1.1.3 respectively. The remaining possibilities are the five standard simple types.

6.1.1.1 Standard simple types

There are five of these: real, longreal, integer, boolean and char. The corresponding identifiers (real, etc.) are predeclared, at a level enclosing the outer level of every program or segment. A type is real-type if it is the identifier real or any type-identifier which has been defined to be a synonym, and similarly for the other standard types.

6.1.1.1.1 Real

real-type = "real"

These items take on real values, which are signed floating-point values whose magnitude may range from 5.9E-39 to 6.8E+38, and which are held internally to just over 7 decimal digits of precision.

Constants of this type are of the form

[sign] unsigned-real

where "unsigned-real" is as in 1.1.5.2.

6.1.1.1.2 Longreal

longreal-type = "longreal"

These items take on longreal values, which are signed floating-point values whose magnitude may range from 1.1D-308 to 3.6D+308, and which are held internally to just under 16 decimal digits of precision. It is worth noting that integral values of up to 9000000000000000 in magnitude are represented with complete precision; also that, provided the result is an integral value in this range, the operations of addition, subtraction, multiplication and division are performed with complete precision. Longreals can therefore be used for "whole-number" applications where the range of integer (-maxint..maxint) is insufficient.

Constants of this type are of the form

[sign] unsigned-longreal

where "unsigned-longreal" is as in 1.1.5.3.

The predefined type longreal is an extension to Standard Pascal.

6.1.1.1.3 Integer

    integer-type = "integer"

Integer-type items take values in the range -maxint..maxint, where maxint is defined in section 5. Constants of this type are of the form

    [sign] unsigned-integer

where "unsigned-integer" is as in 1.1.5.1.

6.1.1.1.4 Boolean

    boolean-type = "boolean"

Boolean items take the values false or true, which are predefined constant-identifiers, with ordinal values 0 and 1, respectively. It is as if there were the following type definition at a level enclosing the outermost level of every program or segment:

```
TYPE boolean = (false, true);
```

6.1.1.1.5 Char

    char-type = "char"

These take values which are any of the 128 characters of the ASCII character set (see Appendix D). The ordinal value of the character is its ASCII value, and so lies in the range 0..127.

6.1.1.2 Enumerated types

```
enumerated-type = "(" identifier-list ")"
identifier-list = identifier  { "," identifier}
```

An enumerated type determines an ordered set of values by enumerating the identifiers which denote those values. The ordinal value of each identifier is determined by its place in the list, the first (left-most) having ordinal value 0, the next 1, and so on. The list may contain at most 256 identifiers, corresponding to a maximum ordinal number of 255.

Examples:

```
(red, orange, yellow, green, blue, indigo, violet)
(false, true)
```

6.1.1.3 Subrange types

```
subrange-type = constant ".." constant
```

The constants must be of one ordinal-type, known as the "host" type of the subrange type. The two constants delimit the range of values which the subrange-type may take. The first constant must be less than or equal to the second.

Examples:

```
-128..127
'A'..'Z'
yellow..blue
```

6.1.2 Structured types

The second class of "type-denoter" (see 6.1) is composed of the structured types.

```
    structured-type = ["PACKED"] unpacked-structured-type |
                      dynamic-string-type |
                      structured-type-identifier
    unpacked-structured-type =
        array-type | record-type | set-type | file-type
    dynamic-string-type = "string" [ "[" constant "]" ]
```

A structured type is classified as array, record, set or file type according to the nature of the unpacked-structured-type in its declaration, i.e. without regard to whether PACKED is specified.

A structured type is classed as packed if, and only if, the token PACKED is explicitly present in its definition.

A packed structured type occupies the same storage as the corresponding unpacked type. However, some features of the language, notably those involving arrays, differ depending on whether or not a type is packed; see, in particular, 6.1.2.1 and 8.3.3.

A dynamic-string-type is declared using the predefined identifier "string", with an optional length-specifier. For example

```
string[32]
```

represents a dynamic-string-type which can hold a maximum of 32 characters (the actual length of the dynamic-string being a run-time variable quantity, as the name implies). If the length-specifier is omitted, then a default length of 80 characters is assumed. Dynamic-strings are an extension to Standard Pascal.

6.1.2.1 Array types

```
array-type = "ARRAY" "[" index-type {"," index-type} "]"
             "OF" type-denoter
index-type = ordinal-type
```

An array type consists of a fixed number of components, whose type is given by "type-denoter" in the above definition. Components may be of any type. The index-type specifies the range of values which the array index may take.

If the component type is itself an array-type, the definition

```
ARRAY [t1] OF ARRAY [t2] OF t
```

may be replaced by

```
ARRAY [t1, t2] OF t
```

and similarly for three or more indexes. The two notations are completely equivalent. If the second form is used, and the array type is packed, then the token PACKED is taken to apply to each and every array-type in the expanded (first) form of notation. For example:

```
PACKED ARRAY [0..9, red..violet] OF wavelength
```

is equivalent to

```
PACKED ARRAY [0..9] OF
        PACKED ARRAY [red..violet] OF wavelength
```

If t1 is a subrange of integer-type, with lower bound 1, then any type of the form

```
PACKED ARRAY [t1] OF char
```

is known as a string-type. The constants of string-type are the character-strings (see 1.1.6), the upper bound of the associated subrange type t1 being the length of the string. For example, 'ABC' is a constant of type PACKED ARRAY [1..3] OF char.

6.1.2.2 Record types

```
record-type = "RECORD" [field-list [";"] ] "END"
field-list = fixed-part [";" variant-part] | variant-part
fixed-part = record-section {";" record-section}
record-section = identifier-list ":" type-denoter
variant-part = "CASE" [tag-field ":"] tag-type "OF"
                variant {";" variant }
tag-field = identifier
tag-type = ordinal-type-identifier
variant = case-constant-list ":" "(" [field-list [";"] ] ")"
case-constant-list = case-constant {"," case-constant}
case-constant = constant
```

A record type consists of a fixed number of components, possibly of differing types. The record may consist of a fixed part only, or a "variant" part only, or a fixed part followed by a variant part.

The syntax of "fixed-part" is the same as that of "variable-declaration-sequence" (see section 7), the identifiers in "identifier-list" representing fields in the former and variables in the latter. However, the meanings are different. For instance, the occurrence of an identifier in a record-section causes no storage to be allocated: only when a variable of that record-type is declared is storage allocated for the fields which constitute that record. Furthermore, fields are referenced differently from variables (see 3.2.1.1.3).

If there is a variant part, the tag-type must be an ordinal-type. All the case-constants must be distinct, and be of a type compatible with the tag-type. The set of case-constant values must be equal to the set of values specified by the tag-type. (In particular, therefore, the tag-type cannot be "integer".)

Examples:

```
RECORD
  hours: 0..23;
  minutes, seconds: 0..59;
END

RECORD
  name: string24;
  age: 0..119;
  salary: integer;
  CASE female: boolean OF
    true:  (maidenname: string24);
    false: ()
END
```

6.1.2.3 Set types

```
set-type = "SET" "OF" ordinal-type
```

The ordinal-type defines the "base type" of the set. The set-type itself takes values in the powerset of the base type.

The base type may be either char, or an enumerated type, or any subrange of integer lying within the range 0 to 2039, or a subrange of any of these types.

Examples:

```
SET OF char
SET OF red..green
```

6.1.2.4 File types

```
file-type = "FILE" "OF" type-denoter
```

A file type represents a sequence of components all of which are of the same type, given by "type-denoter". Components may be of any type, except one having a file as a component.

There is one predefined standard file-type: text. Variables of type text are known as textfiles. Their components are of type char, but are, additionally, structured into lines. Lines are terminated by line-markers, the presence of which can be determined by calling the standard function eoln (see 8.3.1.1).

Examples:

```
FILE OF integer
FILE OF PACKED ARRAY [1..7] OF char
```

6.1.3 Pointer types

The third and final kind of "type-denoter" (see 6.1) is the pointer type.

```
pointer-type = "^" type-identifier | pointer-type-identifier
```

A pointer type has a value which points to a variable of an associated type, specified by "type-identifier" in the above definition. In addition, a pointer type can take the value NIL, which does not point to any variable.

Pointer values, and the variables to which they point, are created only by calls of the standard procedure new (see 8.3.2).

6.2 Type compatibility

At various points in the language definition, there are requirements that two types shall be "compatible", or that they shall be "assignment-compatible". These two terms will now be defined.

6.2.1 Compatible types

Two types, t1 and t2, are compatible if at least one of the following assertions holds:

(1) t1 and t2 are the same type.

(2) t1 is real-type and t2 is longreal-type, or vice versa.

(3) t1 is a subrange of t2, or vice versa, or t1 and t2 are both subranges of the same host type.

(4) t1 and t2 are set-types with compatible base types, and either both, or neither, is packed.

(5) t1 and t2 are string-type (see 6.1.2.1), with the same index-type.

(6) t1 is a dynamic-string-type (see 6.1.2) and t2 is a string-type or vice versa, or t1 and t2 are both dynamic-string-types.

6.2.2 Assignment-compatible types

A value of type t2 is assignment-compatible with the type t1 if at least one of the following assertions holds:

(1) t1 and t2 are the same type, and this is not a file-type nor a type containing a file-type component.

(2) t1 is real-type and t2 is integer-type or longreal-type.

(3) t1 is longreal-type and t2 is integer-type or real-type.

(4) t1 and t2 are compatible ordinal-types, and the value of type t2 is in the range of t1.

(5) t1 and t2 are compatible set-types, and all the members of the value of type t2 are in the range of the base type of t1.

(6) t1 and t2 are compatible string-types.

(7) t1 is a dynamic-string-type (see 6.1.2), and t2 is a string-type or a dynamic-string-type.

7 VARIABLE DECLARATIONS

Variables are declared in the variable-declaration-part of a block (see 2.3).

```
variable-declaration-part =
    ["COMMON" variable-declaration-sequence ";"]
    ["VAR" variable-declaration-sequence ";"]
variable-declaration-sequence =
      variable-declaration {";" variable-declaration}
variable-declaration = identifier-list ":" type-denoter
```

In a "variable-declaration", each identifier in the identifier-list names a variable of the type specified by the type-denoter.

The COMMON facility is a Pro Pascal extension. COMMON declarations can only be made at the outermost block level of a program or segment. Using COMMON, rather than VAR, causes the names of the variables to be accessible from other segments, and the same identifier declared in several segments represents one and the same variable. A COMMON variable exists throughout the execution of a program.

Variables declared in VAR declarations exist from the time the block in which they are declared is activated until the statement-part of the block is completed. They may be referenced in statements of that block and of any textually enclosed block.

For an account of how variables are referenced, see 3.2.1.1.

Example:

```
COMMON
  basearray: ARRAY [baserange] OF integer;
  errfile: errfiltype;

VAR
  student, teacher, parent: person;
  attendance: 0..maxnumber;
  promised,
  empty:  boolean;
  c1, c2: RECORD
            realpart, imagpart: real
          END;
  mark: (good, fair, indifferent);
```

8 PROCEDURES AND FUNCTIONS

A procedure is a self-contained part of a program which can be activated from elsewhere. A function is similarly independent, but differs in that it returns a value. Procedures and functions may be declared by the user according to his own requirements; there are also a number of so-called "standard" procedures and functions whose declarations are part of the definition of the language and which can be used at any point.

Much of this section applies equally to procedures and functions. References to "procedure" should be taken to include function unless stated otherwise.

8.1 Procedure and function declarations

Procedures and functions are declared in the fifth (optional) part of a block (see 2.3):

```
    procfunc-declaration-part = {procfunc-declaration ";" }
```

A procedure or function declaration introduces and names part of a program.

```
        procfunc-declaration =
                procfunc-heading ";" ( block | directive ) |
                procfunc-identification ";" block
```

The purpose of the second form is explained in 8.1.4. The form most commonly used is the first, consisting of a heading and a block separated by a semicolon.

8.1.1 Procedure and function heading

```
procfunc-heading = procedure-heading | function-heading
```

8.1.1.1 Procedure heading

```
procedure-heading = "PROCEDURE" procedure-identifier
                          [ formal-parameter-list ]
procedure-identifier = identifier
```

The heading names the procedure, and lists the formal parameters if any. Parameters are discussed in 8.1.1.3.

Example without parameters:

```
PROCEDURE listfile
```

8.1.1.2 Function heading

```
function-heading = "FUNCTION" function-identifier
        [ formal-parameter-list ] ":" result-type
function-identifier = identifier
result-type = simple-type-identifier |
              pointer-type-identifier
```

The distinction between function and procedure lies in the result returned by a function (and hence the method of activation). The type of the result is given in the heading.

Example without parameters:

```
FUNCTION rand: real
```

8.1.1.3 Parameters

The declaration of a procedure or function may include a list of formal parameters. Formal parameters are of four kinds.

```
formal-parameter-list = "(" formal-parameter-section
        { ";" formal-parameter-section } ")"
formal-parameter-section =
        value-parameter-specification |
        variable-parameter-specification |
        procedural-parameter-specification |
        functional-parameter-specification
```

When the procedure or function is invoked, an "actual parameter" is supplied to match each formal parameter in the declaration (see 8.2.3).

8.1.1.3.1 Value parameter

```
value-parameter-specification =
        identifier-list ":" type-identifier
```

Value parameters are local variables of the procedure, with the special property that initial values are supplied by the caller on activation.

The type may not be a file-type, nor a type containing a file-type component.

8.1.1.3.2 VAR parameter

```
variable-parameter-specification =
        "VAR" identifier-list ":" type-identifier
```

A VAR formal parameter is matched on activation with a variable-access (see 3.2.1.1) of identical type. Within the body of the procedure, a reference to the formal parameter, including assignment to it, becomes a reference to the actual variable. Thus a VAR parameter can be used to return results as well as to provide initial values.

Example:

```
PROCEDURE maxval (a, b: integer; VAR max: integer);
  {Returns larger of a and b}
  BEGIN
    IF a > b THEN max := a
    ELSE max := b;
  END;
```

8.1.1.3.3 Procedural and functional parameters

```
procedural-parameter-specification = procedure-heading
functional-parameter-specification = function-heading
```

Procedural and functional parameters allow a procedure (or function) to be substituted at the time of activation. A routine to print histograms, for instance, could be written with a functional parameter which when called returned the value for one column of the display.

8.1.2 Procedure or function block

Following the heading, there is either a directive (see 8.1.3 below) or a block. The block may contain any of the components of a program block (LABEL, CONST, TYPE, etc.) except COMMON variable declarations. All objects declared in a procedure block are "local" to the procedure, and are not accessible from outside. The ideas of locality and scope are discussed in section 2. In particular, the procedure block may contain procedure and function declarations, which are therefore "nested". A nested procedure may reference any local variables - including formal parameters - of the enclosing procedure.

Within a function block, there must be an assignment to the function identifier, and the value so assigned is returned as the result of the function. If there is more than one such assignment, the last is taken as the result.

Example (compare the example in 8.1.1.3.2):

```
FUNCTION max (a, b: integer): integer;
  {Returns larger of a and b}
  BEGIN
    IF a > b THEN max := a
    ELSE max := b;
  END;
```

8.1.3 Directives

A procedure heading, instead of introducing the complete declaration, may name (and make available for activation) a procedure whose full declaration is elsewhere.

```
directive = "FORWARD" ¦ "EXTERNAL"
```

The FORWARD directive indicates that the defining block is textually later in the same compilation-unit (see 8.1.4). The EXTERNAL directive (not part of Standard Pascal) indicates that the definition is in another compilation-unit. An EXTERNAL procedure may be in a separately-compiled Pascal segment, or may be in an assembler-coded module as described in 9.6 below. Names of EXTERNAL procedures are limited to 7 characters in the relocatable binary format, and must be distinct in these first 7 to avoid confusion during the link-edit process.

8.1.4 FORWARD declarations

A FORWARD declaration is introduced to enable a procedure to be referenced prior to its full definition. Each such declaration must be matched by a definition later in the source program (at the same level of nesting), the latter being associated with the original declaration by an identification with the same name. Continuing the formal definition from 8.1 :

```
procfunc-identification =
        "PROCEDURE" procedure-identifier ¦
        "FUNCTION" function-identifier
```

Example:

```
PROCEDURE fproc (fpb: boolean); FORWARD;

.
.

PROCEDURE fproc;
  BEGIN
    IF fpb THEN ...
    ELSE ...
  END;
```

Note that the parameters are not repeated.

8.2 Activation of procedures and functions

8.2.1 Activation of procedures

A procedure is activated by a procedure-statement quoting the procedure name (see 3.1.1.3). If the procedure heading included any formal parameters, corresponding actual parameters must be supplied (see 8.2.3 below).

8.2.2 Activation of functions

A function is activated when its name is used as a factor within an expression (see 3.2.1.3). During the execution of the function, a result value is assigned to the function name, and this result is returned to the expression. If the function has formal parameters, corresponding actual parameters must be supplied.

8.2.3 Actual parameters

```
actual-parameter-list =
     "(" actual-parameter { "," actual-parameter } ")"
actual-parameter = expression |
                   variable-access |
                   procedure-identifier |
                   function-identifier
```

8.2.3.1 Value parameters

The actual parameter corresponding to a value formal parameter is an expression, which must be assignment-compatible with the type of the formal. The current value is assigned to the formal parameter as its initial value.

The assignment-compatibility includes the implied type coercion of an integer actual parameter to real if the formal is of real type, and of an integer or real actual parameter to longreal if the formal is of longreal type.

Note that a value parameter may be of a structured type (e.g. a record). A local variable of the same type is allocated in the procedure, and the value of the actual is assigned to it (i.e. copied into it). A local copy may be needed by the procedure, but if the structure is a large one the effect on the size, and possibly also the execution time, of the program may be significant; in such cases, use of a VAR parameter (see next subsection) should be considered.

8.2.3.2 VAR parameters

The actual parameter corresponding to a variable (VAR) formal parameter is a variable-access, which must be of identical type to the formal. It may not be a tag-field, nor may it be a component of a PACKED type. Any reference to the formal parameter during the activation of the procedure is treated as a reference to this variable. If the selection of the variable involves indexing or pointer dereference, then such operations are carried out before the procedure block is activated.

Consider the example procedure maxval in 8.1.1.3.2, namely:

```
PROCEDURE maxval (a, b: integer; VAR max: integer);
  BEGIN
    IF a > b THEN max := a
    ELSE max := b;
  END;
```

Parameters a and b are value parameters, to be matched by actuals which are expressions. Parameter max is a VAR parameter, and must be matched by a variable. Possible calls of maxval are:

```
maxval (maxtotal, current, maxtotal)
maxval (float+50, 500, limit[item].flim)
```

The first has the effect of updating maxtotal if current is larger. Both maxtotal and flim must be of type integer; current and float may be of subrange types or integer.

8.2.3.3 Procedural (functional) parameters

The actual parameter corresponding to a procedural (functional) formal parameter is a procedure (function) identifier. The formal and actual must have compatible parameter-lists, and in the case of a function the result types must be identical.

Two parameter lists are compatible if (1) they contain the same number of parameters, and (2) corresponding entries match. Entries match if (1) they are both value parameters of identical type, or (2) they are both variable (VAR) parameters of identical type, or (3) they are both procedural parameters with compatible parameter lists, or (4) they are both functional parameters with compatible parameter lists and identical result types.

8.3 Standard procedures and functions

The declarations of the standard procedures and functions form part of the definition of Pascal. They need not be declared before use - indeed, though the names may be redeclared, the original definitions would then be lost.

Activation is as for user-declared procedures and functions (see 8.2).

Additional standard procedures for this implementation are defined in section 9.

8.3.1 Operations on files

This section describes the facilities of Standard Pascal relating to the use of files. Extra procedures associated with the operating system will be found in section 9.1.

The procedures are described in terms of a file variable gf, which may be of any file type, and a variable txf, which must be of type text.

8.3.1.1 eof and eoln

The "predicate" eof(gf) indicates when the file gf is at the end-of-file position. The parameter may be omitted, in which case the standard file input is assumed.

Examples:

```
IF eof (transactions) THEN summarise;
WHILE NOT eof DO ...
```

Similarly, eoln(txf) indicates when the textfile txf is at an end-of-line marker. In this condition, reading from the file obtains a space character.

It is an error to perform any input operation on a file if eof is true (even to test eoln), and the eof condition should therefore always be the first test.

8.3.1.2 reset and rewrite

The procedure reset(gf) prepares gf for input. If the file is empty, eof(gf) becomes true, otherwise eof(gf) becomes false and the buffer variable gf^ is positioned to the first element in the file.

The procedure rewrite(gf) prepares gf for output. The buffer variable gf^ is positioned to the first element, and eof(gf) becomes true. Any previous contents of the file are lost.

In general, any file must be initialised before input or output operations can be performed. Exceptions are the standard files "input" and "output" for which initialising is done before the Pascal program is entered (see 2.1).

8.3.1.3 get and put

These are the basic operations which advance the file buffer pointer to the next element (moving the "window"). They are in practice less frequently used than read and write.

> get(gf) obtains the next element of an input file. If the end of file has been reached, eof(gf) becomes true, and gf^ is undefined. It is an error to call get(gf) when eof(gf) is already true.
>
> put(gf) advances gf^ for an output file to point to the next element.

8.3.1.4 page

The procedure page(txf) causes a new page to be taken on an output textfile txf. The parameter may be omitted, in which case the standard file output is assumed.

8.3.1.5 read

In its basic form, read(gf, varbl) is equivalent to

```
BEGIN  varbl := gf^;  get(gf)  END
```

i.e. the current file element is assigned to varbl and the next element made accessible. The component type of gf must be assignment-compatible with the variable varbl.

There are some additional facilities of read. In the first place, it may have a list of parameters (rather than the single varbl) into which successive values are to be read. If gf is a text file, characters may be read into variables of type char, since this is the basic file element; but variables of integer, real or longreal type may also be included and conversion from the external character format is performed automatically. The external representation in these cases must conform to the layout of integer, real or longreal constants in a source program, any leading spaces or line separators being ignored.

The file parameter may be omitted, in which case the standard file input is assumed.

Example:

```
read (txf, itemnumber, quantity)
```

8.3.1.6 readln

This procedure advances a textfile to the beginning of the next line, making the first character available as the current file element (or sets eof if the end of the file has been reached). readln(txf) is equivalent to

```
BEGIN
  WHILE NOT eoln(txf) DO get(txf);
  get(txf);
END
```

Similarly to read, readln may also be called with one or more variable parameters, implying reading successive values from the current line before advancing to the next. readln(txf, v1, v2) is equivalent to

```
BEGIN  read(txf, v1, v2);  readln(txf)  END
```

8.3.1.7 write

The possible forms of write are essentially similar to the forms of read. There is a basic form, write(gf,expr) being equivalent to

```
BEGIN  gf^ := expr;  put(gf)  END
```

i.e. the value of expr is assigned to the buffer variable and the file advanced to the next element. The write operation takes an expression (rather than a variable), which for non-text files must be assignment-compatible with the component type of the file.

More than one element may be written with a single call of write, and if the file is a textfile then expressions of integer, real, longreal, dynamic-string, string, and boolean types (as well as char) may be included, and conversion is provided automatically.

Write operations to textfiles may optionally include specification of field widths in the output. The examples below show the effect for each expression type. The integer i contains the value 12345, and the real r contains 123.45.

statement	result	comment
write('X')	X	
write('X':5)	X	4 leading spaces
write('ABC')	ABC	
write('ABC':5)	ABC	2 leading spaces
write(i:6)	12345	1 leading space
write(i)	12345	default width = 11
write(i:1)	12345	left justified
write(r)	1.2344999E+02	default real format
write(r:10)	1.234E+02	
write(r:10:4)	123.4500	"fixed point" format

If no field width is specified, default widths are assumed, as follows:

type	default width
integer	11
real	14
longreal	24
boolean	6
char	1
string	declared length
dynamic-string	current actual length

The file parameter to write may be omitted, in which case the standard file output is implied.

8.3.1.8 writeln

The statement writeln(txf) outputs a line marker to the textfile txf. The parameter may be omitted, the standard file output being implied.

writeln(txf, e1, e2, e3) is equivalent to

```
        BEGIN  write(txf, e1, e2, e3);  writeln(txf)  END
```

i.e. the values are written, followed by a line marker.

8.3.2 new and dispose

These procedures are used to request space in the heap and to return it when no longer needed. In each case, p is a pointer-type variable, and t is the type with which the pointer is associated.

> new(p) allocates space for a variable of type t and sets p to point to it. The value of the new variable is undefined.
>
> dispose(p) returns the space occupied by p^. No further reference may be made to the variable.

If t is a record type with a variant part, the space may be requested for a particular variant. The tag-type value is included as an extra parameter: new(p,tag). If this form of new is used, the matching form dispose(p,tag) must be used to return the correct amount of space. If the variant part itself has a variant part, a tag value for that, too, may be specified, as a third parameter to new - and so on, if subvariants are even deeper nested.

Note 1. On dispose, the contents of the pointer variable p become obsolete, as do any copies made of it. This includes the implied copies generated when the variable is passed as an actual parameter, or included in a WITH statement.

Note 2. The form of new which includes a tag may result in a smaller allocation of heap space than other variants of the same record type. (Indeed, this is the object of using it.) The variant must therefore not be changed during execution, and operations which reference the whole ("entire") record are not permitted since some adjacent but quite independent occupant of the heap might be corrupted. These short records must be referenced by their individual fields.

8.3.3 pack and unpack

The procedures pack and unpack transfer one or more elements between an array of some type, and a PACKED array of the same type. If unp is the first array, and pkd is the other, then:

> unp must have at least as many elements as pkd;
>
> the operations include an index value i of array unp at which the transfer starts, and the value of i must leave "room" in the remainder of unp for all the elements of pkd.

The statement pack(unp,i,pkd) moves successive elements from unp to pkd, starting at unp[i] and continuing to the end of pkd. The statement unpack(pkd,unp,i) performs the transfer in the opposite direction.

8.3.4 trunc and round

The functions trunc and round perform conversion from real or longreal to integer type, truncating or rounding as the name implies. Each accepts a real or longreal argument and returns an integer result.

Examples:

```
trunc (5.2)      gives   5
trunc (5.7)      gives   5
trunc (-5.7)     gives  -5
round (5.2D0)    gives   5
round (5.7D0)    gives   6
round (-5.7D0)   gives  -6
```

8.3.5 ord and chr

The function ord converts an argument of any ordinal type (e.g. enumerated or char) to integer. Function chr takes an integer argument and returns the character value corresponding to it. The operations involved may sometimes be trivial, but the use of these functions to cross type boundaries contributes to program portability.

Example (v is in the range 0 to 15):

```
IF v < 10 THEN ch := chr (v + ord('0'))
ELSE ch := chr (v - 10 + ord('A'))
```

leaves in ch the hex character representing v.

8.3.6 succ and pred

These functions take an argument of an ordinal type, and return a result of the same type. succ(v) returns the value "1 after v" and pred(v) returns the value "1 before v". If v is integer, succ(v) is equivalent to v+1 and pred(v) to v-1.

If weekday is defined as

```
TYPE weekday = (Monday,Tuesday,Wednesday,Thursday,Friday)
```

then succ(Monday) is Tuesday and pred(Thursday) is Wednesday.

8.3.7 abs and sqr

These functions take an argument of integer, real or longreal type, and return a result of the same type. abs(x) returns the absolute value of x (i.e. -x if x is negative, +x otherwise). sqr(x) returns the square of x (i.e. x*x).

8.3.8 sqrt, sin, cos, exp, ln, arctan

These mathematical functions take an argument which may be integer, real or longreal. If the argument is integer or real, the result is real; if the argument is longreal, the result is longreal.

Function	Result	Illegal
sqrt(x)	non-negative square root	x < 0.0
sin(x)	sine of x (x in radians)	abs(x) > 32768.0 (x real) abs(x) > 4.3D9 (x longreal)
cos(x)	cosine x (x in radians)	abs(x) > 32768.0 (x real) abs(x) > 4.3D9 (x longreal)
exp(x)	exponential of x	x > 89.4 (x real) x > 710.4D0 (x longreal)
ln(x)	natural logarithm of x	x <= 0.0
arctan(x)	principal value (radians) of arctangent of x	

8.3.9 odd

The function odd(i) takes an integer argument i, returning true if the argument is an odd value (i.e. if i MOD 2 = 1) and false if it is an even value.

8.3.10 dynamic-string procedures and functions

Three procedures and four functions are provided for manipulating variables and expressions of dynamic-string type (see 6.1.2).

The examples in the following subsections assume the declaration:

```
VAR sv: string;
```

and that sv currently has the value 'PQRSTUV' (so that its dynamic length is 7).

8.3.10.1 concat(s1,s2, ..)

The function concat has two or more dynamic-string arguments, and returns a dynamic-string result consisting of the arguments concatenated together. For example:

```
sv := concat('A',sv,'YZ')
```

sets sv to the value 'APQRSTUVYZ'.

The arguments are expressions of dynamic-string type; in particular, therefore, they may be dynamic-string functions such as copy. It is an error if the combined length of the arguments exceeds 255 characters.

8.3.10.2 copy(stringval,index,count)

The function copy returns the dynamic-string value containing "count" characters, taken from "stringval" and starting at character-position "index". For example:

```
sv := copy(sv,4,3)
```

sets sv to 'STU'. Again, the parameter "stringval" is a general dynamic-string expression; and the other two parameters are in general integer expressions. It is an error if the substring defined by "index" and "count" extends beyond the current limits of "stringval".

8.3.10.3 insert(stringval,stringvar,index)

This procedure inserts "stringval" into "stringvar" at position "index", moving up any characters in higher index positions. The first parameter is a dynamic-string expression, the second a dynamic-string variable. "Index" may take any value up to the current length of "stringvar" plus 1 (i.e. insert may be used to append to the current contents), but it may not exceed this value. It is also an error if the resulting length exceeds the defined length of "stringvar".

As an example:

```
insert('XY',sv,5)
```

leaves sv holding 'PQRSXYTUV'.

8.3.10.4 delete(stringvar,index,count)

This procedure alters the contents of "stringvar" by deleting "count" characters, starting at position "index". For example:

```
delete(sv,4,2)
```

removes 'ST' from the original contents of sv, leaving 'PQRUV'. It is an error if the substring defined by index and count extends beyond the current limits of the contents of stringvar.

8.3.10.5 length(stringval)

The integer function length returns the number of characters in the dynamic-string "stringval". If the parameter stringval is a dynamic-string variable, the length is determined from its current contents, not from its nominal maximum length. "stringval" may in fact be any string expression, so that, for example

```
i := length(concat(s1,s2))
```

is quite permissible.

8.3.10.6 pos(substr,stringval)

The integer function pos searches "stringval" for the first occurrence of the substring "substr". If the latter does not occur within stringval, then pos returns the value zero; otherwise, it returns the index within "stringval" of the first matching character. For example:

```
i := pos('RS',sv)
```

sets i to 3; whereas

```
i := pos('X',sv)
```

sets i to 0. Both parameters may be general dynamic-string expressions.

8.3.10.7 str(intexp,stringvar)

This procedure converts the value of the integer expression "intexp" to decimal character form (as in writing to a textfile), and places the result in the dynamic-string variable "stringvar". It is an error if stringvar is not long enough to hold the decimal representation. (The maximum which is ever required is 11 characters.)

9 IMPLEMENTATION-DEPENDENT ASPECTS

9.1 Pascal files and CP/M

9.1.1 Declaration of files

The standard predeclared files input and output are always available. Any other file must be declared. Local files are permitted, also COMMON files in segmented programs.

9.1.2 File assignment

A variable comes into existence when the block in which it is declared is activated - for declarations at the outer program level this is on entry to the program. The contents of a file (i.e. the elements which make up the value of a file variable) are not held within the computer memory like other variables except when being referenced, but are kept on an external disc file or device. A connection must be set up between the Pascal program and the CP/M file (or device) to provide access to the contents, since the name given to the Pascal file variable is not in general the same as the CP/M filename. The variable is said to be "assigned" to the CP/M file, implying simply an association between the variable and a certain filename.

The connection is made, in the sense of an "open file" operation, when reset or rewrite is called, and at this time an input file must already exist. The corresponding "close" is performed automatically on exit from the block in which the file is declared. (A "close" procedure is provided for cases of abnormal exit, see below.)

When it comes into existence, each file variable is given a default assignment (CP/M filename with which it is associated) which may be changed by means of the procedure "assign", see below, before any reset or rewrite operation. A file which is simply a workfile, being written and read back within one program, need not be explicitly assigned; but any more permanent file should have a name, and the Pascal file be assigned to it. The default assignments of the standard files input and output are to the console (CP/M device CON:). All other files are defaulted to disc file names in the form PROTEM.nnn, where nnn is a number sequence starting at 001. Such files are erased on termination of the program unless renamed.

9.1.3 File formats

A Pascal text file (on disc) follows the conventions of ED and other CP/M processes: lines are terminated by c/r l/f, and the end-of-file is marked by ctrl Z. (The end-of-file marker is supplied automatically when a text file used for output is closed, and it causes eof to become true on input.)

Non-text files on disc are automatically blocked and unblocked if the file element size is less than half the size of a CP/M sector (i.e. if it is 63 bytes or less). Two bytes per sector are taken by the controlling software in this case. Larger file elements occupy an integral number of CP/M sectors, and in particular an element size of 128 bytes provides a one-to-one correspondence between elements and sectors. To read a binary CP/M file not produced by Pro Pascal, it should be declared as FILE OF sector (where sector has been declared ARRAY [0..127] OF byte), rather than FILE OF byte.

Random access facilities are available with non-text files on disc. For this purpose the elements in the file are numbered starting at zero.

9.1.4 Delayed input from files

The technique known as "lazy i/o" is employed on input to ensure sensible conversational use of the console. A get operation is not actually performed until the next reference is made to that file (by f^, eof(f), etc.). There is no effect on the operation of programs written according to the standard rules.

9.1.5 Additional standard procedures and functions

The following additional predeclared procedures are provided. Their use is explained individually below.

```
PROCEDURE assign (VAR f: genfile;  name: CPMname);
PROCEDURE seek (VAR f: ntfile;  elnumber: integer);
PROCEDURE update (VAR f: ntfile);
PROCEDURE close (VAR f: genfile);
PROCEDURE erase (VAR f: genfile);
FUNCTION fstat (name: CPMname): boolean;
FUNCTION checkfn (name: CPMname): boolean;
PROCEDURE append (VAR f: genfile);
PROCEDURE rename (VAR f: genfile; name: CPMname);
PROCEDURE ramfile (VAR f: text);
PROCEDURE echo (VAR f: text; onoff: boolean);
```

Here "genfile" implies a generalised file type, for which any valid Pascal file type may be substituted (as in the standard procedure reset, for example), and "ntfile" is any file type except text. "CPMname" is any string or dynamic-string type, the associated actual parameter being an expression representing a CP/M file or device name.

9.1.5.1 assign

A file is assigned to a CP/M disc file or device by a call of the procedure assign. The CPMname parameter may be the name of a disc file with or without drive specifier and filetype extension (e.g. 'X', 'GOOD.BYE' or 'B:READ.ME'), or a device name. The device names recognised are CON:, LST:, RDR: and PUN:, together with two pseudonyms KBD: and NUL: described below.

A textfile assigned to CON: and used for input works on a line-at-a-time basis. The operator may backspace and make corrections until c/r (Return) is given, when the complete line is made available to the program. This is the default arrangement with the standard file "input". Devices LST:, RDR:, and PUN: transfer a character at a time. The pseudo "device" KBD: gives access to the console keyboard without echo, and without hold-up if no key has been pressed (a null character being returned in this case). The KBD: facility is not available under MP/M, and should not be mixed with use of the cstat function described later. The other pseudo device NUL: accepts and throws away output - this may avoid the need for tests at several points in a program.

A non-text file may be assigned to a device provided that the element size is 1 byte. If a FILE OF char is assigned to CON: and reset, the transfers are carried out a character at a time, with echo, and holding until a key is pressed. This can be a useful alternative to the line input method used with text files.

Note that reset or rewrite (or update or append) must be called following assign before any other reference is made to the file.

9.1.5.2 seek

The seek operation provides random access to the elements of a non-text file by means of the element number. The file is regarded rather as an array, with "index" values starting at zero. To read random records, assign and reset are called, then seek(f,n) positions f^ to element number n. (Note that no "get" is needed, indeed get advances f^ to the next element.) It is not necessary to have prepared the file specially when writing it; a sequentially-written file can be read in this way. Following a seek, the standard get or read operations progress sequentially from the new position. To write random records, assign and update are called, followed by seek and put or write, as described in the next subsection.

9.1.5.3 update

Random access updating can be performed on a non-text file. After assign, procedure update(f) is called in place of reset/rewrite. The buffer variable f^ is thereby positioned at the first element of the file. (This is equivalent to the operation seek(f,0).) A seek operation may then be used to position the file window at the required element, and f^ can be used to examine and modify the contents. The standard procedure put(f) causes the modified element to be rewritten to disc, and advances the file window.

A file can be extended with the update facility, using seek to position to the first empty position and then writing sequentially. However, a file with "holes" in it cannot be processed reliably; it should be initialised by writing (e.g.) dummy records sequentially first.

9.1.5.4 close

Files are closed automatically at completion of execution of the program, or in the case of local files on normal exit from the procedure in which they are declared. The procedure "close" must be invoked for any output file if this normal exit path is not taken (because of chaining to another program, or a GOTO out of a procedure, for example).

9.1.5.5 erase

When a Pascal file has been assigned to a CP/M disc file, the file may be erased by calling the Pro Pascal procedure erase.

9.1.5.6 fstat

The boolean function fstat has as parameter a string (constant, variable, or expression) containing a CP/M filename, including optional drive specifier and filetype as for the second parameter of assign. Fstat returns the value true if the file exists, false if there is no such file or if the string is not a correct CP/M filename. That is, if fstat returns true, a Pascal file can be assigned to the same name and opened reset without error. For example

```
        fstat('LST:')          returns false (not a filename)
        fstat('A:B:X')         returns false (bad format)
        fstat('A:NIM.TXT')     returns true if a file NIM.TXT
                                    is present on drive A
                               false otherwise
```

9.1.5.7 checkfn

This is another boolean function, having a string as parameter. It returns the value true if the parameter is a correctly-formed CP/M filename, without any check as to whether the file exists or not.

9.1.5.8 append

To write additional data at the end of an existing sequential file, call assign followed by append (in place of rewrite). After append, the file is prepared for output (as by rewrite) but f^ is positioned just after any existing data. If the file does not in fact exist, append is equivalent to rewrite.

9.1.5.9 rename

This procedure has as parameters a file and a filename (CPMname). The file must already be connected to a disc file by an assign operation. The name of the disc file is changed to CPMname, which must be a correctly-formed disc file name. If a drive identifier is included, it must match the existing assignment; however, normal usage is to omit the drive specifier, implying the same. After rename, the file remains available for use by the program, but reset/rewrite/etc must be given before any further reading or writing can be done.

9.1.5.10 ramfile

This procedure has one parameter, which must be a textfile. The file is assigned to a workfile in memory (a "silicon file"), and rewrite must be called to prepare it for output. Data can then be written, with implied conversion of binary operands, the file reset and the data read back in character form; or alternatively, character data can be re-read with input conversions; or again, the file can simply be used to buffer text without the overhead of disc access. The length of the file is limited only by the amount of heap space available (cf. 9.3.1).

9.1.5.11 echo

The parameters are a file (which must be a textfile) and a boolean "onoff". The file must have been assigned to a discfile or ramfile, and set for output by rewrite. A call of echo with onoff true causes any subsequent output to the file to appear also on the console. To switch off the console echo, call echo again with onoff false.

9.2 Additional standard procedures

Section 9.1 above includes details of the additional procedures related to file handling. There are other additional standard procedures and functions, definitions of which are given below.

9.2.1 move

This procedure permits transfers of data without the type checking normally carried out on assignments. It is therefore to be used with care. The call must specify source (the first parameter) and destination (second parameter) for the transfer, and also the length (in bytes). Source and destination may be any variable references, length is an integer expression and may in principle be up to 64k bytes. Note that if source and destination areas overlap, it is relevant that the transfer is performed starting at the low-address end.

Examples:

```
move (sv, dv, 4);
move (srec.sarr[2], dc, 1);
move (srec.sf, darr[inx], sz);
```

9.2.2 chain

This procedure allows control to be passed from one program to another. A Pascal file must be declared, assigned to the CP/M .COM file, and reset. The chain procedure has just one parameter, the Pascal file (which can be of any type).

Note that chaining to another program does not provide the automatic closing of files which normally takes place on termination. Any output files must be closed explicitly (see under 9.1 above). See also putcomm and getcomm below.

Example:

```
assign (chainfile, 'NEXTPROG.COM');
reset (chainfile);
chain (chainfile);
```

9.2.3 putcomm

The procedure putcomm (put command) specifies a variable, typically a string or a record, which is to be passed to the next program after a chaining operation. The call of putcomm should immediately precede the call of chain. The variable is limited in size to 80 bytes.

Example:

```
        putcomm (interprog_record);
```

9.2.4 getcomm

The partner procedure getcomm (get command) is called on entry to the chainee program, to copy the command into one of its own variables. The layout of the command is arbitrary, though there must of course be agreement in specification between the two programs.

The area used for passing the command is in fact the CP/M default buffer, and getcomm can therefore be used to pick up the residue of any command line, quite apart from its use in chained programs. The residue is in the format of a dynamic string, and can be manipulated as a string after getcomm has moved it into a program variable. For example, if a program "compare" includes

```
        VAR  fnames: string[35];
                  ..
        BEGIN
             getcomm(fnames);
```

then following the command

```
        A>COMPARE S1.PAS, B:S2.PAS
```

the string fnames will be set to ' S1.PAS, B:S2.PAS'. (Note that all characters in the command line except the actual program name are transferred, including the space preceding S1.)

9.2.5 sizeof

The integer function sizeof is a notation for obtaining the storage occupied by a data type, for example a record. Because the value is in fact known to the compiler, it is simply introduced into the object code in the same way as for example a named constant. No run-time computation is involved. The parameter must be a type-identifier, and the value returned is in bytes, for example

```
sizeof (longreal)
```

yields the value 8.

9.2.6 addr

The function addr has a parameter which is a variable-access, and returns an integer result which is the machine address at which the variable is located.

9.2.7 peek

Function peek has an argument of type integer which is a machine address, and returns the value of the byte at that address, as an integer value in the range 0..255.

9.2.8 poke

Procedure poke has two parameters. The first is a machine address (as for function peek), the second is an integer expression which will be truncated if necessary and stored in the byte at that address.

9.3 Library facilities

These "library facilities" are routines provided in the standard library but not predeclared in the compiler. An appropriate declaration must be included in the program before one of these routines can be used.

9.3.1 memavail

The function memavail returns an integer value which is a measure of the amount of free space remaining (in bytes) between the heap and the stack. It is declared as

```
FUNCTION memavail: integer; EXTERNAL;
```

The value does not include space returned to the heap by dispose operations, and in general should be regarded as a useful guide rather than an exact figure.

9.3.2 rand

The function rand yields at each call a pseudo-random real value, uniformly distributed in the range 0.0 to 1.0. It is declared as

```
FUNCTION rand: real; EXTERNAL;
```

9.3.3 cstat

The function cstat returns a boolean value which is true if a key on the console has been pressed, false otherwise. No actual read operation takes place. It is declared as

```
FUNCTION cstat: boolean; EXTERNAL;
```

Note that if a key has been pressed, the cstat processing includes the checks made by CP/M for the special characters ctrl-C and ctrl-S, which have the normal consequences of aborting the program or suspending output, respectively.

9.3.4 dreset

Procedure dreset performs a reset of the disc system, so that a program can continue writing if a disc is changed during execution. The default drive is re-established as it was before the operation. The declaration is simply

```
PROCEDURE dreset; EXTERNAL;
```

9.3.5 ownerr

This procedure is provided to enable the user to perform his own exception handling, as an alternative to the normal reporting of run-time errors. The procedure ownerr installs a procedure nominated by the user as his error handler, which will then receive control in the event of any error arising. The handler is invoked for all types of error, but has the option of processing some and leaving the others to be reported at the console in the usual way. The handler must be written to the parameter specification shown below in the declaration of ownerr.

```
        PROCEDURE ownerr
                (PROCEDURE handler (errorletter: char;
                                    erroraddress: integer;
                                    VAR errorstring: string;
                                    fatal: boolean;
                                    VAR processed: boolean)
                );  EXTERNAL;
```

The procedure nominated as the error handler when ownerr is called must be at the outer level. (It can itself be an EXTERNAL, for example in a library.) Its parameters must agree with the list above, where the purpose of the first four is to provide "handler" with the information from the standard error message - letter, address, supplementary string (which may be empty), and fatal/recoverable flag. The fifth ("processed") is an inout parameter defaulted to false. If handler leaves this as it is, then on exit the normal report will be produced; if it is set to true, reporting will be skipped.

The handler routine may well refer to other variables of the program. For example, it may be useful to maintain a global variable (called "marker" say) which indicates to the handler the part of the program in which an exception occurred.

If the error is classed as recoverable (i.e. if "fatal" is false), then on normal exit from the handler execution will be resumed. (The normal report will be produced first if "processed" is false.) If the error is fatal, then on exit the program is terminated. However, the handler can use a GOTO to pass control back to the program body as a means of avoiding termination, in cases where recovery is feasible. Some types of error - for example stack overflow - may result in corruption of data, and any attempt at recovery from fatal errors must be carefully planned and tested.

The information in the VAR string parameter can be altered or extended, to a maximum of 30 characters, and the normal reporting process will display the amended string. (The limit of 30 is not checked, and exceeding it may have dramatic consequences.)

It is possible to call ownerr more than once in the same program, installing different exception handlers at different times.

The definition ensures that a handler with the correct parameter specification but which does nothing is "transparent", all error reports appearing in the normal way. Thus the error trapping can effectively be turned off.

9.3.6 iport

This routine must be declared as

```
FUNCTION iport (portno: integer): integer; EXTERNAL;
```

and when invoked reads a byte from the specified port, returning an integer value in the range 0..255.

9.3.7 oport

The declaration required is

```
PROCEDURE oport (portno,value: integer); EXTERNAL;
```

When called, "value" is output to Z80 port number "portno" (both expressions being truncated to byte width if necessary).

9.4 Storage allocation

9.4.1 Overall layout

Object programs and segments can in general contain requirements for the following kinds of storage.

Program code and constants.
Static data areas.
COMMON data blocks.

The link-edit process combines these areas, together with any library routines, into a loadable file (.COM). When the program is executed, the space remaining in the TPA is used dynamically for stack and heap.

```
 ________________________________________________________
|                   |         |        |                |
|  Loaded program   | Heap >  |        |   < Stack      |
|___________________|_________|________|________________|
```

Variables declared at the outer level of a program or segment are allocated static data space. Parameters and local variables of procedures are placed on the stack. In the link-edit operation there is an option to separate the code and data areas - see the description of the linker in Part III - though for simplicity the default is to combine them.

All allocation of data space is on a byte basis (i.e. no "slack" bytes are introduced), except for single-byte value parameters which occupy a word in the stack.

9.4.2 Formats of variables

Variables of "ordinal" types may be 1, 2, or 4 bytes in length.

1 byte: boolean, char, enumerated, and subranges of integer within -128..127 or within 0..255.

2 bytes: subranges of integer within -32768..32767 or within 0..65535 (but outside byte subranges). The normal low-high arrangement is followed.

4 bytes: integer, and subranges of integer outside word subrange. The bytes are arranged least-significant to most-significant in ascending addresses.

Real values occupy 4 bytes in a format corresponding to the proposed IEEE Standard. The 32 bits are made up as follows (from most to least significant):

1-bit sign
8-bit binary exponent, biassed by 127
23-bit mantissa, with an implied 1 in the most significant (24th) bit position

Longreal values occupy 8 bytes in the IEEE format:

1-bit sign
11-bit binary exponent, biassed by 1023
52-bit mantissa, with an implied 1 in the most significant (53rd) bit position

In both formats, the implied binary point is between the implied '1' bit and the most significant actual bit of the mantissa. Thus the value 1.0, for example, is represented by the following bit-patterns:

32-bit	3F800000H
64-bit	3FF0000000000000H

Pointers occupy 2 bytes.

Set variables occupy from 1 to 255 bytes, depending upon the upper limit of the range of the base type in the declaration (SET OF 0..7, for instance, requires 1 byte, while SET OF char occupies 16 bytes and SET OF 0..2039 is the maximum 255 bytes). Element 0 of a set is always present, and is represented in bit 0 of the first byte.

Arrays are arranged with the element having the lowest index value in the lowest address (the obvious way in this machine).

A variable of type string[n] occupies (n+1) bytes, the lowest-addressed byte containing the length of the string (a value in the range 0..255).

Record layouts are simply derived by placing the component fields in ascending addresses.

9.5 Object code for integer arithmetic

9.5.1 Modes of code generation

The previous section details the storage allocated to variables, and in particular to integers and subranges of integer. Use of suitable subrange declarations has several advantages: the program is more self-documenting, the compiler can allocate for each variable only the space it needs (so economising on data space) and generate the most appropriate machine operations (so economising on code size), and the range checking options can be applied when debugging.

Because the Z80 is essentially an 8-bit machine, most arithmetic operations can be performed between byte-length operands more efficiently than on 2-byte or 4-byte values. However, there are complications when the range of the operands approaches the limits of what can be held in a byte or a word, and Pro Pascal therefore allows a choice between two regimes of code generation for such quantities. The default regime avoids all potential overflow problems by extending the operands wherever necessary before the arithmetic is performed. This generally results in somewhat longer code, but is recommended except where space is at a premium.

The alternative regime is invoked by the "restricted width" (R) option at compile time. More economical code will generally be produced, but the programmer must be prepared to guard against any possible overflows. Details of the rules followed by the compiler are given below to assist in this process.

9.5.2 Restricted-width object code

Types which are subranges of integer are classified into byte, word and "long" (i.e. 4-byte), and in the case of byte and word length values into signed and unsigned. A subrange such as 1..10 which can be accommodated in a signed or unsigned byte is classed as signed, and similarly with word subranges such as 0..999. However, a literal (constant) in one of the overlapping ranges is treated where possible as having the same type as the operand with which it is being combined.

The following rules determine the result type of an operation, which except for multiply also determines the length at which the operation takes place. The notation used is sb (for signed byte), ub (for unsigned byte), sw, uw and i (integer).

9.5.2.1 Negate and abs

Operand type:	sb	ub	sw	uw	i
Result type:	sb	sw	sw	i	i

9.5.2.2 Mixed-length operands

If the two operands in an add, subtract, multiply or divide operation are of different lengths, the shorter is extended to the size of the longer before the operation takes place.

9.5.2.3 Add, subtract, divide and modulus

For byte and word add and subtract, in-line code instructions are produced. Other operations are performed out-of-line. The result type (which dictates the operation length) is determined from:

add, divide, modulus

	sb	ub	sw	uw	i
sb	sb	sb	sw	sw	i
ub	sb	ub	sw	uw	i
sw	sw	sw	sw	sw	i
uw	sw	uw	sw	uw	i
i	i	i	i	i	i

subtract

	sb	ub	sw	uw	i
sb	sb	sb	sw	sw	i
ub	sb	sw	sw	i	i
sw	sw	sw	sw	sw	i
uw	sw	i	sw	i	i
i	i	i	i	i	i

The cases of +1 and -1 are recognised and treated as increment and decrement.

9.5.2.4 Multiply

Multiply operations are performed out-of-line, with the exceptions noted below. The product of two signed bytes is a signed word, of two unsigned bytes is an unsigned word, and any other combinations produce a four-byte product.

The particular instance *1 is recognised, and has the effect of extending a byte or word value without calling any out-of-line routine. Similarly *2 causes byte or word to be extended and the first operand is then added to itself.

A further special case is byte*256, which produces a word result having the original byte as its more significant half.

9.5.2.5 Overflow

As noted earlier, the restricted range code can give rise to the possibility of overflow, and the programmer must make appropriate provision. (As a simple example, an addition of two signed bytes is performed at byte width, and the result classed as signed byte. If the original operands were 72 and 79 the sum would be out of range. One useful technique is to use *1 to force widening of one of the values, so that the operation is done at word width.)

To assist in the tracing of errors arising from overflow, extra code is introduced when the R option is used in conjunction with the range-checking options A or I. This code tests the overflow flag provided in the Z80 after in-line add and subtract operations.

The out-of-line add and subtract for 4-byte operands always check for overflow. The byte and word divide routines also check for the particular instances (-128 DIV -1) and (-32768 DIV -1). All these situations produce run-time error indications.

9.6 Interfacing to assembler

9.6.1 Use of assembly language

To make use of machine features not available through the Pascal language, for example interrupts, procedures may be written in assembly language and combined with the generated code during the link-edit process.

9.6.2 Choice of assembler

The Z80 version of Pro Pascal generates relocatable object code in Microsoft format. Assembly language segments may be processed by any assembler which generates this format, and linked with the other components of the program. In particular, Microsoft's Macro-80 assembler will be found satisfactory, supporting as it does the full range of Z80 instructions. The description of the linker in Part III gives details of certain constraints which must be taken into account. The .REL files produced by the Pro Pascal compiler will also be accepted by most other linkers which have been designed to handle the Microsoft relocatable format.

9.6.3 ENTRY/EXTERNAL linkage

The assembler-coded procedure must be declared within the Pascal program as EXTERNAL, for example

```
PROCEDURE asproc; EXTERNAL;
```

Note that the name is restricted in length to 6 characters by the assembler. Once declared, the procedure is called in the usual way.

In the assembly language module, the name is quoted in an ENTRY directive, or made global in some equivalent way. More than one procedure can be placed in the module. Return is made by RET instruction.

An outer-level Pascal procedure can be called from the assembler code. The procedure name is quoted in an EXT directive (or equivalent), and the procedure can then be CALLed. For the purpose of external access, all outer-level Pascal procedures are given ENTRY status in the relocatable form of the object program (unless the program is entirely self-contained), but note that the limitation to 6 characters applies if calls are to be made from assembler code.

9.6.4 COMMON data

Pascal variables which have been declared in COMMON can be referenced directly from assembler code. The compiler treats the variable names as COMMON block names, and the linker matches them with assembler COMMON statements. Again there is a limit of 6 characters on the length of these names.

As an example, the following might appear in a Pascal program:

```
TYPE  time = RECORD
                hours: 0..24;
                mins:  0..60;
                secs:  0..60;
             END;
COMMON
      letter: char;
      timer: time;
..
BEGIN
..
      letter := 'P';
      WITH timer DO
        write(hours:1,':',mins:1,':',secs:1);
```

In an assembler module linked with the above, letter and timer can be referenced as COMMON, by e.g.

```
        COMMON/LETTER/
PLETT:  DS      1
        COMMON/TIMER/
HOURS:  DS      1
MINS:   DS      1
SECS:   DS      1
```

The assembler declarations must describe the layout of the Pascal variables. Section 9.4 gives details of storage layout for different types.

The COMMON mechanism provides a simple and direct means of conveying data to or from an assembler-coded procedure.

9.6.5 Contents of IX

The generated Pascal code depends upon the contents of the IX register being unchanged on return from a procedure. Assembler-coded procedures must either leave IX undisturbed or save and restore it. The stack pointer must also be left after return at the same position as before the call (which is the natural result of returning from a simple procedure by means of a RET).

9.6.6 Parameters

When a procedure has parameters, the actuals are pushed on the stack prior to the call. The first parameter is pushed first, and so is furthest from the return link on entry to the procedure.

```
 ________________________________________
   |      |      |      |      |
   | Link |  p3  |  p2  |  p1  |
   |      |      |      |      |
 ________________________________________
   ^SP
```

On return, parameters as well as link must have been removed.

9.6.6.1 Value parameters

Values of simple type (see 6.1.1) occupy 1, 2, 4 or 8 bytes (see 9.4.2 for details). The corresponding number of bytes is pushed onto the stack, except that a 1-byte value is passed by pushing a pair of bytes (with the value in the high-addressed byte of the pair). In the case of 4- or 8-byte values, the high-order pair is pushed first, followed by the lower-order pair(s).

Set and dynamic-string values are passed adjusted to the length of the formal parameter.

Structure value parameters (arrays and records) are passed by address.

9.6.6.2 VAR parameters

In all cases, the address of the "first" (i.e. lowest addressed) byte is passed.

9.6.7 Function results

The result of a longreal function is returned in an 8-byte COMMON variable called $QACC. For any other type the result is returned in registers, determined by the length of the type:

1 byte	result returned in A
2 bytes	result returned in HL
4 bytes	result returned in HLBC (H most significant, C least)

9.7 Non-CP/M object programs

Pro Pascal can be used to generate object programs which are for the Z80 processor but independent of CP/M. This subsection provides some indications for doing this. It is assumed that the user has appropriate experience to code interfacing routines as assembler procedures.

9.7.1 Compiled object code

The generated object code (as distinct from the run-time library) contains no use of CP/M facilities. The code itself is pure, COMMON variables and those declared at the outer level of a program or segment are allocated in static memory, parameters and local variables of procedures are placed in the run-time stack.

9.7.2 Run-time library

The coding of the routines in the run-time library (in particular the integer and floating-point arithmetic and the set and string operations) makes them able to be used re-entrantly, i.e. as with compiled procedures the code is pure and the stack is used for all workspace requirements.

Exceptions to this general rule are the heap management (which keeps some base pointers in static storage), longreal arithmetic (which uses an 8-byte "software accumulator", $QACC, in static storage), and the file-handling routines. The latter are in any case oriented round CP/M devices and file operations.

9.7.3 RST instructions and alternate registers

If the "compact code" option is exercised, RST 2 to RST 4 are used as short calls to out-of-line sequences; this is the only application of the RST instructions. No use is made of the alternate registers (which are therefore available for interrupt handlers).

9.7.4 Library modules H1LIB and H2LIB

Source code for these modules is included in the distribution package for the benefit of users who wish to generate non-CP/M object programs, and they should be adapted to meet any special requirements.

H1LIB contains the routine $HINIT, which is called when a program is first entered, sets up the stack and heap, and initialises the standard files input and output. The console buffer is declared as a common block ($FLNB), and its size can be modified if required. The variables $DCLN, $DNLGT, $DSLFL and $DPRST relate to the option for maintaining source line numbers at run time (the last is actually in H2LIB). $MEMRY is filled in by the linker with the address of the first free location above the loaded program, which is the starting point for file areas and the heap.

H2LIB contains the routines for program termination and for run-time error reporting. There is also a routine $BDOS through which all CP/M calls are routed.

9.7.5 Preparation of non-CP/M programs

This section outlines at a general level some considerations related to the preparation of object programs to run in a non-CP/M environment. In the main, internal processing presents few problems - the difficulties tend to be in the area of input/output. This may well involve special devices which are not addressable via CP/M.

One approach is to dispense entirely with Pascal file handling, and to define procedures which interface with the devices and allow data to be transferred. The interfacing procedures are written in assembler, declared as EXTERNAL in the Pascal program, and included in the link-edit process. In some cases this is the only practical approach, and is always likely to result in a compact object program. For testing, either the special routines are included, or equivalents may be substituted (possibly coded in Pascal, and in any case presenting the same interface to the object program), to allow a CP/M device or file to represent the special one.

If this approach is [illegible] a version of H1LIB should be included in the final link-edit fro[illegible]ch have been removed the references to $INOUT and $FINIO. (The latter reference is to [illegible] library module which is redundant when Pascal files are not u[illegible]

However, if the application involves input and/or output of information formatted as text (i.e. character data separated into lines by c/r l/f), it may be worth retaining the standard files INPUT and OUTPUT. During initial testing, the CP/M console (or a disc file) can be substituted for any special device to be used in the eventual application, which may be a significant help. Also, the normal range of Pascal operations (get, put, read, write, etc.) is available, with any conversions that may be involved. Against the convenience must however be set the extra library modules that are needed to implement the facilities.

In Pro Pascal, a file variable consists of a pointer to an information block referred to as File Control Area or FCA. (The FCA is kept by the Pascal software, and is distinct from the FCB needed for CP/M disc files.) In bytes 6 and 7 of the FCA for a textfile is the address of a routine which will be called to obtain each input character, and in 8 and 9 the address of a routine which will be called to pass each output character. The character in both cases is kept in byte 23. In H1LIB, the call to $FINIO causes FCA's to be set up for the standard files INPUT and OUTPUT, after which bytes 6 and 7 of the FCA for INPUT contain the address of a routine to place the next character from CON: in byte 23 (using the "read console buffer" BDOS call), and similarly bytes 8 and 9 of the FCA for OUTPUT contain the address of a routine to take the character in byte 23 and transfer it to CON: (using the "console output" BDOS call). H1LIB can be modified to replace the default entry addresses with the addresses of routines which read from or write to non-CP/M devices, observing the convention that byte 23 of the FCA is used as a buffer. The higher-level file handling software will then invoke the supplied routines whenever get, put, read or write operations address these files.

INDEX

In this index, word-symbols are distinguished by the use of capital letters (as in BEGIN). Standard names (such as reset, maxint) and words from the formal syntax (such as array-type, identifier) are distinguished by not having an initial capital letter.

PART III - PRO PASCAL OPERATION

1 INSTALLATION DETAILS

1.1 Hardware requirements

The hardware required to run the Pro Pascal compiler is a computer with Z80 processor, a console, and at least 200K bytes of disc storage. CP/M (version 1.4 or later) or CDOS is needed, and memory (RAM) of 52K bytes. (To be exact, the CP/M Transient Program Area including CCP must be at least 44K bytes.)

The minimum requirement for object programs is a computer with Z80 processor and console, running CP/M (version 1.4 or later) or CDOS. Other requirements are dependent on the program. It is possible to generate object programs that do not use CP/M at all - see Part II, section 9.7.

1.2 Delivery and installation

The Pro Pascal software is delivered on a disc, or discs, containing the following files:

PROPAS.COM	Compiler (Pass 1)
PROPAS2.COM	Compiler (Pass 2)
PROPAS3.COM	Compiler (pass 3)
PROPAS.ERR	Compile-time error messages
PASLIB.REL	Run-time library
PROLINK.COM	Linker program
PROLIB.COM	Librarian program
XREF.COM	Cross-reference generator
PCONFIG.COM	Configuration program
H1LIB.MAC,H2LIB.MAC	Library module sources - see Part II, 9.7.4

together with a few example source program (.PAS) files. If there are any special comments relating to the software, for instance descriptions of extra files included on the disc, they are placed in a file called READ.ME. If this file is present, consult it before installing the software.

Before use, a Pascal system disc (or discs) should be established. Normally, the .COM, .ERR and .REL files are added to a disc containing at least a CP/M system, a text editor (ED, for example), and PIP. During compilation and linking, this disc is in the default drive. There must be sufficient space on it for the compiler's work files, which typically occupy together as much space as the Pascal source file (e.g. 16K bytes for a fairly large program). It is possible to hold Pascal programs on the same disc, but to avoid the need for periodical clearing out they are usually kept on another disc (or discs) mounted on a different drive.

If the available disc capacity makes the normal arrangement unworkable, the PCONFIG program should be used, to distribute the supplied files and/or the compiler's work files onto disc drives other than the default drive (see section 8).

In the descriptions which follow, it is assumed that a Pro Pascal system disc is on the default drive A, and a Pascal source program disc on drive B.

2 SIMPLE COMPILE AND LINK

To prove that the software is installed correctly, copy the sample program RESULTS.PAS to the Pascal program disc. Set up the system disc in A and program disc in B. After the prompt A>, type

```
PROPAS B:RESULTS
```

and the following output is generated on the console:

```
A>PROPAS B:RESULTS

Pro Pascal Compiler - Version zz 2.1
Copyright (C) 1982 Prospero Software
Serial No:    nnnn

Pass 1

Pass 2

Pass 3

Name: RESULTS     Lines:   58
                  Code:   657
                  Data:    20

Pro Pascal Linker  -  Version zz 1.6
Copyright (C) 1982 Prospero Software
Serial No:    nnnn

   Linking:
   B:RESULTS.REL
   PASLIB.REL

   Data:             0103      1F81
   Program:          0103      1F81
   Start address:    011B

   Executable file: B:RESULTS.COM
```

Note that the linking operation is entered automatically. The result of linking is the file RESULTS.COM on disc B.

The command STAT B:RESULTS.* shows three files: the source .PAS, the object program .COM, and also the "relocatable" version .REL which is generated by the compiler and read by the linker.

Program RESULTS is designed to read from the file "input" which is by default assigned to CON:, and write to file "listing" which is assigned to RESULTS.PRN. Execute the program, and type a few lines such as those in the sample below, ending with ctrl Z on a new line. The contents of RESULTS.PRN can then be displayed.

```
A>B:RESULTS
105  76 65 47 59 81 69
108  55 58 68 67 42
110  67 39 72 73 65 71
114  70 78 76 82
119  69 43 38 46 39
121  52 47 32 43 48 55 72
122  74 56 65 42 88 69
124  46 63 72 42 59 60
127  50 51 9 48 67
^Z

A>TYPE RESULTS.PRN
  105       76   65   47   59   81   69                397  66.17
  108       55   58   68   67   42                     290  58.00
  110       67   39   72   73   65   71                387  64.50
  114       70   78   76   82                          306  76.50
  119       69   43   38   46   39                     235  47.00
  121       52   47   32   43   48   55   72           349  49.86
  122       74   56   65   42   88   69                394  65.67
  124       46   63   72   42   59   60                342  57.00
  127       50   51    9   48   67                     225  45.00

    Winner is number  105  with average 66.17

A>
```

To demonstrate the complete process, use ED to create a new source program H.PAS (source files should normally be given the .PAS suffix). As a variation, this could be put on the default disc. The program consists of

```
PROGRAM h(output);
  BEGIN
    writeln (output, 'Hello');
  END.
```

When this has been set up, type

```
PROPAS H
```

File H.COM is generated, and when executed will display Hello on the console.

Of course, the Pro Pascal software is designed to be able to handle much larger programs than this, but the operation is not necessarily any more complicated.

The extra facilities of the compiler and linker are explained in the next two sections.

3 OPERATION OF THE COMPILER

The compiler processes a source file, containing a Pascal PROGRAM or SEGMENT, and converts this into a binary output file in Microsoft relocatable format.

The previous section has described the simple form of operation of the compiler, in which all the compile-time options are left at their default (or "off") values. In this section, the various options and messages are explained.

3.1 Form of invocation

The PROPAS command may be given with, or without, a source filename: the "one-line" and "conversational" modes of invocation, respectively.

3.1.1 The one-line command

When a source filename is specified, it may optionally be followed on the command line by the character "/" together with one or more letters, as in:

```
PROPAS B:RESULTS/NAP
```

Each letter stands for a particular compile-time option (see section 3.2). The letters may be run together, as in this example, or may be separated by spaces, commas, or further / characters. It makes no difference whether they are in upper or lower case.

3.1.2 Conversational mode

If no file name is given in the first command line (i.e. just PROPAS), a conversational mode of operation is entered. The first request is for the name of the source file, the response being e.g. B:RESULTS, terminated by Return. If no filename extension is given, the default .PAS is supplied automatically.

There is then a series of questions relating to compile-time options. There are three possible responses to each question in the list:

Y or y	to select the option
N or n	to reject the option and go to the next
.	to terminate the list.

(Note that any characters other than these are ignored, and that it is not necessary to press Return for the reply to be accepted.) When the list is terminated, or the end is reached by Y and N responses, the compilation process begins.

3.2 Compile-time options

The various compile-time options are described in the following sub-sections, the associated letter for invoking the option in a one-line command being given in brackets in each sub-heading. The default setting for each option is "off", or N, when the software is shipped, but this can be altered by runnning the configuration program (see 8.2).

3.2.1 Source listing (L)

A listing of the source can be generated as a by-product of compilation. Each line is preceded by its line number within the file and by the relative address of the start of that line within the object code. The listing is output to a file with the name of the source and extension .PRN, on the same drive as the source. After the compilation it may be listed or typed as desired.

Specifying this option also causes the current source line to be displayed as part of any compile-time error messages.

3.2.2 Compact code (C)

If the compact code option is invoked, the compiler substitutes shorter (but somewhat slower) alternatives for certain object code sequences. The amount of difference this will make depends on the nature of the program, but might typically be in the region of 10% saving in compiled code.. Use of the option would only be recommended for large programs.

If the program is segmented, and the option is to be used for any of the segments, it is essential that it be used also when compiling the main program. (The latter sets up the mechanism on initial entry.)

Compact code uses RST instructions 2, 3, and 4.

3.2.3 Restricted-width arithmetic (R)

The effect of specifying this option is fully described in Part II, section 9.5. If in doubt as to its meaning, simply leave it in its default (i.e. off) state. As with the /C option, its use would only be recommended for large programs and/or for those in which speed is critical (processing of real-time events, perhaps).

3.2.4 Source line numbers (N)

This option instructs the compiler to insert extra code into the object program to maintain during execution a record of the source line number corresponding to the code currently being obeyed. This line number will be displayed in the event of any run-time error, and if it is within a procedure then also the line number at the point of call, and the sequence of calls is then followed right back to the main program. (For further details, see under section 5.)

3.2.5 Range checks (I and A)

Range checks can be made on values of enumerated, subrange or set type. The checks are carried out just before a value is to be used, and take into account the specific destination. The checks can be invoked separately for index bounds (I) and for assignments (A) (which includes value parameters).

Extra code is generated, checking the expression value against the precise stated bounds of index, variable, or formal parameter. (Constant values are checked at compile time.) Range checks can be valuable in the early stages of program testing to pick up errors such as use of a variable before any value has been given to it, and may usefully be kept longer for index bounds.

3.2.6 Checks on pointers (P)

This option causes checks to be inserted at each "pointer dereference" (p^) in the program. The address resulting is tested to be reasonable as the position of an object in the heap. The check will certainly detect any attempted use of a pointer which has been set to NIL, and has a good chance of picking up cases where no value has been assigned.

3.2.7 Hold before REL output (H)

The hold option is provided mainly to ease the compile/link process when limited disc space is available. The compilation process is suspended before generation of the .REL file to allow the disc to be changed; in this way, .REL files can be kept separately from the .PAS sources.

3.2.8 Accept only strict Standard Pascal (S)

When this option is invoked, the compiler disables use of the non-standard features of Pro Pascal, namely:

segmentation (SEGMENT/COMMON/EXTERNAL),
OTHERWISE clause in CASE statement,
additional predefined types, procedures or functions,
compiler directives (source file insertion, page throw),
hexadecimal or longreal constants,
underscore characters within identifiers.

If a program is to be transferred to a different Pascal implementation, this check will pick out any points which may call for attention.

3.2.9 Double precision floating-point constants (D)

With this option, the compiler is instructed to treat every unsigned-real constant (see Part II, 1.1.5.2) as the corresponding unsigned-longreal constant (see Part II, 1.1.5.3). That is, each of the following constants

```
1.2    12e-1   0.1200E+1
```

is treated just as if it had been written as

```
1.2D0
```

A possible use for this option is when it is desired to run an existing program (using reals) in the extended-precision mode which longreal provides. Provided care is taken in handling any EXTERNAL interface which involves reals, a simple one-line edit to include the TYPE declaration

```
real = longreal
```

together with recompilation using the D option, may be all that is needed.

3.2.10 Console output to .LOG file (G)

When this option is specified, the messages output by the compiler to the console during compilation are written also to a disc file. The name of the file is the same as that of the source, with the extension .LOG. This can be a useful facility, both for inspection of compile-time errors and for recording the compilation status of each source program (code size, etc.).

3.3 Compiler messages

When the compilation process proper begins, messages are output to the console to report progress and any irregularities. At any stage, the compilation can be interrupted by pressing any key, and then either resumed or aborted.

3.3.1 Normal messages

In the main, these are self-explanatory. The start of each of the three passes is indicated. If the compiler activation was conversational, each pass also reports the amount of free space remaining, from which may be judged the limit on source program size that can be handled in any particular Transient Program Area.

If any use is made of the source file insertion facility (see Part II, 1.2.1.1), then the line numbers at which the included files are started and ended are written to the console. The first column of line-numbers represents the overall line numbering, as used by the compiler to number lines in compile-time error messages and in the listing (.PRN) file. Each subsequent column of line-numbers, up to the maximum allowed depth of 4 "current" source files, represents the line number within the source file at which an "event" occurred, namely, at which reading of another source insert file started (the filename is printed against the fictitious line number 0) or ended (a fictitious line number one greater than the actual last line of the file is printed).

On completion of compilation, the number of source lines, and the sizes of the code and data areas generated, are reported. These are all decimal values. The data size does not include any COMMON variables. If there are no EXTERNAL references, and the one-line form of command was used, the linker is entered immediately; in the conversational mode, the question

```
Link ? (Y/N)
```

is output.

3.3.2 Error messages

If the source filename is illegal, or the file does not exist, a

```
? filename
```

message is produced, and the compilation process is terminated (in the one-line command mode) or the request for a file name is repeated (in the conversational mode).

Errors in the source program may be detected during any of the three passes, though the majority generally appear in pass 2. The format in each case is source line number and error code, followed by an explanatory sentence if the file PROPAS.ERR is present. In Appendix B is a list of the error codes, with somewhat fuller descriptions where appropriate.

If the source listing option (L) is in force, the line in error is displayed immediately after the error number(s) for that line.

A single error, as the programmer sees it, may sometimes give rise to a number of reports. An obvious instance is a missing declaration, which will be signalled at each reference. It is also possible for one error to have a "cascading" effect. Large error counts should, therefore, not be taken at face value.

The other possible problems which may arise during compilation are connected with running out of space, either in memory or on disc (e.g. insufficient room for the .REL file). Such events give rise to error messages in the normal run-time error format (see under section 5). In particular, an error S or H signifies that the compiler's work space has become full; the only remedy for this is to reduce the size of the source program, perhaps by splitting it into several segments.

4 OPERATION OF THE LINKER

The linker processes a sequence of one or more files in Microsoft relocatable format and combines them into an executable .COM file. There is essentially no limit to the size of the executable file, whatever the size of work area available, since it is built up using a paging (rather than memory-resident) technique.

The linker allocates storage to items in the order in which they are encountered in the input file(s), in increasing memory addresses (except for ASEG items, which must of course be loaded at specific addresses). There is an option, however, to request that "code" and "data" items be separated from one another. In either case, the Microsoft convention is adhered to, whereby the address of the byte beyond the top of the data area is stored by the linker in the word named $MEMRY (if such a symbol is encountered).

All addresses printed, or requested, by the linker are in hexadecimal (without a trailing 'H' character).

There is both a "one-line" and a "conversational" mode of operation, the latter making available to the user a number of link-time options.

4.1 <u>Form of invocation</u>

The simple mode of executing the linker may be illustrated by returning to the first example in section 2. To repeat just the link part of the process, type

```
PROLINK B:RESULTS,PASLIB/S
```

which will cause the relocatable-binary file RESULTS.REL on disc B to be read, the Pascal library file PASLIB.REL to be scanned (selecting only the required modules), and the executable file RESULTS.COM to be generated. The console output is as follows:

```
Pro Pascal Linker  -  Version zz 1.6
Copyright (C) 1982 Prospero Software
Serial No:    nnnn

   Linking:
   B:RESULTS.REL
   PASLIB.REL

   Data:            0103     1F81
   Program:         0103     1F81
   Start address:   011B

   Executable file: B:RESULTS.COM
```

The command line following the program name (PROLINK) must consist of one or more filenames, separated by commas. Any of the filenames may be followed by the two characters /S, to indicate that a "selective" scan of that file is to be made, i.e. that only those modules are to be incorporated that have been referenced by previously-encountered modules. (In this context, a "module" is the result of compiling one PROGRAM or SEGMENT, or the output from one execution of Macro-80.) In the case of the Pascal library, the selective scan mode should always be specified, i.e. PASLIB/S.

If, on the other hand, no filenames at all are supplied on the command line, i.e. all that is typed is

    PROLINK

then the conversational mode of operation is entered. There is a series of questions relating to link-time options, and then an invitation to input one or more lines containing filename(s). This is described in detail in section 4.2.

In either mode, if filenames with no extension are given, the extension .REL is supplied automatically by the linker. The name of the executable file is constructed by appending the extension .COM to the name of the first input (relocatable) file. The .COM file is designed to load at address 0100, in the normal way. Locations 0100 thru 0102 will contain a jump instruction (created by the linker) to the start of the program.

4.2 Link-time options

The first question is

    Separate program and data areas ? (Y/N)

Here, "program" refers to the Z80 instructions generated by the compiler, which are read-only (i.e. not altered at execution time), and "data" refers to everything else: variables declared at the outermost level of a program or segment, COMMON variables, data areas of Assembler-coded library routines, etc. This distinction between "program" and "data" is the same as that between CSEG, on the one hand, and DSEG and COMMON on the other, in Assemblers such as Microsoft's Macro-80.

If the reply to the question is N or n, storage is allocated by the linker using sequentially increasing addresses starting at 0103, with no distinction between the two types of item.

If the reply is Y or y, there is an invitation to input values for the start addresses of two areas, one for "program" and one for "data". The values are to be input in hexadecimal. If just c/r (Return) is given, a default value of 0103 is used. Neither value should be less than 0100. If either is less than 0103, it is important to realise that the linker will always put a jump instruction in locations 0100 thru 0102, as described in section 4.1.

The second question is

```
Map ? (Y/N)
```

If the reply is N (or n), no storage map of the executable program will be produced.

If the reply is Y (or y), there is a further question

```
Map $names too ? (Y/N)
```

By replying N or n, the listing of the (often rather numerous) library names beginning with $ may be suppressed.

The storage map is output to the console when the linking process is complete. It is in two parts; first, the names and absolute addresses of any COMMON blocks; second, the same for all External/Entry names, that is: Pascal outer-level procedures or functions together with ENTRY names from Assembler-coded modules. As with all console output, the CP/M "ctrl P" facility can be used to obtain a hard-copy print of the map on the listing device.

After these questions, there is a prompt to input any number of lines containing filename(s). Just as was described in section 4.1, the filenames must be separated by commas, and any of them may be followed by /S.

When all input filenames have been specified, respond to the prompt

```
Filename(s) -
```

with a full stop (.) character, or with just c/r (Return). The linking process will then be completed.

4.3 Linker messages

During linking, messages are output to the console to indicate progress, to flag any errors and/or to report a successful conclusion.

4.3.1 Normal messages

As it starts to process each input file, the linker writes the full filename to the console.

At the end of the link, the start and end of the program and data areas, and the start-of-execution address, are reported. The name of the executable (.COM) file is also printed.

4.3.2 Error messages

If an input file cannot be found (perhaps because its name has been misspelt), the linker reports this fact and invites more filename(s).

If a character other than 'S' is supplied after '/', the linker reports this, and ignores the spurious character.

A further situation leading to an error message is when all the input files have been processed and yet there are still EXTERNAL references outstanding for which no corresponding Pascal procedure or function (or Assembler ENTRY name) has been encountered. This may be because a .REL filename has been inadvertently omitted from the list. The message

```
Unsatisfied external(s):
```

is printed on the console, followed by a list of the unmatched names. Then there is the question

```
Terminate ? (Y/N)
```

If the response is Y or y, the linking process will be brought to a conclusion. In particular, a .COM file will be produced which is normal except that any location containing a reference to a missing routine will not contain a sensible value. Caution should therefore be exercised if execution of the program is attempted.

If the response is N (or n), the linking process will resume with a renewed invitation to input filename(s); in this way, the link can be completed successfully.

If an error situation other than the above is detected, the link operation is aborted immediately, after outputting a message. No usable .COM file will have been produced. The messages in this category are the following.

Text	Meaning
Not enough memory	The linker needs a work area of at least 7K bytes.
Multiply-defined entry	The input files contain more than one Pascal procedure/function or Assembler ENTRY name with the same (up to 7 characters) spelling. The name is printed before the message.
Load address below 0100	The .COM file would, if completed and executed, involve items being loaded at memory addresses below 0100. This is probably due to incorrectly chosen values for the start addresses of the program and/ or data areas (see 4.2).
Program exceeds 64K	The .COM file would, if completed and executed, involve items at memory addresses above FFFF.
Prog overlaps data area) Data overlaps prog area)	These two errors can only occur if the answer to Separate program and data areas ? was Y or y. The remedy is to specify a more suitable start address for either or both of the areas.
ASEG overlays prog/data	An absolute load address (from use of Macro-80's ASEG directive, for example) falls within the current code and/or data area(s).
Attempt to extend COMMON	If a COMMON variable is declared in more than one segment, then either all declarations should be the same or the variant with the largest size must be the first one encountered by the linker.

Extern +- not supported	The linker is not designed to handle link items which correspond to Assembler constructs like EXT NAME DW NAME+3 ; or NAME-2, etc.
Extension not supported	The linker is not designed to handle "extension link items", which are reserved in Microsoft relocatable format for use by certain compilers (currently, only COBOL).
COMMON not selected) Nonexistent COMMON block) Chaining error) Relocation error) End of file encountered)	These five - and the previous - errors should never occur. If they do, one of the input files is most probably not a relocatable file at all: check on this. If the error persists, it possibly indicates a linker or compiler malfunction.

5 OPERATION OF OBJECT PROGRAMS

The operation of an object program under CP/M is determined very much by the program itself. Programs are run by typing the name of the .COM file after the normal prompt, and return to CP/M on completion (unless chaining to another program). Default assignments of files are as described in 9.1.2 of Part II, the standard files input and output in particular being defaulted to the console. The program may include assignments to specific named CP/M files (their names being expressed in the program text as string constants) or it may start by requesting entry of a filename from the keyboard, as for instance the compiler does.

The only aspect of program operation not determined from the program itself arises if an error is detected by the run-time software.

5.1 Run-time errors

Errors may be detected in a number of situations: file handling, dynamic space management, arithmetic operations, and so on. In some cases they may be found by the checking code incorporated by one of the compile-time options (see 3.2 above). In all cases a report is made on the console, giving error type - identified by a code letter - and the absolute location (in hex):

```
Error x at address aaaa
```

A list of the run-time error codes is given in Appendix C. In some cases, additional information is given, preceding the standard message.

If the option to carry source line numbers into the object program has been selected, the standard message is followed by the line number information. The first number gives the error location, and if this is within a procedure it is followed by the line number at the point of call, the sequence of calls being followed back to the program level.

In a segmented program, the line number information is ambiguous (i.e. segments are not distinguished in the display), and procedure names must be used to resolve any queries. If the segments were not all compiled with the line numbers option, nothing at all is printed for calls in "unnumbered" segments.

Finally, many classes of error allow continuation, and this choice is offered as a console option with (Y/N) response.

6 OPERATION OF THE LIBRARIAN

The purpose of the Librarian utility program is to administer files which are in Microsoft relocatable format - such as those produced by the Pro Pascal or Pro Fortran compilers or by the Macro-80 assembler. Individual modules may be extracted, and/or files may be merged together into libraries. A number of report options are also available.

6.1 Form of invocation

As with the other programs in this package, there is both a "one-line" and a "conversational" mode of operation. All the options are available in either mode.

6.1.1 The one-line command

The command line following the program name (PROLIB) must be constructed as follows. First must come the name of the "library" file. This may optionally be followed by the character "/" together with one or more letters, as in:

```
PROLIB  B:RESULTS/MX
```

Each letter stands for a particular option regulating the report(s) that are produced by the librarian (see 6.2). The letters may be run together, as in this example, or may be separated by spaces or further / characters; it makes no difference whether they are in upper or lower case.

A one-line command of the above form indicates a "read-only" operation on the library file: the file must already exist, and the purpose of the PROLIB execution is solely to list certain information about this relocatable file.

Alternatively, the library filename (and any option letters) may be followed by an "=" sign and one or more input filenames, separated by commas, as in:

```
PROLIB  NEWLIB/M = MOD1, MOD2
```

A one-line command of this form indicates a "create" mode of operation: if the library file already exists it will be overwritten, and the purpose of the PROLIB execution is to combine the input filenames into a new library with the given name. (The librarian actually creates the file, in the first place, with an extension of .$$$, and only renames this to the required library filename on successful completion of processing. For this reason, it is perfectly possible to include the library filename (assuming the file already exists) as one of the input files, although this may not be considered good data processing practice.)

Any of the input filenames may be immediately followed by a "module selector" (see 6.3).

If no filename extension is given (whether for the library or the component input file names), the extension .REL is supplied automatically by the librarian.

6.1.2 Conversational mode

If, on the other hand, no filenames at all are supplied on the command line, i.e. all that is typed is

```
PROLIB
```

then the conversational mode of operation is entered.

The first request is for the library filename. There is then a series of questions relating to the report options (cf. 6.2). Reply Y (or y) to select the option, otherwise N (or n). The final question is whether or not to create a new library with the given filename. If the answer is affirmative, the librarian repeatedly issues an invitation to input a line containing filename(s). The filenames are entered just as for the one-line mode of operation, that is, they must be separated by commas and each may be followed by a "module selector". To terminate this process, respond to the prompt

```
Input filename(s) -
```

with a full-stop (.) character, or with just c/r (Return) on its own.

If no filename extension is given (whether for the library or the input filenames), the extension .REL is supplied automatically.

6.2 Report options

Whether or not in the "create" mode of execution, the librarian can be requested to produce a report describing the library file. (If in the create mode, the report will reflect the contents of the library file on completion of processing.)

The various report options are described in the following sub-sections. Each sub-heading contains (in brackets) the associated letter which must be written after the library filename in the one-line form of execution in order to invoke the option.

6.2.1 Module listing (M)

A report is produced which gives, for each module in the library file (in order of occurrence within the file), the name of the module, all ENTRY names defined and EXTERNAL names referenced within it, together with the sizes of code, data and any common blocks which it contains.

6.2.2 Cross-reference listing (X)

The report consists of two parts. The first part gives, for each ENTRY/EXTERNAL name in the library file (in alphabetical order), the name of the module in which it is defined (i.e. is an ENTRY name) plus the names of all modules in which it is referenced (i.e. is an EXTERNAL name).

The second part is a listing of all common blocks (in alphabetical order) together with the names of the modules which reference them.

6.2.3 Unsatisfied references listing (U)

This report is concerned with the requirement imposed by PROLINK (along with many other linkers) that, for a library which is to be "selectively" searched (/S option), the component modules must be ordered in such a way that, if module A contains an external reference to an entry point in module B, then module B must follow module A in the library file (cf. 4.1 above). The report lists all EXTERNAL names (in alphabetical order) which do not obey this rule, either because they are defined in an earlier module or because they are not defined at all.

6.2.4 Suppress $names (N)

(This option is only meaningful if at least one of M, U or X has been selected.)

In order to avoid conflict with user-defined names, most ENTRY and common-block names in the Pascal library begin with $. Since they are rather numerous, it can on occasion be desirable to suppress them. By specifying this option, no name beginning with $ will appear in the report(s).

The default is that all names, including those beginning with $, are listed.

6.2.5 Listings to disc (D)

(This option is only meaningful if at least one of M, U or X has been selected.)

The default destination for reports is the console. (Of course, as with all console output, the CP/M "ctrl P" facility can be used to obtain a hard-copy print-out on the listing device.)

If this option is chosen, the reports are written instead to a disc file. The file is given the same name as the library file, but with the extension .PRN.

6.3 Module selection

In the "create" mode of operation (only), the user may specify that only some of the modules in an input file are selected. (The default is to select all modules from each file.) For this purpose, two kinds of "selector" are provided.

The first kind is the "selective scan" of an input file, and is specified by following the filename with the two characters /S: only those modules that have been referenced by previously selected modules will be incorporated into the output library file (and so into any reports).

Example: FNAME/S

The second kind is by "module enumeration", and is specified by following the filename with the character [, then a collection of module names, and finally the character]. This "collection" of module names is to be written as a list of names, separated by commas; optionally, in place of a module name, the list can contain, at any point, two names separated by ".." (i.e. name1..name2), signifying "all modules from name1 to name2 inclusive".

Example: FNAME1 [MOD1, MOD4..MOD8, MOD16]

A particular filename can be followed by at most one of these two kinds of selector.

An example of an input line containing all the above features is:

```
FNAME1, FNAME2 [M6], FNAME3 [MOD3..MOD9], LIBNAME/S
```

When reading an input file in the "selective scan" (/S) mode, the librarian and the linker adopt identical selection criteria. Use may be made of this to obtain an analysis of the composition of a fully-linked .COM file. Suppose, for example, one wishes to know which modules from PASLIB are needed by the RESULTS program referred to in section 2 above. The one-line command

```
PROLIB TEMP/M=B:RESULTS,PASLIB/S
```

will produce a report giving details of RESULTS.REL itself and of all the contributory modules from PASLIB. At the same time, the relocatable file TEMP.REL is produced. This file can subsequently be made the object of other reports, as in:

```
PROLIB TEMP/X
```

It can even be converted into an identical copy of RESULTS.COM by:

```
PROLINK TEMP
```

6.4 Librarian messages

6.4.1 Normal messages

If in the "create" mode, when it starts to process each input file the librarian writes the full filename to the console.

6.4.2 Error messages

If an input file cannot be found (perhaps because its name has been misspelt), the librarian reports this fact and invites more filename(s).

If a character other than 'S' is supplied after '/' following an input filename (i.e. where a "selective scan" directive is anticipated), the librarian reports this error and ignores the incorrect character.

If an error situation other than these is detected, execution is aborted immediately, after outputting a message to the console. The messages in this category are the following.

Text	Meaning
Command line improperly terminated	In the one-line command mode, the library filename and switches have been read, followed by a character other than "=".
Illegal module-selection syntax	The rules given in 6.3 have been broken (in particular, ".." must have a module name on either side of it, and "[" must have a matching "]" on the same line).
End of file encountered	If this error occurs, the most probable explanation is that an input file is not in Microsoft relocatable format at all.

7 CROSS-REFERENCE GENERATOR

A cross-reference generator XREF is provided as part of the Pro Pascal package. It is a very useful facility when developing programs of any size, and is tailored to the Pro Pascal syntax (extra reserved words, hexadecimal and longreal constants, underscore within identifiers, source file insertion).

As with the compiler and linker, there is both a "one-line" and a "conversational" mode of invocation.

7.1 Simple cross-reference

This mode is chosen by entering the source filename with the command, e.g.

```
XREF B:RESULTS
```

The source file is read, and the cross-reference listing is output direct to device LST: with a page width of approx. 100 characters.

7.2 Additional facilities

When XREF is activated without a source filename in the command line, the conversational mode is entered. The source filename is first requested. As with the compiler (and the simple mode described in 7.1) the .PAS suffix is supplied if none is given in the input. The destination for the listing is specified next. It may be directed to LST:, as in the simple mode, or to a disc file, or indeed to CON:. If output is to a disc filename, and no suffix is given, the suffix .XRF is supplied. If simply c/r (Return) is replied, the default destination (LST:) is assumed. Finally, the linewidth can be be set. Replying to this question with just c/r (Return) gives the default width of 100; otherwise, narrower or wider listings can be selected. (The value determines the point on the line at which a new entry will not be started, but a new line taken; it is possible that the actual listing width may exceed the value entered by up to one entry, i.e. 4 or 5 characters.)

7.3 Messages

At any time before production of the report commences, the program can be interrupted by pressing a key, and then either resumed or aborted.

If use is made of the source file insertion facility (see Part II, 1.2.1.1), then the line numbers at which the included files are started and ended are written to the console, as described in 3.3.1 above. The line numbers in the cross-reference listing are the "overall" line numbers, obtained after insertion of the source files.

If the source file is not a correct Pascal program, XREF may produce error messages, using certain of the numbers listed in Appendix B.

8 THE CONFIGURATION PROGRAM

A program (PCONFIG.COM) is provided to enable the user to "tailor" the Pro Pascal compiler to his own requirements. Two aspects of the system may be changed: the disc drives on which the various files known to the compiler are to reside, and the default values of the compile-time options.

Before executing PCONFIG, make sure that the copy of PROPAS.COM which you wish to amend is on the default disc drive. Then type

    PCONFIG

The program will invite you to make changes to the disc drive configuration and/or to the compile-time option defaults.

8.1 Disc drives

The question-and-answer session is largely self-explanatory. The only point worth expanding on is that, to specify that file(s) shall reside on the default drive (rather than A:, or whatever), you may reply either by pressing the space bar or by a c/r (Return) on its own.

(The software is shipped in a state which corresponds to replying with a space or Return to all the questions.)

8.2 Compile-time options

Again, the question-and-answer session is self-explanatory. The program runs through all the compile-time options, in the same order as when the compiler is executed in "conversational" mode (see 3.1.2), and for each, you are requested to respond with Y (or y), meaning that the default setting of that option in all future compilations is to be "yes" or "on"; or N (or n), meaning that the default for that option is to be "no"/"off". (All other characters typed in response to the question are ignored.)

(The software is shipped in a state which corresponds to replying N or n to all the questions.)

A LANGUAGE SUMMARY

A.1 NOTATION

The notation used throughout this manual for the Pascal syntax is summarised in the following table:

Notation	Meaning
=	is defined to be
¦	alternatively
[x]	zero or one instance of x
{x}	zero or more instances of x
(x¦y¦...¦z)	grouping: any one of x, y, ..., z
"xyz"	the terminal symbol xyz
lower-case-name	a non-terminal symbol

(For increased readability, the non-terminal symbols are often hyphenated.)

In this appendix, the nature of the source file which is input to the compiler (the 'compilation-unit') is viewed from two complementary aspects: the lexical (or bottom-up) and syntactic (or top-down). These views merge at about the level of the 'token'. The division of the remainder of this appendix into two subsections is designed to mirror these two viewpoints.

The definitions in each of the following subsections are grouped and ordered according to their 'level'. At the first level comes, in each case, the definition of the 'compilation-unit': the only concept given two (complementary) definitions. The definition of any other concept is to be found on the next level to that in which the concept first appears. The definition level is printed at the left margin.

Taken together, subsections A.2 and A.3 contain one, and only one, definition for every nonterminal symbol. The only exceptions - apart from 'compilation-unit' - are 'end-of-line', which is self-explanatory, and 'ASCII-character', which stands for any one of the 128 characters in the ASCII set (these are enumerated in Appendix D).

Except within a 'character-string', there is no distinction in meaning between the upper- and lower-case versions of any letter.

A.2 LEXICAL ASPECTS

```
1  compilation-unit = {token {separator} }

2  token = special-symbol | directive | identifier | label | unsigned-number |
           character-string

   separator = space | end-of-line | comment

3  special-symbol = "+" | "-" | "*" | "/" | "=" | "<" | ">" | "[" | "]" | "." |
                    "," | ":" | ";" | "^" | "(" | ")" | "<>" | "<=" | ">=" |
                    ":=" | ".." | word-symbol

   directive = "FORWARD" | "EXTERNAL"

   identifier = letter { (letter | digit | underscore) }

   label = digit-sequence

   unsigned-number = unsigned-integer | unsigned-real | unsigned-longreal

   character-string = "'" string-element {string-element} "'"

   space = " "

   comment = ("{" | "(*") character-sequence ("}" | "*)")
```

```
4 word-symbol = "AND" | "ARRAY" | "BEGIN" | "CASE" | "COMMON" | "CONST" |
                "DIV" | "DO" | "DOWNTO" | "ELSE" | "END" | "FILE" | "FOR" |
                "FUNCTION" | "GOTO" | "IF" | "IN" | "LABEL" | "MOD" | "NIL" |
                "NOT" | "OF" | "OR" | "OTHERWISE" | "PACKED" | "PROCEDURE" |
                "PROGRAM" | "RECORD" | "REPEAT" | "SEGMENT" | "SET" |
                "THEN" | "TO" | "TYPE" | "UNTIL" | "VAR" | "WHILE" | "WITH"

  letter = "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" |
           "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" |
           "w" | "x" | "y" | "z"

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

  underscore = "_"

  digit-sequence = digit {digit}

  unsigned-integer = decimal-integer | hexadecimal-integer

  unsigned-real = decimal-integer "." digit-sequence ["E" scale-factor] |
                  decimal-integer "E" scale-factor

  unsigned-longreal = decimal-integer ["." digit-sequence] "D" scale-factor

  string-element = string-character | apostrophe-image

  character-sequence = {ASCII-character}
```

```
5  decimal-integer = digit-sequence
   hexadecimal-integer = digit {hexdigit} "H"
   scale-factor = [sign] decimal-integer
   string-character = ASCII-character
   apostrophe-image = "''"

6  hexdigit = digit | "A" | "B" | "C" | "D" | "E" | "F"
   sign = "+" | "-"
```

A.3 SYNTACTIC ASPECTS

```
1  compilation-unit = program | segment

2  program = program-heading ";" block "."

   segment = segment-heading ";" segment-declarations "BEGIN" "END" "."

3  program-heading = "PROGRAM" identifier ["("global-parameter-list")"]

   segment-heading = "SEGMENT" identifier ["("global-parameter-list")"]

   block = label-declaration-part
           constant-definition-part
           type-definition-part
           variable-declaration-part
           procfunc-declaration-part
           statement-part

   segment-declarations = constant-definition-part
                          type-definition-part
                          variable-declaration-part
                          procfunc-declaration-part
```

```
4  global-parameter-list = identifier-list

   label-declaration-part = ["LABEL" label {"," label} ";"]

   constant-definition-part = ["CONST" constant-definition ";"
                               {constant-definition ";"} ]

   type-definition-part = ["TYPE" type-definition ";" {type-definition ";"} ]

   variable-declaration-part = ["COMMON" variable-declaration-sequence ";"]
                               ["VAR" variable-declaration-sequence ";"]

   procfunc-declaration-part = {procfunc-declaration ";"}

   statement-part = compound-statement

5  identifier-list = identifier {"," identifier}

   constant-definition = constant-identifier "=" constant

   type-definition = type-identifier "=" type-denoter

   variable-declaration-sequence = variable-declaration {";" variable-declaration}

   procfunc-declaration = procfunc-heading ";" (block | directive) |
                          procfunc-identification ";" block

   compound-statement = "BEGIN" statement-sequence "END"
```

```
6  constant-identifier = identifier

   constant = [sign] (unsigned-number | constant-identifier) | character-string

   type-identifier = identifier

   type-denoter = simple-type | structured-type | pointer-type

   variable-declaration = identifier-list ":" type-denoter

   procfunc-heading = procedure-heading | function-heading

   procfunc-identification = "PROCEDURE" procedure-identifier |
                             "FUNCTION" function-identifier

   statement-sequence = statement {";" statement}


7  simple-type = ordinal-type | real-type | longreal-type

   structured-type = ["PACKED"] unpacked-structured-type |
                     dynamic-string-type |
                     structured-type-identifier

   pointer-type = "^" type-identifier | pointer-type-identifier

   procedure-heading = "PROCEDURE" procedure-identifier [formal-parameter-list]

   function-heading = "FUNCTION" function-identifier [formal-parameter-list]
                      ":" result-type

   procedure-identifier = identifier

   function-identifier = identifier

   statement = [label ":"] (simple-statement | structured-statement)
```

```
8  ordinal-type = enumerated-type | subrange-type | integer-type |
                  boolean-type | char-type | ordinal-type-identifier

   real-type = "real"

   longreal-type = "longreal"

   unpacked-structured-type = array-type | record-type | set-type | file-type

   dynamic-string-type = "string" [ "[" constant "]" ]

   structured-type-identifier = type-identifier

   pointer-type-identifier = type-identifier

   formal-parameter-list = "(" formal-parameter-section
                           {";" formal-parameter-section} ")"

   result-type = simple-type-identifier | pointer-type-identifier

   simple-statement = empty-statement | assignment-statement |
                      procedure-statement | goto-statement

   structured-statement = compound-statement | conditional-statement |
                          repetitive-statement | with-statement

9  enumerated-type = "(" identifier-list ")"

   subrange-type = constant ".." constant

   integer-type = "integer"

   boolean-type = "boolean"

   char-type = "char"
```

```
ordinal-type-identifier = type-identifier

array-type = "ARRAY" "[" index-type {"," index-type} "]" "OF" type-denoter

record-type = "RECORD" [field-list [";"]] "END"

set-type = "SET" "OF" ordinal-type

file-type = "FILE" "OF" type-denoter

formal-parameter-section = value-parameter-specification |
                           variable-parameter-specification |
                           procedural-parameter-specification |
                           functional-parameter-specification

simple-type-identifier = type-identifier

empty-statement =

assignment-statement = (variable-access | function-identifier) ":=" expression

procedure-statement = procedure-identifier [actual-parameter-list]

goto-statement = "GOTO" label

conditional-statement = if-statement | case-statement

repetitive-statement = repeat-statement | while-statement | for-statement

with-statement = "WITH" record-variable-list "DO" statement
```

```
10  index-type = ordinal-type

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

    value-parameter-specification = identifier-list ":" type-identifier

    variable-parameter-specification = "VAR" identifier-list ":" type-identifier

    procedural-parameter-specification = procedure-heading

    functional-parameter-specification = function-heading

    variable-access = entire-variable | indexed-variable | field-designator |
                      referenced-variable | buffer-variable

    expression = simple-expression [relational-operator simple-expression]

    actual-parameter-list = "(" actual-parameter {"," actual-parameter}")"

    if-statement = "IF" boolean-expression "THEN" statement ["ELSE" statement]

    case-statement = "CASE" case-index "OF" case-list-element
                     {";" case-list-element} [";" "OTHERWISE" statement][";"] "END"

    repeat-statement = "REPEAT" statement-sequence "UNTIL" boolean-expression

    while-statement = "WHILE" boolean-expression "DO" statement

    for-statement = "FOR" control-variable ":=" initial-value
                    ("TO" | "DOWNTO") final-value "DO" statement

    record-variable-list = record-variable {"," record-variable}
```

```
11  fixed-part = record-section {";" record-section}

    variant-part = "CASE" [tag-field ":"] tag-type "OF" variant {";" variant}

    entire-variable = variable-identifier

    indexed-variable = array-variable "[" index-expression
                                  {"," index-expression} "]" |
                       dynamic-string-variable "[" index-expression "]"

    field-designator = record-variable "." field-identifier

    referenced-variable = pointer-variable "^"

    buffer-variable = file-variable "^"

    simple-expression = [sign] term {adding-operator term}

    relational-operator = "=" | "<>" | "<" | ">" | "<=" | ">=" | "IN"

    actual-parameter = expression | variable-access |
                       procedure-identifier | function-identifier

    boolean-expression = expression

    case-index = expression

    case-list-element = case-constant-list ":" statement

    control-variable = entire-variable

    initial-value = expression

    final-value = expression

    record-variable = variable-access
```

```
12  record-section = identifier-list ":" type-denoter

    tag-field = identifier

    tag-type = ordinal-type-identifier

    variant = case-constant-list ":" "(" [field-list [";"]] ")"

    variable-identifier = identifier

    array-variable = variable-access

    dynamic-string-variable = variable-access

    index-expression = expression

    field-identifier = identifier

    pointer-variable = variable-access

    file-variable = variable-access

    term = factor {multiplying-operator factor}

    adding-operator = "+" | "-" | "OR"

    case-constant-list = case-constant {"," case-constant}
```

```
13  factor = variable-access | unsigned-constant | function-designator |
             set-constructor | "(" expression ")" | "NOT" factor

    multiplying-operator = "*" | "/" | "DIV" | "MOD" | "AND"

    case-constant = constant

14  unsigned-constant = unsigned-number | character-string |
                        constant-identifier | "NIL"

    function-designator = function-identifier [actual-parameter-list]

    set-constructor = "[" [member-designator {"," member-designator}] "]"

15  member-designator = expression [".." expression]
```

B COMPILE-TIME ERRORS

For each error number, the text which is printed at compile time (provided PROPAS.ERR is present) is given, plus extra explanation where necessary.

Number	Meaning
1	Simple type expected
2	Identifier expected
3	PROGRAM or SEGMENT expected
4	) expected
5	: expected
6	Symbol illegal in this context May be due to an error in the preceding line, such as missing semicolon.
7	Error in parameter list
8	OF expected
9	(expected
10	Error in type
11	[expected
12	] expected
13	END expected
14	; expected
15	Integer expected (in a LABEL declaration, or after GOTO)
16	= expected (in a CONST or TYPE declaration)
17	BEGIN expected

18	Error in declaration part Declaration processing has finished, and statement-part is expected. May be due to incorrect ordering of declarations, e.g. TYPE before CONST
19	Error in field-list
20	, expected
21	. expected
24	Illegal source character
25	One identifier may not follow another
49	No cases in case statement
50	Error in constant
51	:= expected (in an assignment statement or FOR statement)
52	THEN expected
53	UNTIL expected
54	DO expected
55	TO or DOWNTO expected
58	Error in factor
59	Error in variable
101	Identifier declared twice The first 8 characters of the identifier are printed
102	Low bound exceeds high bound
103	Identifier is not of appropriate class The first 8 characters of the identifier are printed
104	Identifier has not been declared The first 8 characters of the identifier are printed
105	Sign not allowed
106	Number expected
107	Incompatible subrange types In c1..c2, type of c1 is not compatible with type of c2

108 File not allowed here
A file may not be a component of another file-type

109 Type must not be real

110 Tagfield type must be ordinal type

111 Incompatible with tagfield type
Refers to a case-constant in a record variant, or a tag value in a call of new or dispose

113 Index type must be ordinal type

114 Base type exceeds set range
Base type of set is outside the (ordinal) range 0..2039

115 Base type of set must be ordinal type

116 Bad parameter type for standard procedure

117 Unsatisfied forward reference
The name of the undefined type (which will have occurred after ^ in a type denoter) is printed on next line

119 Repetition of parameter list not allowed
Parameter list may only be specified at the place where the FORWARD declaration is made

120 Function type may be scalar/subrange/pointer
These are the only permitted result types for a function

121 File value parameter not allowed
This applies also to structured types with file components

122 Repetition of result type not allowed
The result type may only be specified at the place where the FORWARD declaration of the function is made

123 Function declaration with no result type

124 Second width parameter is for reals only
(In actual parameter list of write or writeln to a textfile)

125 Bad parameter type for standard function

126 No. of parameters differs from declaration

127 Illegal variable parameter
Actual parameter corresponding to a VAR formal may not be a tag-field, nor a component of a PACKED structured-type

128 Functional-parameter's type is incorrect

129 Operand types incompatible

130 Expression is not of set type

131 Tests on equality allowed only
(for pointer types)

132 Strict set inclusion not allowed
If s1, s2 are set-type, s1<s2 and s1>s2 are illegal

133 Relational operation not allowed
(on arrays, records or files)

134 Illegal type of operand(s)

135 Type of operand must be boolean

136 Set element type must be ordinal type

137 Set element types not compatible

138 Type of variable is not array

139 Index type incompatible with declaration

140 Type of variable is not record
The item preceding '.' in a variable-access, or the variable in a WITH statement, is not record-type

141 Type of variable must be file or pointer
(before '^')

142 Incompatible parameter type

143 Illegal type of loop control variable
Control variable of FOR statement must be ordinal-type

144 Illegal type of expression

145 Incompatible type
Initial- or final-value in a FOR statement incompatible with type of control-variable

146 Assignment of files is not allowed
This applies also to structured types with file components

147 Case-constant of invalid type
(in a CASE statement)

148 Subrange bounds must be ordinal type

149 Index- or tag-type must not be integer
If integer-type, must be a subrange

150 Standard function name not allowed here

151 Assignment to formal function is illegal

152 No such field in this record

154 Actual parameter must be a variable

155 Control variable must be local to block
In particular, may not be in COMMON

156 Multidefined case label
(in a CASE statement)

157 Too many cases in case statement
More than 4096

158 No corresponding variant declaration
Too many parameters in a call of new or dispose

159 Real or string case-constant not allowed

160 Previous declaration was not FORWARD
A second declaration of the same procedure or function has been encountered in a block

161 Already declared as FORWARD

162 Error in record structure

163 Missing variant(s) in declaration
(If Standard Pascal option selected, only.) In a record declaration with variant part, there must be one case-constant for every value of the tag-type

164 Standard procedure/function not allowed here

165 Multidefined label

166 Multideclared label

167 Undeclared label

168 Undefined label
The value of the label is printed

175 Missing file 'input' in program heading
(If Standard Pascal option selected, only)

176 Missing file 'output' in program heading
(If Standard Pascal option selected, only)

177 Assignment to function not allowed here
Must be within function's block

178 Multidefined record variant
Case-constant not unique

179 Control variable is not secure
The control varaible of a FOR-loop may not be modified in the body of an inner-level procedure or function

180 Control variable must not be formal
(in FOR statement)

181 Attempt to alter control variable
The control variable has been modified during the FOR-loop

182 Label value out of range
Not in 0..9999

183 Label jumped to from an illegal position
(If Standard Pascal option selected, only)

184 GOTO: label is not accessible
(If Standard Pascal option selected, only)

201 Error in real constant: digit expected

202 String constant exceeds source line

203 Integer constant too large
Exceeds maxint

205 Null string not allowed
'' is illegal in Standard Pascal

207 Error in hex constant

208 Constant not properly terminated

209 Exponent of real constant out of range

250 Too many nested scopes of identifiers
Depth (including scopes opened by RECORD or WITH) exceeds 17

251 Too deep nesting of procedures/functions
Depth exceeds 15

252 Too deep nesting of FOR/WITH statements
(Message appears at end of procedure or function block)

253 Too deep nesting of source file inserts
Depth exceeds 3

261 Compiler update-stack overflow
More than 192 names at inner block levels are in scope

263 Too many COMMON names
More than 128

270 Static data area exceeds 64K bytes

271 Stack requirement exceeds 32K bytes
(in one procedure or function)

272 Code area exceeds 64K bytes

301 No case provided for this value
Illegal tag value in call of new or dispose

302 Index expression out of bounds

304 Set element expression out of range
Outside the (ordinal) range 0..2039

305 Warning: FOR loop will never be executed
Final-value < initial-value (if TO), or > initial-value (if DOWNTO). Warning only: a useable .REL file will be produced

306 Range error
Range checking requested, and constant out of bounds. Warning only: a useable .REL file will be produced

308 Case-constant outside range of tag-type

310 String size must be integer constant
(in the declaration string[n])

311 String length exceeded

320 Incompatible parameter-lists

322 Undeclared FORWARD procedure or function
The first 8 characters of the name are printed on the next line. This error can be due to incorrect block structure - an extra END, for example

323 Identifier referenced before declaration
The first 8 characters of the identifier are printed

324 No value assigned to function

325 Variable referenced but never defined
The name is printed after the error, which occurs at end of block in which the variable is declared, and means the variable has not been given a value

330 Source insert filename illegal/not found

333 End of source file encountered
Probably due to incorrect block structure (a missing END, etc.), or to an unclosed comment (a missing "}")

349 COMMON declaration not allowed here
Only allowed at outermost level

350 Illegal SEGMENT structure
Segment contains, at outer level, a LABEL declaration, or a statement-part different from BEGIN END

351 COMMON name not unique in 7 characters
The name is printed

352 EXTERNAL name not unique in 7 characters
The name is printed

353 Entry name not unique in 7 characters
The name - that of an outer-level procedure or function - is printed

380 eof on compiler work file
Errors 380 thru 386 should not normally occur, and may indicate a compiler malfunction

381 eof on compiler work file

382 Compiler work file contents invalid

386 Compiler work file contents invalid

398 Pro Pascal implementation restriction
An enumerated type can have at most 256 identifiers

399 Pro Pascal extension to Standard
If the compiler has been requested to 'Accept only strict Standard Pascal', this error indicates that a Pro Pascal extension has been used, e.g. a hex constant, or an implementation-dependent additional predeclared procedure

C RUN-TIME ERROR CODES

The format of the messages produced for run-time errors is given in Part III under "Operation of object programs". This appendix lists the error codes, with significance and possible causes.

Code	Meaning
A	Angle argument error. From sin or cos when the argument is so large that range reduction would lead to serious loss of accuracy. Real argument: abs(value) > 32768.0 Longreal argument: abs(value) > 4.295D9
B	Bounds exceeded. An index bound has been exceeded (with /I compile-time option selected) or a value is outside the range of the receiving field in an assignment (with the /A option selected).
C	Case error. No case constant corresponding to expression value (and no OTHERWISE specified). Continuation is to the statement following the CASE statement.
D	Disc or Device error. Unable to open input file (filename displayed). Disc or directory space insufficient for output. Attempted reset of an output device (e.g. LST:) or rewrite of an input device.
E	Reading beyond EOF. Program does not correctly check for end-of-file condition.
F	File programming error. Incorrect sequence of operations on a file (e.g. attempted read before reset).
H	Heap overflow. Insufficient free space for "new" operation.
J	Divide error (integers). In "i DIV j" j is zero; in "i MOD j" j is zero or negative. Continuation possible, but results not predictable.

K Overflow on TRUNC or ROUND.

Conversion of the real value to integer gives a value outside integer range.

L LN argument error.

Argument to "ln" function is zero or negative.

N Name format error.

The name given in an "assign" operation is not in correct format for a CP/M file or device name.

O Overflow during integer arithmetic.

From 32-bit add, subtract, multiply (always checked), or from 8-bit/16-bit DIV when option /R invoked, or from 8-bit/16-bit add, subtract when option /A or /I invoked in combination with /R. Continuation possible, truncated result used.

P Pointer not valid.

The variable used as a pointer contains a value which is not valid. From "dispose" (checked always) or from pointer dereference when checking option specified.

Q SQRT argument error.

Negative argument to "sqrt" function. Continuation possible, the result returned being zero.

R Read error on textfile.

During reading of an integer, real or longreal value from a textfile, incorrect format of input. Continuation possible, but value unpredictable.

S Space insufficient.

The dynamic stack used for parameters and local variables of procedures has exceeded the space available.

T Error in string handling operation

String value assigned or passed as an actual parameter exceeds the size of the receiving variable or formal parameter; "concat" exceeds 255 characters; index value given to "copy", "delete", or "insert" is zero or beyond current length of the string.

U Illegal argument to SEEK.

The specified element number is negative or beyond the maximum file size.

V Set construction error.

A set expression contains element(s) outside the range 0 to 2039.

W Write error on textfile.

Fieldwidth parameter is outside the range 1 to 255.

X Overflow during real or longreal arithmetic

Exponent out of range. Continuation possible but results not predictable.

Z Divide by zero (reals or longreals).

Continuation possible, but results not predictable.

The Pascal Standard (ISO 7185) contains as Appendix D a list of 59 "errors", and requires that there be a statement describing how each is treated.

If all compile-time options have their default ("off") values, the following errors are detected prior to, or during, execution of a program:
D.9, D.10, D.11, D.14, D.15, D.16, D.23, D.32, D.33, D.34, D.35, D.36, D.40, D.41, D.42, D.44, D.45, D.46, D.47, D.51, D.54, D.56, D.57, D.58

If the /A compile-time option is specified, the following additional errors are detected at run time:
D.7, D.8, D.17, D.18, D.37, D.49, D.50, D.52, D.53, D.55

If the /I compile-time option is specified, the following additional errors are detected at run time:
D.1, D.26, D.29

If the /P compile-time option is specified, error D.3 is, additionally, detected.

The following errors are not, in general, reported:
D.2, D.4, D.5, D.6, D.12, D.13, D.19, D.20, D.21, D.22, D.24, D.25, D.27, D.28, D.30, D.31, D.38, D.39, D.43, D.48

(These last errors are mainly to do with referencing undefined or uninitialised variables, referencing fields in "non-active" variants of records, and so on. For full details, refer to the Standard.)

D ASCII CHARACTER SET

Hex	Character	Hex	Character	Hex	Character	Hex	Character
00	NUL	20	space	40	@	60	`
01	SOH	21	!	41	A	61	a
02	STX	22	"	42	B	62	b
03	ETX	23	£	43	C	63	c
04	EOT	24	$	44	D	64	d
05	ENQ	25	%	45	E	65	e
06	ACK	26	&	46	F	66	f
07	BEL	27	'	47	G	67	g
08	BS	28	(	48	H	68	h
09	HT	29	)	49	I	69	i
0A	LF	2A	*	4A	J	6A	j
0B	VT	2B	+	4B	K	6B	k
0C	FF	2C	,	4C	L	6C	l
0D	CR	2D	-	4D	M	6D	m
0E	SO	2E	.	4E	N	6E	n
0F	SI	2F	/	4F	O	6F	o
10	DLE	30	0	50	P	70	p
11	DC1	31	1	51	Q	71	q
12	DC2	32	2	52	R	72	r
13	DC3	33	3	53	S	73	s
14	DC4	34	4	54	T	74	t
15	NAK	35	5	55	U	75	u
16	SYN	36	6	56	V	76	v
17	ETB	37	7	57	W	77	w
18	CAN	38	8	58	X	78	x
19	EM	39	9	59	Y	79	y
1A	SUB	3A	:	5A	Z	7A	z
1B	ESC	3B	;	5B	[	7B	{
1C	FS	3C	<	5C	\	7C	¦
1D	GS	3D	=	5D	]	7D	}
1E	RS	3E	>	5E	^	7E	~
1F	US	3F	?	5F	_	7F	DEL