

PRO PASCAL USER MANUAL

Version mmq 1.1

for Sinclair QL
with QDOS

April 1986

Copyright (C) 1986 Prospero Software
190 Castelnau London SW13 9DH England

1. Pascal	9
1.1. Pro Pascal	9
2. Format of this Manual	9
3. Introduction To Pascal	10
3.1. Example Pascal Program	10
3.2. General Layout and Appearance	11
3.2.1. Program skeleton	11
3.2.2. Symbols, Words, and Constants	12
3.2.3. Identifiers and reserved words	12
3.2.4. Program appearance	13
3.2.5. Comments	13
3.3. Statements	13
3.3.1. Expressions	13
3.3.2. Simple, conditional, and repetitive statements	14
3.3.3. Simple statements	15
3.3.4. Conditional statements	15
3.3.5. Repetitive statements	16
3.4. Labels and Label Declarations	17
3.5. CONST Declarations	17
3.6. Data Types and Type Declarations	17
3.6.1. Data types	17
3.6.2. Built-in types	18
3.6.3. User-defined types	18
3.6.3.1. Set	19
3.6.4. Arrays	20
3.6.5. Strings	20
3.6.6. Records	21
3.6.7. Pointers	22
3.6.8. Files	23
3.6.8.1. Text files	23
3.6.8.2. Other files	23
3.6.8.3. Common concepts	23
3.6.8.4. Special features of text files	24
3.6.8.5. Special features of non-text files	25
3.7. Variables and VAR Declarations	25
3.7.1. Variable Declarations	25
3.7.2. Reference to variables	25
3.8. Procedures and Functions	26
3.8.1. Declaring and using simple procedures	27
3.8.2. Parameters	27
3.8.3. Functions	28
3.8.4. Standard procedures	29
3.8.4.1. String-handling procedures	29
3.9. Getting Started	30
3.9.1. Think ahead	30
3.9.2. Choice of names	30
3.9.3. Compilation and linking	30
3.10. Conclusion	31
3.11. Further Reading	31
4. Pro Pascal Language Definition	32
4.1. Lexical Aspects	32
4.1.1. Tokens	32
4.1.1.1. Special symbols	32
4.1.1.2. Identifiers	33
4.1.1.3. Directives	33
4.1.1.4. Labels	33

4.1.1.5. Unsigned numbers.....	33
4.1.1.5.1. Unsigned integers	33
4.1.1.5.2. Unsigned reals	34
4.1.1.5.3. Unsigned longreals.....	34
4.1.1.6. Character strings.....	34
4.1.2. Separators	35
4.1.3. Comments	35
4.1.3.1. Source file insertion	35
4.1.3.2. Page throw on listing.....	35
4.2. Programs, Segments and Blocks.....	35
4.2.1. Programs	36
4.2.2. Segments	36
4.3. Statements and Expressions	37
4.3.1. Statements	37
4.3.1.1. Empty statement.....	37
4.3.1.2. Assignment statement	38
4.3.1.3. Procedure statement.....	38
4.3.1.4. GOTO statement.....	38
4.3.1.5. Structured statements.....	39
4.3.1.6. Compound statement.....	39
4.3.1.7. Conditional statements	39
4.3.1.8. IF statement.....	39
4.3.1.9. CASE statement	40
4.3.1.10. Repetitive statements	40
4.3.1.10.1. REPEAT statement	41
4.3.1.10.2. WHILE statement	41
4.3.1.10.3. FOR statement.....	41
4.3.1.11. WITH statement	42
4.3.2. Expressions	43
4.3.2.1. Factors.....	43
4.3.2.1.1. Variable access	43
4.3.2.1.2. Entire variable	44
4.3.2.1.3. Indexed variable.....	44
4.3.2.1.4. Field designator.....	44
4.3.2.1.5. Referenced variable	45
4.3.2.1.6. Buffer variable	45
4.3.2.2. Unsigned constant	45
4.3.2.3. Function designator	45
4.3.2.4. Set constructor.....	46
4.3.2.5. Arithmetic operators.....	46
4.3.2.5.1. +	46
4.3.2.5.2. -	46
4.3.2.5.3. *	47
4.3.2.5.4. /	47
4.3.2.5.5. DIV	47
4.3.2.5.6. MOD	47
4.3.2.6. Boolean operators.....	47
4.3.2.6.1. OR.....	47
4.3.2.6.2. AND.....	47
4.3.2.6.3. NOT.....	47
4.3.2.7. Set operators	47
4.3.2.8. Relational operators.....	48
4.3.2.8.1. = and <>	48
4.3.2.8.2. < and >	48
4.3.2.8.3. <= and >=	48
4.4. Labels.....	49
4.4.1. Declaration of labels	49

4.4.2. Definition of labels	49
4.4.3. Reference to labels	49
4.5. CONST Declarations	50
4.6. Type Definitions	50
4.6.1. Type denoters	50
4.6.1.1. Standard simple types	51
4.6.1.1.1. Real	51
4.6.1.1.2. Longreal	51
4.6.1.1.3. Integer	51
4.6.1.1.4. Boolean	52
4.6.1.1.5. Char	52
4.6.1.2. Enumerated types	52
4.6.1.3. Subrange types	52
4.6.1.4. Structured types	52
4.6.1.5. Array types	53
4.6.1.6. Record types	54
4.6.1.7. Set types	54
4.6.1.8. File types	55
4.6.1.9. Pointer types	55
4.6.2. Type Compatibility	55
4.6.2.1. Assignment-compatible types	55
4.7. Variable Declarations	56
4.8. Procedures and Functions	56
4.8.1. Procedure and function declarations	57
4.8.1.1. Procedure and function heading	57
4.8.1.1.1. Procedure heading	57
4.8.1.1.2. Function heading	57
4.8.1.1.3. Parameters	57
4.8.1.1.4. Value parameter	58
4.8.1.1.5. VAR parameter	58
4.8.1.1.6. Procedural and functional parameters	58
4.8.1.2. Procedure or function block	58
4.8.1.3. Directives	59
4.8.1.3.1. FORWARD declarations	59
4.8.1.3.2. EXTERNAL declarations	59
4.8.2. Activation of procedures and functions	60
4.8.2.1. Activation of procedures	60
4.8.2.2. Activation of functions	60
4.8.2.3. Actual parameters	60
4.8.2.3.1. Value parameters	60
4.8.2.3.2. VAR parameters	60
4.8.2.3.3. Procedural (functional) parameters	61
4.8.3. Standard Procedures and Functions	61
4.8.3.1. Operations on files	61
4.8.3.1.1. eof and eoln	61
4.8.3.1.2. reset and rewrite	62
4.8.3.1.3. get and put	62
4.8.3.1.4. page	62
4.8.3.1.5. read	62
4.8.3.1.6. readln	63
4.8.3.1.7. write	63
4.8.3.1.8. writeln	64
4.8.3.2. new and dispose	64
4.8.3.3. pack and unpack	65
4.8.3.4. trunc and round	65
4.8.3.5. ord and chr	65
4.8.3.6. succ and pred	66

4.8.3.7. abs and sqr	66
4.8.3.8. sqrt, sin, cos, exp, ln, arctan	66
4.8.3.9. odd	66
4.8.3.10. Dynamic-String Procedures and Functions	66
4.8.3.10.1. concat(s1,s2,...)	67
4.8.3.10.2. copy(stringval,index,count)	67
4.8.3.10.3. insert(stringval,stringvar,index)	67
4.8.3.10.4. delete(stringvar, index, count)	67
4.8.3.10.5. length(stringval)	67
4.8.3.10.6. pos(substr,stringval)	68
4.8.3.10.7. str(intexp,stringvar)	68
4.9. Implementation-dependent Aspects	68
4.9.1. Pascal Files and QDOS	68
4.9.1.1. Declaration of files	68
4.9.1.2. File assignment	68
4.9.1.3. File formats	69
4.9.1.3.1. Text files	69
4.9.1.3.2. Non-text files	69
4.9.1.4. Additional procedures and functions	69
4.9.2. Additional standard procedures	70
4.9.2.1. assign	70
4.9.2.2. update	71
4.9.2.3. seek	71
4.9.2.4. position	72
4.9.2.5. close	72
4.9.2.6. erase	72
4.9.2.7. fstat	72
4.9.2.8. checkfn	72
4.9.2.9. append	72
4.9.2.10. rename	72
4.9.2.11. ramfile	73
4.9.2.12. echo	73
4.9.2.13. handle	73
4.9.2.14. move	74
4.9.2.15. getcomm	74
4.9.2.16. sizeof	74
4.9.2.17. addr	74
4.9.2.18. peek	74
4.9.2.19. poke	75
4.9.3. Library facilities	75
4.9.3.1. memavail	75
4.9.3.2. rand and seed	75
4.9.3.3. consingle	75
4.9.3.4. consilent	75
4.9.3.5. prompt	75
4.9.3.6. ownerr	76
4.9.3.7. textnote and textpoint	76
4.9.3.8. execprog	77
4.9.3.9. exitprog	78
4.9.3.10. date	79
4.9.3.11. time	79
4.9.3.12. qtrap	79
4.9.3.13. mode	80
4.9.3.14. Window routines	80
4.9.3.14.1. wopen	80
4.9.3.14.2. window	81
4.9.3.14.3. wstatc	81

4.9.3.14.4. wstatp	81
4.9.3.14.5. recol.....	81
4.9.3.14.6. border	82
4.9.3.14.7. ink.....	82
4.9.3.14.8. paper	82
4.9.3.14.9. strip.....	82
4.9.3.14.10. block.....	83
4.9.3.14.11. cls	83
4.9.3.14.12. pan	83
4.9.3.14.13. scroll	84
4.9.3.15. Print style routines	84
4.9.3.15.1. csize	84
4.9.3.15.2. flash.....	85
4.9.3.15.3. under	85
4.9.3.15.4. over	85
4.9.3.16. Cursor positioning routines	85
4.9.3.16.1. atc.....	85
4.9.3.16.2. atp	86
4.9.3.16.3. atg	86
4.9.3.17. Graphics drawing routines	86
4.9.3.17.1. fill	86
4.9.3.17.2. scale	86
4.9.3.17.3. point.....	87
4.9.3.17.4. line.....	87
4.9.3.17.5. arc	87
4.9.3.17.6. circle	87
4.9.3.17.7. ellipse	88
4.10. Storage allocation	88
4.10.1. Overall layout.....	88
4.10.1.1. Formats of variables	90
4.10.1.2. Preservation of registers	94
4.10.1.3. Parameters	94
4.10.1.3.1. Value parameters	95
4.10.1.3.2. VAR parameters	95
4.10.1.4. Function results.....	95
4.10.1.5. Reserved section names	95
5. Pro Pascal Operation.....	95
5.1. Installation Details	95
5.1.1. Simple compile. link and execute	97
5.1.1.1. Compile.....	97
5.1.1.2. Link	97
5.1.1.3. Execute	98
5.2. Operation of the Compiler	98
5.2.1. Forms of invocation	98
5.2.1.1. Interactive mode	98
5.2.1.2. Batch mode.....	99
5.2.2. Compile-time options.....	99
5.2.2.1. G -console output to LOG file	99
5.2.2.2. I -range checks on index bounds	99
5.2.2.3. A -range checks on assignments.....	100
5.2.2.4. P -checks on pointers	100
5.2.2.5. N -track source names & line numbers at run time.....	100
5.2.2.6. L -source listing.....	100
5.2.2.7. D -double precision floating-point constants	100
5.2.2.8. C -compact object code	100
5.2.2.9. S -accept only strict Standard Pascal	101
5.2.3. Compiler messages.....	101

5.2.3.1. Normal messages	101
5.3. Operation of the Linker	102
6. Operation of Object Programs	103
6.1. Execution of Pascal object programs	104
6.1.1. Invocation by EXEC or EXEC_W	104
6.1.1.1. Invocation by EX or EW Toolkit commands	105
6.1.1.2. Handling of pre-connected files	105
6.1.2. Invocation using execprog	105
6.1.3. Run-time errors	105
6.2. Miscellaneous error messages	106
6.3. The Configuration Programs	107
6.3.1. Configuring the compiler	107
6.3.2. Configuring object programs	108
6.3.2.1. Default device -SETDDEV	108
6.3.2.2. Initial dialogue -NOQNS	108
6.4. Operation of the Librarian	109
6.4.1. Forms of invocation	109
6.4.1.1. The one-line command	109
6.4.1.2. Conversational mode	110
6.4.1.3. Indirect mode	110
6.4.2. Report options	111
6.4.2.1. Module listing (M)	111
6.4.2.2. Cross-reference listing (X)	111
6.4.2.3. Unsatisfied references listing (U)	111
6.4.2.4. Suppress .names (N)	111
6.4.2.5. Listings to disc (D)	111
6.4.2.6. Module selection	111
6.4.3. Librarian messages	112
6.4.3.1. Normal messages	112
6.4.3.2. Error messages	112
6.4.3.2.1. Non-fatal errors	112
6.4.3.2.2. Fatal errors	112
6.5. Cross-Reference Generator	113
6.5.1. Forms of invocation	113
6.5.2. Messages	114

COPYRIGHT

This document is copyright and may not be reproduced by any method, translated, transmitted, or stored in a retrieval system without prior written permission of Prospero Software.

Permission is granted to Pro Pascal licence holders to abstract and use any of the programming examples.

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 QL, ODDS, Microdrive, QL Toolkit and SuperBASIC, which are trademarks of Sinclair Research Limited, and to Motorola and MC68000, which are trademarks of Motorola, Inc.

Pro Pascal and Pro Fortran-77 are trademarks of Prospero Software.

1. 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.

1.1. Pro Pascal

Pro Pascal complies with all the requirements of ISO 1185, Level (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.

2. 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, linker, 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 observations on mixed-language programs.

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.

3. Introduction To Pascal

3.1. 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
```

```
1: PROGRAM results (input);
2:
3: CONST maxevents = 8; {maximum events for one entrant}
4:   colwidth = 5; {column width on tabulation}
5:
6: TYPE competitor = 100.. 999; {range of entrants' numbers}
7: score = 0.. 100; {possible scores for one event}
8:
9: VAR thiscomp, winner: competitor;
10:    eventscore: score; totalscore: integer;
11:    eventcount: 0..maxevents;
12:    average, wlnningav: real;
13:    listing: text; {output file for tabulation}
14:
15: BEGIN
16:   winningav := 0;
17:   assign (listing, 'MDV1_RESULTS_PRN'); rewrite (listing);
18:
19:   {process input and produce listing}
20:   WHILE NOT eof (input) DO
21:   BEGIN {read competitor number}
22:     read (thiscomp);
23:     write (listing, thiscomp:5, I ':3);
24:
25:     eventcount := 0; totalscore:= 0;
26:     {now his scores until end-or-line}
27:     WHILE NOT eoln (input) DO
28:     BEGIN
```

```

29:         read (eventscore);
30:         write (listing, eventscore:colwidth);
31:         totalscore := totalscore + eventscore;
32:         eventcount := eventcount + 1;
33:     END {of processing one result};
34:
35:     {space across to totals column}
36:     IF eventcount < maxevents THEN
37:         write (listing, ': (maxevents-eventcount).colwidth);
38:         {calculate & print average}
39:         IF eventcount = 0 THEN average := 0
40:         ELSE average := totalscore / eventcount;
41:         writeln (listing, totalscore:8, average:6:2);
42:         {is average greater than current winner? }
43:         IF (eventcount >= 5) AND (average > winningav) THEN
44:             BEGIN
45:                 winner := thiscomp; {best so far}
46:                 winningav := average;
47:             END;
48:         readln ;
49:     END {of processing one competitor};
50:
51:     {at end of input, print winner}
52:     writeln (listing); writeln (listing); {blank lines}
53: IF winningav > 0 THEN
54:     writeln (listing, Winner is number', winner:5,
55:     with average', winningav:6:2)
56: ELSE writeln (listing, ' No entrant qualified as winner');
57:
58: 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.

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.

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) .

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.

3.2. General Layout and Appearance

3.2.1. 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.

3.2.2. 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.)

3.2.3. 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 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 11.

(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.)

3.2.4. 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.

3.2.5. 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.

3.3. 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.

3.3.1. 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 programming 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)

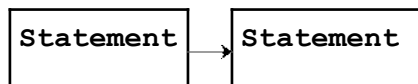
Logical operations are called for by the reserved-word symbols **AND**, **OR** and **NOT**. (Pascal does not allow **AND** to be used as a masking operation, since that implies implementation-dependent knowledge about the internal representation.) Example expressions:

```
5
intvar
ind + interval
units-10
balance ; limit
(a < b) OR (0 = 6)
thickness + 4 * (length -height)
```

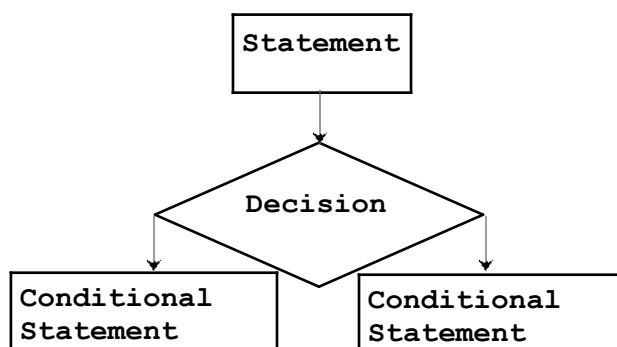
Use of functions within expressions is covered in the description of procedures and functions, and a few other special forms are mentioned as they arise.

3.3.2. Simple. conditional. and repetitive statements

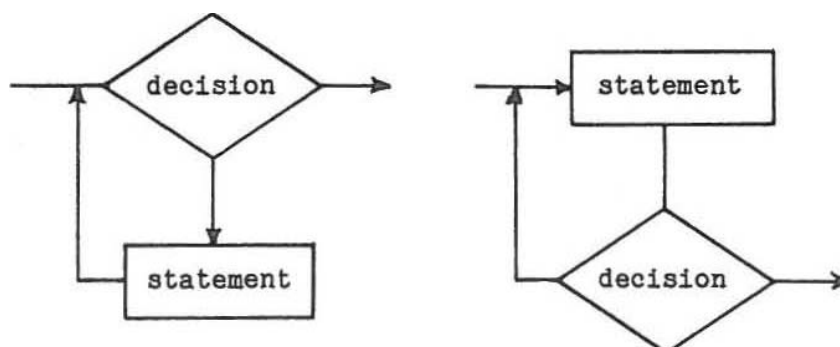
It is useful to categorise statements according to the way the flow of control passes through them. The so-called "simple" statements are obeyed just once:



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:

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 or 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 layout a compound statement so that the **BEGIN** and **END** can be seen to match.

3.3.3. Simple statements

There are just three kinds of simple statement.

- An assignment statement gives the value or 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.)

- A procedure statement invokes execution of the procedure - this is described later under "procedures and functions".
- Special instances of procedure calls are the read and write operations, for example:

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

- **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.

3.3.4. Conditional statements

Pascal has two forms of conditional statement:

- 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**).

- 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}
```

3.3.5. Repetitive statements

There are three repetitive statements:

- 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.

- The **REPEAT** statement has its exit at the end, and the controlled statement is obeyed at least once:

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

- The **FOR** statement provides a combined loop and count facility:

```
FOR a := 1 TO 10 DO
  process;
FOR b := 10 DOWNT0 1 DO
  anotherproces;
FOR i := 2-n TO twicemax DO
  yetanother;
```

The variable (a, b, or i) 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 **DOWNT0** form if **e1** is less than **e2**). The increment/decrement is always 1.

3.4. 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;
```

3.5. 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:

1. The name can be chosen to make the program self-documenting;
2. 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:';
```

3.6. 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.

3.6.1. 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 .

3.6.2. Built-in types

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

1. **char** - the data item is a character, in Pro Pascal (as in many other implementations) one from the ASCII character set.
2. **integer** - the item is an integer. In Pro Pascal the range of integers is nine decimal digits (to be exact: -2178367 to +2178367) •
3. **real** - the item is a floating-point quantity.
4. **longreal** - an extended-precision floating-point quantity.
5. **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.

3.6.3. User-defined types

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.

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 anyone of the values Sunday to Saturday, but may not take a value which is not in the list {IO, 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.

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.

3.6.3.1. Set

A set declaration is of the form

```
... 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 11. 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.

3.6.4. 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. Here 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.

3.6.5. 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 as-

signed 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]"**.

3.6.6. 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..2050;
  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 11 of this manual.

3.6.7. 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 6.3.1 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:

3.6.8. 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.

3.6.8.1. 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.)

3.6.8.2. 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 11, 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 sequential - 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.

3.6.8.3. 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 possibilities 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.

3.6.8.4. 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 **eofln**. 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 11.

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.

3.6.8.5. 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.

3.7. Variables and VAR Declarations

3.7.1. 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 or 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 or 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.

3.7.2. 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.

3.8. 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.

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.

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.

3.8.1. Declaring and using simple procedures

A procedure such as alpha in the example above 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.

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

Here, upper has a "Formal parameter" **fcb** 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

1. they are used without being declared,
2. 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 11.)

3.8.3. Functions

In fact, `ord` and `chr` are examples of *functions*. Other examples are `sqr(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));
```

3.8.4. 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**, **In**, and **arctan** which can be used within expressions whenever needed.

3.8.4.1. 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);  
VAR  
  {Copy a textfile from "source" to "output" with line numbers.}  
  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;

```

3.9. 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 11 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.

3.9.1. 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.

3.9.2. 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.

3.9.3. 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 lan-

guage 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 misspelled 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

3.10. 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.

3.11. 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, 1982 (2nd edition)
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.

4. Pro Pascal Language Definition

4.1. 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. Tab characters encountered in the source file are expanded into one or more blanks, until the next "tab stop" is reached, these being at columns 1, 9, 17 and so on (every 8 columns).

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

4.1.1. Tokens

These are of 6 kinds:

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

4.1.1.1. Special symbols

The special-symbols are tokens with special fixed meanings.

```
special-symbol = "+" | "-" | "a" | "|" | "=" | "(" | ">" | "[" |
                "1" | "." | "," | ":" | ";" | If"" "(" a)" |
                ">" | "(=" | ">=" | ":=" | " " | word-symbol
word-symbol = "AND" | "ARRAY" | "BEGIN" | "CASE" | "COMMON" |
              "CONST" | "DIV" | "DO" | "DOWNT0" | "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.

4.1.1.2. 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".."z" | "A".."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:

```
Z80 UP2down4
```

4.1.1.3. 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).

4.1.1.4. 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.

4.1.1.5. Unsigned numbers

These are *or* three types, integer, real and longreal:

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

4.1.1.5.1. 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** = 21783647.

Examples:

```
1066
0FFH
```

4.1.1.5.2. 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
```

4.1.1.5.3. 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
```

4.1.1.6. Character strings

A character-string is a sequence of zero or more 8-bit 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-character | apostrophe-image
apostrophe-image = "'"'
```

A character-string containing no string-elements is a null string; this is a Pro Pascal extension.

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'
```

4.1.2. Separators

These are of three kinds:

```
separator = space | end-of-line | comment
space = " "
comment = "{" any-sequence-of-string-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.

4.1.3. 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.)

4.1.3.1. 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 mdv1_typedefs }
```

causes the inclusion of the source file *mdv1_typedefs_pas*. Any spaces after the I are ignored, and the remainder of the comment is treated as a filename. The extension *_PAS* is supplied by the compiler.

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.

4.1.3.2. 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 }
```

4.2. Programs, Segments and Blocks

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

```
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.

4.2.1. 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.

4.2.2. Segments

```
segment = segment-heading "in 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.

```

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 or 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

```

4.3. Statements and Expressions

4.3.1. 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)

```

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

```

4.3.1.1. 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
```

4.3.1.2. 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~)J 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[1,j] := 0.0
p^next := NIL
```

4.3.1.3. 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], z)
```

4.3.1.4. GOTO statement

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

```
goto-statement : "GOTO" label
```

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

Example:

```
GOTO 999
```

4.3.1.5. 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
```

4.3.1.6. 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
```

4.3.1.7. Conditional statements

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

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

4.3.1.8. 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

```

4.3.1.9. CASE statement

```

case-statement = "CASE" case-index "OF"
    case-list-element { "in case-list-element"
        [ ";" "OTHERWISE" statement] [ "j" ] "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 base-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(13);
END

```

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

4.3.1.10. Repetitive statements

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

```

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

```


4.3.1.10.1. 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 > 1
```

4.3.1.10.2. 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 1 <= j DO  
BEGIN  
  j := 12*i;  
  i :=succ(i);  
END
```

Note the difference in behaviour compared with the **REPEAT** example above. 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.

4.3.1.10.3. FOR statement

```
for-statement = "FOR" control-variable ":=" initial-value  
                ( "TO" | "DOWNT" )  
                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 e_1 and e_2 are evaluated, and if $e_1 > e_2$ then nothing remains to be done; otherwise, e_1 is assigned to v , body is performed, v is compared with e_2 , and, for as long as it is not equal to e_2 , v is incremented and body is again executed.

When the **FOR** statement

```
FOR v := e1 DOWNT0 e2 DO
  body
```

is executed, the sequence of events is as follows. The expressions e_1 and e_2 are evaluated, and if $e_1 < e_2$ then nothing remains to be done; otherwise, e_1 is assigned to v , body is performed, v is compared with e_2 , and, for as long as it is not equal to e_2 , v is decremented and body is again executed.

Example:

```
FOR i := j TO 10 DO
  proc (1, 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 (1, j)
```

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

4.3.1.11. 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.

4.3.2. Expressions

Expressions possess a value, at run time, or 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 anyone 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.

4.3.2.1. 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.

4.3.2.1.1. 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
```

4.3.2.1.2. 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.

4.3.2.1.3. Indexed variable

```
indexed-variable = array-variable "[" index-expression {"", "index-  
expression"} "]" dynamic-string-variable ft[" 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 or the dynamic-string-type, as specified in its declaration.

4.3.2.1.4. 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
```

4.3.2.1.5. 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 pointer." ⁿ

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;
```

4.3.2.1.6. Buffer variable

```
buffer-variable = file-variable nAn  
file-variable = variable-access
```

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

Example:

```
input^
```

4.3.2.2. Unsigned constant

The second possible form of "factor" 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.

4.3.2.3. 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 (x..y)
```

4.3.2.4. 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

```
o <= ord(expression) <= 262127
```

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

The member-designator $\mathbf{x} \mathbf{..} \mathbf{y}$ represents the set of all values in the 3.2.2 Operators closed interval \mathbf{x} to \mathbf{y} ; if $\mathbf{x} > \mathbf{y}$, 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']
```

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

4.3.2.5. 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.

4.3.2.5.1. +

If \mathbf{t} 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).

4.3.2.5.2. -

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).

4.3.2.5.3. `*`

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 `I` is also used for set intersection (see 3 .2.2.3.3).

4.3.2.5.4. `/`

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.

4.3.2.5.5. `DIV`

`DIV` is the operation of integer division with truncation. Both operands, and the result, are integer-type. If $i \geq 0$ and $j > 0$, then the value of $i \text{ DIV } j$ is such that

$1 - j < (i \text{ DIV } j) \cdot j \leq i$ If $j = 0$, a run-time error occurs. If i and/or j are negative, the value of $i \text{ DIV } j$ is such that $ab.(i \text{ DIV } j) = ab.(i) \text{ DIV } ab.(j)$ and the sign of $i \text{ DIV } j$ is positive if i and j have the same signs and negative otherwise. For example: $7 \text{ DIV } 3 = 2$ $-7 \text{ DIV } 3 = -2$ $7 \text{ DIV } -3 = -2$ $-7 \text{ DIV } -3 = 2$

4.3.2.5.6. `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 $j \leq 0$, a run-time error occurs; otherwise, the value of

$i \text{ MOD } j$ is that one out of the sequence of values $(1 - (k * j))$, where k is any integer which is such that $0 \leq i \text{ MOD } j < j$ For example:

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

4.3.2.6. Boolean operators

4.3.2.6.1. `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`) .

4.3.2.6.2. `AND`

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

4.3.2.6.3. `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.

4.3.2.7. Set operators

If placed between two operands of set-type (see 6.1.2.3), `+` stands for the operation of set union.

This yields the set consisting of those elements that are present in either the first operand or the second operand. 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]
```

If placed between two operands of set-type, `-` represents the operation of set difference. This yields the set consisting of those elements of the first operand that are not also present in the second operand. 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 above, the value of the expression

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

is equal to

```
[Monday, Wednesday]
```

If placed between two operands of set-type, `*` represents "set intersection". This yields the set consisting of those elements that are present in both operands. 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 above, the value of the expression

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

is the empty set, `[]`.

4.3.2.8. Relational operators

4.3.2.8.1. `=` 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 long-real-type and this applies also to the operators in 3.2.2.~.2 and 3.2.2.~.3.

4.3.2.8.2. `<` 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 the lexicographic ordering of their constituent 8-bit characters and the same applies to the operators in 3.2.2.~.3.

4.3.2.8.3. `<=` 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 above, the expression

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

is true.

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 above, the expression

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

is true.

4.4. 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.

4.4.1. 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}
```

4.4.2. Definition of labels

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

Example:

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

4.4.3. 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
```

4.5. `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 = 3.14159265356979300;  
  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 = 2147463647;
```

4.6. 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
```

4.6.1. 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.

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

4.6.1.1. 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.

4.6.1.1.1. Real

real-type ; "**real**"

These items take on real values, which are signed floating-point values whose magnitude may range from approximately 5.9E-39 to 3.4E+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 above.

4.6.1.1.2. Longreal

longreal-type ; "**longreal**"

These items take on longreal values, which are signed floating-point values whose magnitude may range from approximately 1.1D-308 to 1.8D+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 above

The predefined type longreal is an extension to Standard Pascal.

4.6.1.1.3. 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.

4.6.1.1.4. 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);

4.6.1.1.5. Char

char-type = "char"

These take values which are any of the 256 8-bit characters. The ordinal value of the character therefore lies in the range 0.. 255.

4.6.1.2. 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)
```

4.6.1.3. 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
```

4.6.1.4. Structured types

The second class or "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. The maximum permitted length-specifier is 32767. Dynamic-strings are an extension to Standard Pascal.

4.6.1.5. 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**.

4.6.1.6. Record types

```
record-type = "RECORD" [field-list ] [-i] "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: string;
  age: 0..119;
  salary: integer;
CASE female: boolean OF
  true: (maidenname: string24);
  false: ()
END
```

4.6.1.7. 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 262121, or a subrange of any of these types.

Examples:

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

4.6.1.8. File types

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

A file type represents a sequence of components all of the same type, given by "type-denoter". It may be of any components 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 **eofln** (see 8.3.1.1).

Examples:

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

4.6.1.9. 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).

4.6.2. 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.

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 stringtype or vice versa, or **t1** and **t2** are both dynamic-stringtypes.

4.6.2.1. 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 stringtype or a dynamic-string-type.

4.7. Variable Declarations

Variables are declared in the variable-declaration-part or a block

```
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 or 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 or a program or segment. Using **COMMON**, rather than **VAR**, causes the names or 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 or 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: errfiletype;
VAR
  student, teacher, parent: person;
  attendance : 0..maxnumber;
  promised,
  empty: boolean;
  c1, c2: RECORD
    realpart,
    imagpart: real
  END;
  mark: (good, fair, indifferent);
```

4.8. 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.

4.8.1. 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 8.1.3.1 . The form most commonly used is the first, consisting of a heading and a block separated by a semicolon.

4.8.1.1. Procedure and function heading

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

4.8.1.1.1. 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
```

4.8.1.1.2. 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
```

4.8.1.1.3. 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).

4.8.1.1.4. 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.

4.8.1.1.5. 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;
```

4.8.1.1.6. 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.

4.8.1.2. 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 as-

signed 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;
```

4.8.1.3. 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 8.1.3.1). The **EXTERNAL** directive (not part of Standard Pascal) indicates that the definition is in another compilation-unit (see 8.1.3.2).

4.8.1.3.1. **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.

4.8.1.3.2. **EXTERNAL** declarations

An **EXTERNAL** declaration indicates to the compiler that a procedure or function is not defined (i.e. its body is not present) in the current compilation-input, but is to be found in a separately-compiled Pascal segment. It may also be a subprogram coded in Pro Fortran-77 (see Appendix E), or it may be in an assembler-coded module as described in

9.6 below. Names of **EXTERNAL** procedures are limited to 32 characters in the relocatable binary format, and must be distinct in these first 32 to avoid confusion during the link-edit process.

Examples of **EXTERNAL** declarations will be found in section 9.3.

4.8.2. Activation of procedures and functions

4.8.2.1. 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 6.2.3 below).

4.8.2.2. 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.

4.8.2.3. Actual parameters

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

4.8.2.3.1. 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.

4.8.2.3.2. **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)j
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.

4.8.2.3.3. 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
 - a they are both value parameters of identical type, or
 - b they are both variable (VAR) parameters of identical type, or
 - c they are both procedural parameters with compatible parameter lists , or
 - d they are both functional parameters with compatible parameter lists and identical result types.

4.8.3. Standard Procedures and Functions

The declarations or 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.

4.8.3.1. 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 9.2.

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.

4.8.3.1.1. `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.

4.8.3.1.2. **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).

4.8.3.1.3. **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.

4.8.3.1.4. **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.

4.8.3.1.5. **read**

In its basic form, **read(gf, varb1)** is equivalent to

```
BEGIN
  varb1 := gf;
  get(gf)
END
```

i.e. the current file element is assigned to **varb1** and the next element made accessible. The component type of **gf** must (for non-text files) be assignment-compatible with the variable **varb1**.

There are some additional facilities of **read**. In the first place, it may have a list of parameters (rather than the single **varb1**) 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, longreal or dynamic-string type may also be specified, and conversion from the external character format is performed automatically. The external representation in these cases must conform to the layout of integer, real, longreal or string 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)
```

4.8.3.1.6. 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 eofln(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
```

4.8.3.1.7. 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
<code>write('X')</code>	X	
<code>write('X':5)</code>	X	4 leading spaces
<code>write('ABC')</code>	ABC	
<code>write('ABC':5)</code>	ABC	2 leading spaces
<code>write(i:6)</code>	12345	1 leading space
<code>write(i)</code>	12345	default width = 11

<code>write(i:1)</code>	12345	left justified
<code>write(r)</code>	1.2344999E+02	default real format
<code>write(r: 10)</code>	1.234E+02	
<code>write(r:10:4)</code>	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.

4.8.3.1.8. `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.

4.8.3.2. `new` and `dispose`

These procedures are used to request space in the heap and to return the space when no longer needed. In the following, let `pt` be a pointer-type (see 6.1.3) and `t` the type with which it is associated, i.e.: to which it points.

`new(p)`, where `p` is a variable of type `pt`, allocates space in the heap for a variable of type `t` and sets `p` to point to it. The value of the new variable is undefined.

`dispose(q)`, where `q` is an expression of type `pt`, frees the space occupied by the referenced variable `q` of type `t` (cf. 3.2.1.1.~). No further reference may be made to the latter 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(q, 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/dispose` - and so on, if subvariants are even deeper nested.

Note 1. After `dispose(q)`, all pointer variables pointing to the referenced variable `q` become undefined.

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.

4.8.3.3. `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` on 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.

4.8.3.4. `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:

<code>trunc (5.2)</code>	gives 5
<code>trunc (5.7)</code>	gives 5
<code>trunc (-5.7)</code>	gives -5
<code>round (5.2D0)</code>	gives 5
<code>round (5.7D0)</code>	gives 6
<code>round (-5.7D0)</code>	gives -6

4.8.3.5. `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 in the range 0 thru 255 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`.

4.8.3.6. `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.

4.8.3.7. `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`).

4.8.3.8. `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
<code>sqrt(x)</code>	non-negative square root	<code>x < 0.0</code>
<code>sin(x)</code>	sine of <code>x</code> (<code>x</code> in radians)	<code>abs(x) > 32768.0</code> (<code>x</code> real) <code>abs(x) > 1.3D9</code> (<code>x</code> longreal)
<code>cos(x)</code>	cosine <code>x</code> (<code>x</code> in radians)	<code>abs(x) > 32768.0</code> (<code>x</code> real) <code>abs(x) > 1.3D9</code> (<code>x</code> longreal)
<code>exp(x)</code>	exponential of <code>x</code>	<code>x > 89.4</code> (<code>x</code> real) <code>x > 710.DO</code> (<code>x</code> longreal)
<code>ln(x)</code>	natural logarithm of <code>x</code>	<code>x < 0.0</code>
<code>arctan(x)</code>	principal value (radians) of arc-tangent of <code>x</code>	

4.8.3.9. `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.

4.8.3.10. 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).

4.8.3.10.1. `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 32767 characters.

4.8.3.10.2. `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:

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

4.8.3.10.3. `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,S)
```

leaves `sv` holding 'PQRSXYTUV'.

4.8.3.10.4. `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 `av`, leaving 'PQRUV'. It is an error if the substring defined by index and count extends beyond the current limits of the contents of `stringvar`.

4.8.3.10.5. `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.

4.8.3.10.6. 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.

4.8.3.10.7. 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.)

4.9. Implementation-dependent Aspects

4.9.1. Pascal Files and QDOS

4.9.1.1. 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.

4.9.1.2. 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 QDOS 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 QDOS filename. The variable is said to be "assigned" to the QDOS 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 by the run-time system on exit from the block in which the file is declared, whether by normal procedure exit or by jumping out from the procedure; and all open files are closed by the system at program termination. However, an explicit "close" procedure is also provided, as in certain cases it may be desirable to release limited system resources (limit on number of open files, for example).

When it comes into existence, each file variable is given a default assignment (QDOS filename with which it is associated) which may be changed by means of the procedure "assign", see below, before any reset, rewrite, update or append 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 stan-

Standard files input and output are to the program's window, unless redirected at execution time (see Part III, section 4.2). All other nameless files are allocated to a filestore device (disc or microdrive). The name of this device is, by default, requested at execution time, but can be configured by the user before execution, as described in Part III, section 5.2.1. These workfiles are given names by the run-time library software of the form:

```
<device>_PAS$_<job id>_<sequence no>
```

where <device> is the chosen/default device name, e.g. mdv2, <job id> is the QDOS job id for the program (8 hex characters), and <sequence no> is a unique sequence number, starting at 0001 and incrementing by 1 for each workfile opened (4 hex characters).

Such workfiles are erased by the run-time system at program termination, unless renamed.

4.9.1.3. File formats

4.9.1.3.1. Text files

A Pascal text file (on disc or microdrive) follows the conventions of QDOS text files: lines are terminated by <LINEFEED>.

When reading such a file from a filestore device (disc or microdrive), end-of-file will be set either by encountering the physical end of the file or by a line consisting of <CTRL-Z> (decimal code 26) followed by <LINEFEED> (decimal code 10).

To signal end-of-file when inputting from the keyboard, a separate line consisting of <CTRL-SHIFT-QUOTE> followed by <ENTER> must be supplied, i.e. decimal codes 130 followed by 10.

4.9.1.3.2. Non-text files

Non-text files consist of a sequence of fixed-length elements. The element size is deducible from the file's declaration, and can be up to a maximum of 32767 bytes.

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

The file elements are stored adjacent to one another, with no control information or unused bytes. During sequential processing of files with a small element size, the elements are blocked up by the Pascal run-time system to improve processing efficiency.

To read a non-text file which was not produced by Pro Pascal, the most general method is to assign it to a Pascal file of 1-byte element size. However, if it has a known structure, some larger element size may simplify the processing. The file size must, however, be an exact multiple of the element size.

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.). This applies to both text and non-text files. However, the programmer does not need to be aware of this, and there is no effect on the operation of programs written according to the standard rules. In particular, exactly the same coding (using readln, eoln, etc., works whether the Pascal file is assigned to an interactive device such as the console or to a disc file (this is the whole point of the "lazy i/o" technique).

4.9.1.4. Additional procedures and functions

To enhance the file handling functionality, a number of extra standard procedures and functions are included in the Pro Pascal system. They are all known to the compiler, and so must not be declared as EXTERNAL in a user program. (In fact, if the names are so declared, or re-declared in

any other way, then these routines become inaccessible.)

These extra file-handling routines are described in sections 9.2.1 thru 9.2.13 below.

There are also some extra file handling procedures and functions which are not known to the compiler and which must therefore be declared as **EXTERNAL**. These are described in section 9.3 below.

4.9.2. Additional standard procedures

In the Pro Pascal system, a number of extra procedures and functions are provided in addition to those described in section 8.3. These are all known to the compiler, and so do not have to be declared in the program in any way.

An important group of these extra procedures, and the first to be described in the following subsections (9.2.1 thru 9.2.13), are to do with file handling. These are:

- **PROCEDURE assign** (VAR f: genfile; name: DOSname);
- **PROCEDURE update** (VAR f: ntfile);
- **PROCEDURE seek** (VAR f: ntfile, elnumber: integer);
- **FUNCTION position** (VAR f: genfile); integer;
- **PROCEDURE close** (VAR f: genfile);
- **PROCEDURE erase** (VAR f: genfile);
- **FUNCTION fstat** (name: DOSname): boolean;
- **FUNCTION checkfd** (name: DOSname): boolean;
- **PROCEDURE append** (VAR f: genfile);
- **PROCEDURE rename** (VAR f: genfile; name: DOSname);
- **PROCEDURE ramfile** (VAR f: text);
- **PROCEDURE echo** (VAR f: text; onoff: boolean);
- **FUNCTION handle** (VAR f: genfile): integer;

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. "DOSname" is any string or dynamic-string type, the associated actual parameter being an expression representing a QDOS file or device name.

4.9.2.1. assign

A file is assigned to a QDOS file or device by a call of the procedure assign. The DOSname parameter may be the name of a microdrive or disc file (e.g. 'MDV2_DATA', 'FDK1_PRIM_PRN') or a device name. Device names recognised by the run-time software are those beginning with any of the following 3-letter combinations:

- **CON** (console),
- **SCR** (screen),
- **SER** (serial port/printer)

A textfile assigned to CONxxxx and used for input works on a line-at-a-time basis. The operator may backspace and make corrections until <ENTER> is pressed, when the complete line is made available to the program. This is the default arrangement with the standard file "input".

Note that **reset** or **rewrite** (or **update** or **append**) must be called following **assign** before any reads or writes are made to a file.

Example:

```
assign(f, 'SER');  
rewrite(f);  
writeln(f, 'This will appear on the printer');
```

4.9.2.2. update

The procedure **update** is provided as an alternative to **reset/rewrite** to enable a file to be both read from and written to. (**Rewrite**, on the other hand, erases the contents of an existing file.)

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** (for example) to position to the first empty position and then writing sequentially.

Example:

```
assign(f, 'MDV2_DATA');  
update(f) ;  
seek(f,i);  
write(f,x);
```

4.9.2.3. 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 like an array, with "index" values starting at zero.

To read a file randomly, **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 both read and write a file randomly, **assign** and **update** are called. As in the case of **reset**, **seek** can be used to make the desired file element accessible (via **f** and so on), and then the modified element can be written back to the file using **put** (which then advances **f** to the next file element).

Example:

```
assign(f, 'FLP1_DATAFILE');  
update(f) ;  
seek(f,i);  
f^:= f^ + 2;  
put(f);
```

4.9.2.4. position

Integer function **position**(**f**) returns the number of the element of file **f** which is accessible as **f**[^]. The numbering is from zero (as for **seek**). The value -1 is returned if the file is not currently open.

4.9.2.5. close

Files are closed automatically at completion or execution of the program, or in the case of local files on exit (either normal or via a non-local **goto**) from the procedure in which they are declared. To close a file explicitly, a "**close**" procedure is available. Example:

```
close(outfile);
```

4.9.2.6. erase

When a Pascal file has been assigned to a QDOS disc or microdrive file, the file may be erased by calling the Pro Pascal procedure **erase**. Example:

```
erase(workfile);
```

4.9.2.7. fstat

The boolean function **fstat** has as parameter a string (constant, variable, or expression) containing a QDOS file or device name. **fstat** returns the value true if the file exists, false if there is no such file or if the string is not a correct QDOS filename. Examples:

fstat(' _ ') returns false (bad format)

fstat('MDV1_NIM_TXT') returns true if a file NIM_TXT is present on drive MDV2, false otherwise

fstat('SER1') returns true (valid device name)

4.9.2.8. checkfn

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

```
OK := checkfn(filename);
```

4.9.2.9. 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**. Example:

```
assign(outfile,s);
append(outfile);
writeln(outfile, 'Another line of text');
```

4.9.2.10. rename

This procedure has as parameters a file and a filename (**DOSname**). The file must already be con-

nected to a filestore file. The name of the filestore file is changed to **DOSname**, which must be a correctly-formed QDOS file name. If the file is currently open, it is closed. If a file with that name already exists, it is erased. A device identifier must be included in **DOSname**, and it must match the existing device. After rename, the file remains available for use by the program, but **reset**, **rewrite**, update or **append** must be called before any further reading or writing can be done. Example:

```
assign(f, 'FDK1_DATA');  
rename(f, 'FDK1_DATA_BAK');
```

Note: this procedure depends on certain extensions to QDOS, such as those provided in the Toolkit or in the ROMs of some manufacturers' disc drives. If these extensions are not present at run time, rename will not work.

4.9.2.11. ramfile

This procedure has one parameter, which must be a textfile. It is called in place of "**assign**", and causes the file to be assigned to a workfile in memory (a "*silicon file*"), and **rewrite** must then 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 accesses. The length of the file is limited only by the amount of heap space available.

Example:

```
ramfile(work);  
rewrite(work);  
writeln(work, x);
```

4.9.2.12. echo

The parameters are a file (which must be a textfile) and a boolean "onoff". The file must have been assigned to a filestore file / ramfile, and then set for output by **rewrite** (or **append**). 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.

Example:

```
assign(output, 'MDV1_PAYFILE_LOG');  
rewrite(output);  
echo(output, true);  
writeln('Start of payfile');
```

4.9.2.13. handle

This function has as parameter a file, and returns an integer result which is the 32-bit QDOS channel ID for the file, or -1 if the file is not currently open. The channel ID is needed for some QDOS system calls (see 9.3.12).

Example:

```
assign(f, name);  
rewrite(f);  
i := handle(f);
```

4.9.2.14. 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 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);
```

4.9.2.15. getcomm

The procedure **getcomm** may be called at any time during execution of a program to obtain the program option string.

The argument to **getcomm** can be of any type, but should be long enough to contain the program option string, otherwise trailing characters will not be transferred. The variable is formatted like a string, beginning with a 2-byte count followed by the actual characters.

For example, if a program includes

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

then if the program is executed with an option string 'SI_PAS', the string **fnames** will be set to 'SI_PAS', with a length of 7.

4.9.2.16. sizeof

The integer function **sizeof** is a notation for obtaining the storage occupied by a data type, for example a record. No run-time computation is involved. The parameter must be a type-identifier, and the value returned is in bytes. For example, the value 8 would be returned by

```
sizeof (longreal)
```

4.9.2.17. addr

Function **addr** has a parameter which is a variable-access, and returns an integer result. This is the absolute machine address of the variable.

A particular use for this function is in supplying address-type parameters to some of the QDOS system calls (cf. 9.3.12) . Example:

```
i := addr(nameptr^);
```

4.9.2.18. peek

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

```
b := peek(12H);
```

4.9.2.19. 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. For example:

```
poke(12H, b);
```

4.9.3. Library facilities

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

4.9.3.1. memavail

Once a number of **new** and **dispose** operations have been performed, the heap becomes in general a mixture of allocated and free areas. The function **memavail** returns an integer value which is the size in bytes of the largest such free area. (It is thus an underestimate of the total amount of free space available.)

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

4.9.3.2. rand and seed

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;
```

The procedure **seed** enables the 4-byte integer which controls the generation of the next "**rand**" value to be altered. It is declared as

```
PROCEDURE seed(seedvalue: integer); EXTERNAL;
```

(If no call of **seed** is made, the pseudo-random number sequence is initialised with a value of **3CA0F712H**.)

4.9.3.3. consingle

This function waits (with a flashing cursor) for the next character from the keyboard, and then echoes it to the program's standard window. It is declared as

```
FUNCTION consingle: char; EXTERNAL;
```

4.9.3.4. consilent

This function waits (with a flashing cursor) for the next character from the keyboard, and returns it without echo. It is declared as

```
FUNCTION consilent: char; EXTERNAL;
```

4.9.3.5. prompt

This function is provided as a simple way of a program outputting a message to the console asking for a Y/N response. It is declared as:

```
FUNCTION prompt(s; string); boolean; EXTERNAL;
```

The string *s* is output to the console with "1" appended automatically. The function result is true if the user types Y or y. false if he types N or n (all other keys being ignored).

4.9.3.6. ownerr

This procedure is provided to enable the user program to perform its own exception handling, as an alternative to the normal reporting of run-time errors. The procedure **ownerr** installs a procedure nominated by the program as its *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 in the declaration of **ownerr**

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

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 5 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 from the handler 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. In the case of stack overflow (error S), recovery is not feasible; this error is not passed to the handler, but causes termination of the program with a message in the normal form.

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.

4.9.3.7. textnote and textpoint

Function **textnote** and procedure **textpoint** allow random access to textfiles. They are explicitly declared thus:

```
FUNCTION textnote (VAR f: text): integer; EXTERNAL;
```

```
PROCEDURE textpoint (VAR f : text; position: integer); EXTERNAL;
```

The file parameter *f* must be a disc or microdrive textfile that has been opened for reading by **reset**. The idea is to call **textnote** at chosen points in the course of sequentially reading a text file, and save the returned values (they represent the "position" of the file at that point). Subsequently, such a value can be passed as the second parameter to **textpoint**, which will re-position the text file to the point at which the earlier call to **textnote** was made.

As an example, suppose a text file contains some 200 error messages, one per line, ordered in ascending error message number. Then as part of an initialisation process, the file can be read sequentially and function **textnote** called every tenth line (say). The values can be stored in an integer array of about 20 elements, thereby effectively creating a simple index to the file. Subsequently, procedure **textpoint** can be used to greatly speed up access to any particular error message, by positioning the file at the next lower error number noted previously and then reading sequentially at most 9 lines before finding the one required.

As an embellishment to this example, the error message index created by **textnote** could be written to a permanent file, and read directly from this file on subsequent occasions.

4.9.3.8. execprog

Execprog allows a user program to dynamically execute other separately linked Pascal programs.

The currently executing Pascal program, the "parent", uses **execprog** to execute a separately linked program, the "child". A string, the "program options", can be specified for passing to the child, which can obtain the string by means of procedure **getcomm** (see 9.2.15).

While the child program runs, the parent program is suspended, and only continues when the child terminates. The child itself may cause child programs of its own to be executed in the same way. The parent program and each child program run by it are separate QDOS jobs, these jobs together forming a dependent job tree.

When the child terminates, a numeric return code is passed back to its parent (see 9.3.9).

Execprog is declared as

```
TYPE
    string136 = string[136] ;

PROCEDURE execprog (command: string136; VAR returncode: integer);
    EXTERNAL;
```

Execprog requires the following parameters

- A "command string" in the form '<childprog_file>[<prog_options>]'

Examples:

The command string '*MDV1_XREF_BIN*' specifies program *XREF_BIN* on microdrive *MDV1*, with no command tail.

The command string '*FDK2_SIZE_BIN ALPHA,9.2*' specifies program *SIZE_BIN* on disc drive *FDK2* and a program options string of *ALPHA,9.2*.

The command string size is limited to 136 characters, and the program options to 126 characters. If the program options string exceeds 126 characters, it will be truncated to this length.

- An integer variable that will receive the child's return code when it terminates (see 9.3.9).

Example:

A parent program could include the following statements

```
VAR
    rc: integer;

execprog ('MDV1_HOM_BIN PARK',rc);
CASE rc OF
    ...
```

The statements cause program *HOM_BIN* to be loaded from drive MDV1 and executed. A call of **getcomm** in program *HOM* will yield the 5-character string 'PARK'. When *HOM* terminates, the return code it sets is stored in variable rc, which the parent then tests.

A hierarchy of programs can be executed in this manner, as in the following example.

IA| A is executed by operator command

|B| B is executed by A

ICI C is executed by B

Program A is loaded and executed by an operator command, and uses **execprog** to load and execute B. B, in turn, uses **execprog** to load and execute C. Thus when C has control, all three programs are in memory. A and B are known as parent and child programs respectively, as are B and C.

When C terminates, it is deleted from memory and control returns to B, which then continues execution. When B terminates, it is deleted from memory and control returns to A, which then continues execution. When A terminates, control passes back to the operating system in the usual way.

This example demonstrates that at each level of the program hierarchy, there is just one program loaded. Thus suppose for example that B executes further programs D and E after C, then D and E are successively loaded, executed and deleted from memory.

In creating trees of executing programs in this way, the user must not of course exceed the total amount of memory available for user programs. To make maximum use of this memory, the Linker DATA option and/or the SETSTACK utility for selecting stack sizes (see Part III) may be found useful.

4.9.3.9. exitprog

Exitprog is declared as

```
PROCEDURE exitprog (returncode: integer); EXTERNAL;
```

Exitprog may be used by any program to terminate its execution at any time, and to pass a return code back to the calling parent program (or the operating system). If a program terminates normally without calling exitprog, a return code of zero will be passed back.

The user-specified return code should be in the range 0..127. In addition, various negative return codes can be generated, as follows:

-1	child terminated by run-time error
-2	error in format of command string
-3	child program file could not be opened, or error reading child program's file header

-4	child program file not marked as executable
-5	failure to load child program file
-6	child program incompatible with PRL software
-7	insufficient memory to run child program
-8	error during initialization of child program
-9	no PRL installed, or PRL is corrupt
-10	error opening/reading/writing child program's window
-11	only one channel passed to child program
-12	error when linking child program
-13	unable to create a job to run child program

For an explanation of run-time errors, see Appendix C for further details of the other errors listed above, see Part III section 4.~.

Example:

```
exitprog (8)
```

causes the issuing program to terminate with a return code of 8.

4.9.3.10. date

This procedure is declared as:

```
PROCEDURE date (VAR year,month,day: integer); EXTERNAL;
```

and gives the current date in binary.

4.9.3.11. time

This procedure is declared as:

```
PROCEDURE time (VAR hours,mins, seconds,hundredths: integer);  
EXTERNAL;
```

and gives the current time in binary.

4.9.3.12. qtrap

This procedure is declared as:

```
PROCEDURE qtrap(trapnum: integer; regrec: regspec); EXTERNAL;
```

and provides an interface enabling QDOS TRAPs to be issued by a Pascal program. The first parameter is truncated to 4 bits to derive the trap number, in the range 0..15. The second parameter is the name of a record variable which has the layout given in the supplied "include" file *TRA-PREG_PAS*.

The user program has the responsibility for setting all necessary parameters required by the QDOS call being issued, including of course the trap number. The **qtrap** library procedure then loads the machine registers with the values specified in the **regrec** record and issues the TRAP machine instruction. On return, the machine registers are stored back into **regrec**, where they can be examined by the user program.

Note: Care should be taken that the QDOS trap call does not interfere with the operation of the Pascal run-time routines, in particular those concerning files.

An example program illustrating use of the **qtrap** procedure will be found among the source files issued with the software.

4.9.3.13. mode

Declared as:

```
PROCEDURE mode(highres: boolean); EXTERNAL;
```

Related SuperBASIC keyword: MODE

Sets the QL screen to either high resolution 4-colour mode (**highres** = true) or low-resolution 8-colour mode (**highres** = false). For example, the following procedure call will set the display to 4-colour mode:

```
mode(true);
```

4.9.3.14. Window routines

This group of procedures enables the programmer to associate a screen output window with a Pascal text file, and to set and inquire about its attributes. Where there is a related SuperBASIC command, this is stated, and the QDOS documentation for that command should be consulted for more details.

All the routines in this section and in sections 9.3.15 thru 9.3.17 operate on "window"s. They must all be declared as **EXTERNAL** (so this will be omitted from now on in the procedure declarations given here). Note that the window must be specified explicitly in each procedure call - there is no equivalent of the SuperBASIC idea of the "default channel". A program may have any number of windows open at one time.

Before calling any other procedure referencing a window, the window must first be opened, *i.e.* associated with a Pascal file variable, by a call of **wopen**.

It is possible to output text to a window using **write** statements, but since the window is for output only, read statements are impermissible.

4.9.3.14.1. wopen

Declared as:

```
PROCEDURE wopen(VAR w: text; width,height,Xorigin,Yorigin: integer)
```

Related SuperBASIC keyword: OPEN

Opens a screen output window. The last four parameters are in pixel units. For example, the call

```
wopen (w, 448, 180, 32, 16)
```

associates the Pascal file variable **w** with the QL console device **SCR_448x180a32x16** (which is in fact the default SCR-device).

4.9.3.14.2. window

Declared as:

```
PROCEDURE window(VAR w: text; width,height,Xorigin,Yorigin: integer)
```

Related SuperBASIC keyword: WINDOW

Allows the user to change the size and/or position of a window. The last four parameters are in pixel units.

Example:

```
WINDOW (lwind, 30, 30, 10, 10)
```

4.9.3.14.3. wstatc

Declared as:

```
PROCEDURE wstatc(VAR w: text; VAR width, height: integer;  
                 VAR Xcursor,Ycursor: integer)
```

Related SuperBASIC keyword: none

An inquiry procedure, which returns the window size and cursor position, both in terms of character coordinates. The last four parameters must be integer variables/array-elements. Note that the top left cursor position is 0,0.

Example:

```
wstatc(w, iw, ih, ix, iy)
```

4.9.3.14.4. wstatp

Declared as:

```
PROCEDURE wstatp(VAR w: text; VAR width,height: integer;  
                 VAR Xcursor,Ycursor: integer)
```

Related SuperBASIC keyword: none

An inquiry procedure, which returns the window size and cursor position, both in terms of pixel coordinates. The last four parameters must be integer variables/array-elements. Note that the top left cursor position is 0,0.

Example:

```
wstatp(iwind, iw, ih, ix, iy)
```

4.9.3.14.5. recol

Declared as:

```
PROCEDURE recol(VAR w: text; c1,c2,c3,c4,c5,c6,c7,c8: integer)
```

Related SuperBASIC keyword: RECOL

Changes the colour values for this window. Each of the last 8 parameters must be integer expressions in the range 0 (= black) thru 7 (= white).

Example:

```
recol (iwind, 2,3,4,5,6,7, 1,0)
```

4.9.3.14.6. border

Declared as:

```
PROCEDURE border(VAR w: text; width,colour: integer)
```

Related SuperBASIC keyword: BORDER

Adds to the window a border of the specified size and colour. "**width**" is in pixels. "**colour**" is in the range 0 to 255; considered as an 8-bit byte, the bottom three bits define the base colour (0 = black thru 7 =white), the next three bits give the exclusive OR of this base colour and the stippling colour (if these three bits are zero, the colour is therefore solid), and the top two bits select one of the four stipple patterns. The special value 80H for "colour" will produce a "transparent" border, i . e. the previous border is unchanged.

Example:

```
border(w, i1-i2, 40H+j*newcol)
```

4.9.3.14.7. ink

Declared as:

```
PROCEDURE ink(VAR w: text; colour: integer)
```

Related SuperBASIC keyword: INK

Sets the ink colour for the given window. The second parameter should be in the range 0 to 255, the associated colour being as described above.

Example:

```
ink (iw3, icolor[j])
```

4.9.3.14.8. paper

Declared as:

```
PROCEDURE paper(VAR w: text; colour: integer)
```

Related SuperBASIC keyword: PAPER

Sets the paper colour for the given window. The second colour parameter should be in the range 0 to 255, the associated colour being as described above

Example:

```
paper(w, icol)
```

4.9.3.14.9. strip

Declared as:

```
PROCEDURE strip(VAR w: text; colour: integer)
```

Related SuperBASIC keyword: STRIP

Sets the strip colour for the given window. The second parameter should be in the range 0 to 255, as above.

Example:

```
strip (w, 7-icol)
```

4.9.3.14.10. block

Declared as:

```
PROCEDURE block(VAR w: text; width,height,Xorigin,Yorigin: integer;  
                colour: integer)
```

Related SuperBASIC keyword: BLOCK

Fills a block of the specified size and position with the given colour. The middle four parameters are in pixel units. The meaning of "colour" is as described in 9.3.14.6.

Example:

```
block (iwind, 10,10, 5,5, 7)
```

4.9.3.14.11. cls

Declared as:

```
PROCEDURE cls(VAR w: text; part: integer)
```

Related SuperBASIC keyword: CLS

Clears the window to the current **PAPER** colour. The second parameter determines how much of the window area will be cleared, according to the scheme:

part = 0	whole window
part = 1	top excluding the cursor line
part = 2	bottom excluding the cursor line
part = 3	whole of the cursor line
part = 4	right end of cursor line (including cursor)

Example:

```
CLS (i, 0)
```

4.9.3.14.12. pan

Declared as:

```
PROCEDURE pan(VAR w: text; distance,part: integer)
```

Related SuperBASIC keyword: PAN

Pans the window "distance" pixels to the right; if distance is negative, pans to the left. The last parameter determines how much of the window area will be panned, according to the scheme:

part = 0	whole window
part =3	Whole of the cursor line
part = 4	right end of cursor line (including cursor)

Example:

```
pan (w, xNew - xOld, 4)
```

4.9.3.14.13. scroll

Declared as:

```
PROCEDURE scroll(VAR w: text; distance, part: integer)
```

Related SuperBASIC keyword: SCROLL

Scrolls the window "distance" pixels downwards; if distance is negative, scrolls upwards. The last parameter determines how much of the window area will be scrolled, according to the scheme:

part =0	whole window
part = 1	top excluding the cursor line
part =2	bottom excluding the cursor line

Example:

```
scroll (w, yNew -yOld, 0)
```

4.9.3.15. Print style routines

This group of procedures enables the programmer to influence the style in which text is output to a window. In each case, there is an associated SuperBASIC command, and the QDOS documentation should be consulted for further details.

4.9.3.15.1. csize

Declared as:

```
PROCEDURE csize(VAR w: text; width,height: integer)
```

Related SuperBASIC keyword: CSIZE

Sets the character size for the window. "**width**" should be in the range 0 to 3, and "**height**" should be 0 or 1.

Example:

```
csize (lwlnnd, 0, 0)
```

4.9.3.15.2. flash

Declared as:

```
PROCEDURE flash(VAR w: text; onoroff: boolean)
```

Related SuperBASIC keyword: FLASH

Sets the FLASH state on or off. If the second parameter is true, the flash state is set on, else off.

Example:

```
flash (w, true)
```

4.9.3.15.3. under

Declared as:

```
PROCEDURE under(VAR w: text; onoff: boolean)
```

Related SuperBASIC keyword: UNDER

4.9.3.15.4. over

Declared as:

```
PROCEDURE over(VAR w: text; mode: integer)
```

Related SuperBASIC keyword: OVER

The second parameter selects the type of over-printing required, according to the scheme:

mode = -1	print in INK over previous screen contents
mode = 0	print INK on STRIP
mode = 1	print in INK on transparent STRIP

Example:

```
over (w, 1)
```

4.9.3.16. Cursor positioning routines

This group of procedures enables the programmer to position the cursor within a window. In each case, there is an associated SuperBASIC command, and the QDOS documentation should be consulted for further details.

4.9.3.16.1. atc

Declared as:

```
PROCEDURE atc(VAR w: text; line,column: integer)
```

Related SuperBASIC keyword: AT

Positions the cursor in the window using character coordinates, with 0,0 corresponding to the top left corner of the window. Example:

```
atc (iwind, 0, 0)
```

4.9.3.16.2. atp

Declared as:

```
PROCEDURE atp(VAR w: text; Xpos,Ypos: integer)
```

Related SuperBASIC keyword: CURSOR (with 2 parameters)

Positions the cursor in the window using pixel coordinates, with 0,0 corresponding to the top left corner of the window. Example:

```
apt (iwind, 20, 30)
```

4.9.3.16.3. atg

Declared as:

```
PROCEDURE atg(VAR w: text; Xorigin,Yorigin: real; Xoffset,Yoffset: integer)
```

Related SuperBASIC keyword: CURSOR (with 4 parameters)

Positions the cursor in the window, using a combination of graphics coordinates (the second and third parameters, which must be real expressions) and pixel offsets (the fourth and fifth parameters, which must be integer expressions).

Example:

```
atg (iwind, gx, gy, 0, 0)
```

4.9.3.17. Graphics drawing routines

This group of procedures enables the programmer to output points, lines and arcs to a window. The facilities provided correspond to those of the SuperBASIC graphics procedures, and in each case the corresponding SuperBASIC keyword is given. The QDOS documentation should be consulted for further details.

4.9.3.17.1. fill

Declared as:

```
PROCEDURE fill(VAR w: text; onoff: boolean)
```

Related SuperBASIC keyword: FILL

Turns the graphics fill on or off. If the second parameter is true, fill is set on, else off.

Example:

```
fill (w, true)
```

4.9.3.17.2. scale

Declared as:

```
PROCEDURE scale(VAR w: text; scalefactor,Xorigin,Yorigin: real)
```

Related SuperBASIC keyword: SCALE

Allows the scale factor used by the graphics procedures (point, line, arc, circle and ellipse) to be altered.

Example:

```
scale (w, 0.5 , 0.1, 0.1)
```

4.9.3.17.3. point

Declared as:

```
PROCEDURE point(VAR w: text; X,Y: real)
```

Related SuperBASIC keyword: POINT Plot a point at the specified position relative to the graphics origin.

Example:

```
point(w, x, 0.0)
```

4.9.3.17.4. line

Declared as:

```
PROCEDURE line(VAR w: text; Xfrom,Yfrom,Xto,Yto: real)
```

Related SuperBASIC keyword: LINE

Draw a straight line between two points, whose locations are specified in absolute graphics coordinates.

Example:

```
line (iwind, 0.0, 0.0, xdest, ydest)
```

4.9.3.17.5. arc

Declared as:

```
PROCEDURE arc(VAR w: text; Xfrom,Yfrom,Xto,Yto: real; angle: real)
```

Related SuperBASIC keyword: ARC

Draw an arc of a circle between two points, whose locations are specified in absolute graphics coordinates. The last parameter gives the angle subtended by the arc, in radians. The following will draw a semi-circle, for example:

```
arc (w, XL, YL, XR, YR, 3.141593)
```

4.9.3.17.6. circle

Declared as:

```
PROCEDURE circle(VAR w: text; Xcentre,Ycentre,radius: real)
```

Related SuperBASIC keyword: CIRCLE (with 3 parameters)

Draw a circle whose centre and radius are given in graphics coordinates.

Example:

```
circle (w, XCEN, YCEN, SIN(GRAD))
```

4.9.3.17.7. ellipse

Declared as:

```
PROCEDURE ellipse(VAR w: text; Xcentre,Ycentre,majoraxis: real;  
                  eccentricity,angle: real)
```

Related SuperBASIC keyword: CIRCLE (with 5 parameters)

Draw an ellipse whose centre and major axis are given in graphics coordinates, with a specified eccentricity and orientation. The "eccentricity" is the ratio between the major and minor axis. The "angle" is the orientation of the major axis relative to the vertical, in radians.

Example:

```
ellipse (w, xeen, yeen, 12.0, 0.5, 0.0)
```

4.10. Storage allocation

4.10.1. Overall layout

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

- Program code.
- Constants (literals).
- Static data areas.
- **COMMON** data areas.
- Stack/work area.

All program variables declared at the outer level, whether in **COMMON** or not, are allocated static data space.

In the object code from the compiler, the static data for each module is located on a word boundary. The stack is kept word-aligned throughout execution of the program.

The result of compiling a **PROGRAM** or **SEGMENT** is a module in Sinclair relocatable object format which consists of a number of *sections*. There are up to four sections generated:

- **.CODE**, which contains object code and constants (integer, real, character, etc.). The code generated for the body of anyone procedure cannot exceed 32K bytes.
- **.ATAB**, which contains control information enabling run-time access to **COMMON** blocks and linked-in routines such as other program units and library procedures
- **.INIT**, which contains control information (used by Pro Fortran-77 modules) enabling the run-time initialisation of **COMMON** blocks.
- **.NAME**, which by default contains just the program's name. If the /N compile-time option is selected, it also contains the names of files and procedures compiled, for utilisation in the event of

a run-time error, when producing a procedure call trace-back.

Each **COMMON** variable is converted into a **COMMON** section with the same name as the common Variable. Note that a program file contains no **COMMON** areas within it, since the **COMMON DUMMY** directive is used at link-time. The actual **COMMON** data areas used by a program are created at run-time.

The user stack pointer SP is set to the highest address in the workspace area plus one, and the stack "grows" from higher to lower addresses. The heap grows from lower to higher addresses. If at run time the library finds that the stack and the heap are about to collide, execution of the user program is terminated. The program must be re-run with a larger amount of workspace (see Part III, section 5).

Depending on a program's requirements, extra memory areas may be allocated dynamically at run-time by the library heap manager.

The detailed layout of a program in memory is as follows:-

There are two separately allocated memory areas:

- the executable code image and heap/stack area
- the **COMMON** areas and static data.

(increasing addresses -->)

1-----+-----1

i

1 Code Image Heap--> <--Stack

i ,

1-----+-----1

Initial user SP

1-----1

i i i COMMON areas + static data 1 i i

1-----1

In addition, there is one further area allocated dynamically by the library, containing global run-time information shared by the topmost job and its dependent child jobs. The size of this area is a few

hundred bytes.

(The description above relates to an individual object program, which may be run as a job by operator command or as a child by a parent program. The Prospero Resident Library, or PRL, is a fixed area of code, of rather less than 16K bytes, which is distinct from the areas just described. Only one copy of the PRL is in the machine at one time _ it is typically in ROM, in fact -and it is shared between any active jobs.)

4.10.1.1. 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 high-low arrangement is followed.
4 bytes	integer, and subranges of integer outside word subrange. The bytes are arranged most-significant to least-significant in ascending addresses, in the normal MC6 <i>BODO</i> long-word memory format

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, biased 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 , biased 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 64-bit

3F800000H 3FF0000000000000H

Pointers occupy 4 bytes.

set variables occupy from 1 to 32766 bytes, depending upon limit of the range of the base type in the declaration (SET

the OF

upper 0..7,

for instance, requires 1 byte, while SET OF char occupies 32 bytes and SET OF 0 .. 262127 is the maximum 32766 bytes). Within each byte, ascending set elements *are* represented in increasingly significant bit-positions. As an example, the set-constructor [2,3] is held as the one-byte value 0CH. 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 for this machine).

A variable of type string[n] occupies (n+2) bytes, the lowest addressed word containing the length of the string (a value in the range 0..32767).

Record layouts are simply derived by placing the component fields in ascending addresses, with no unused bytes between fields.

9. 5 Interfacing to assembler

9.5.1 Use of assembly language

To use machine features not available through the Pascal language, for example interrupts, routines may be written in assembly language and combined with the generated code during the link-edit process.

9.5.2 Choice of assembler

The Pascal compiler generates relocatable object code. Assembler language modules may be processed by any assembler which generates the same format, and linked with the other components of the program. In particular, the GST Macro Assembler will be found satisfactory.

XDEF/XREF linkage

9.5.3.1 Calling assembler from Pascal

An assembler-coded routine can be called from Pascal in the normal way by a procedure-call, or may be invoked as a function if it returns a value. In the assembler module, the name is quoted in an XDEF directive, or made global in some equivalent way. More than one routine can be in the module. Return is made by an RTS instruction or equivalent.

To make these remarks more specific, consider the Pascal fragment:

```
PROCEDURE suba(VAR x,y: real); EXTERNAL;
```

```
BEGIN
```

```
suba(xcoord.ycoord);
```

```
END.
```

The assembler code for a routine SUBA which is called in this way should be structured as shown below .

.....0•0•0.....

• Pascal calling assembler

'0 • 0.00 0.0 ...

XDEF SUBA

SECTION • CODE

SUBA MOVEA.L ~(SP),AO Get address of parameter y MOVEA.L B(SP),AI Get address of parameter x

MOVE.L (SP)+,AO Caller's link ADDA.L #B,SP Remove SUBA's parameters from stack JMP (AO) Return to Pascal

9.5.3.2 Calling Pascal from assembler

A Pascal procedure or function can be called from assembler code. The subprogram name must be quoted in an XREF directive (or equivalent), and called by a BSH.L instruction (this assumes that the distance between the BSH and the called routine is expressible as a long BSR operand; if this is not the case, another technique must be used which is explained below). If the subprogram requires parameters (see 9.5.6) they must be pushed on the stack before the call.

To make these remarks more specific, consider the Pascal fragment:

PROGRAM p;

PROCEDURE subp(VAR i,j: integer);

The assembler code for a routine which calls procedure SUBP, should be structured as shown below. In particular, in the large program example, the .ATAB section name must be used, so that code base addresses will be relocated correctly.

```
*****
* Assembler calling Pascal (small program example)
*****

XREF SUBP
```

```

SECTION .CODE

; Set up SUBP's parameters on stack
PEA I
PEA J
BSR SUBP          ; Direct call ; to Pascal procedure

• (SUBP must be within 32K bytes of this BSR)

I      DS.L      1
J      DS.L      1

```

• Assembler calling Pascal (large program example)

' '

This linkage can always be used, but must be used if the distance between the calling and target routines exceeds 32K bytes.

```

XREF.L      SUBP      ; (The .L is essential)
XREF        .ATABS    ; .ATABS is a reserved public symbol

SECTION     .ATAB     ; .ATAB is a reserved section name
JHPSUBP JMP      SUBP      ; Direct 6-byte jump to SUBP

SECTION . CODE

PEA        I
PEA        J
JSR
JMP        SUBP
Set up SUBP's parameters on stack -.ATABS(A4) Indirect call to SUBP
* A4 contains the address of .ATAB at run-time)

```

9.5.4 COMMON data

Pascal variables which have been declared in COMMON can be referenced from assembler code as shown below. The use of section .ATAB is necessary in order that program initialisation can relocate the common block address at run-time.

In the general case, the assembler declarations must describe the layout of the Pascal common block. Section 9.4.2 gives details of storage layout for different data types.

As an example, the following might appear in a Pascal program:

```

TYPE
  time = RECORD
    hours: 0..24;
    mins, secs: 0..60;
  END;

COMMON

```

```

    timer: time;

BEGIN
    WITH timer DO
        writeln(hours:1, ':',mins:1,':',secs:1);

```

The method of accessing the common block TIMER is as follows

```

*****
* Accessing a COMMON block
*****
    XREF .ATABS

* .ATABS is a reserved public symbol defining the start of
* section .ATAB)

    COMMON TIMER

* Layout of TIMER
    DS.B HOURS
    DS.B MINS
    DS.B SECS

    SECTION .ATAB                                ; .ATAB is a reserved
                                                ; section name

ATIMER    DC.L TIMER                            ; Address of /TIMER/ at
                                                ; run-time

    SECTION .CODE
    MOVEA.L ATIMER-.ATABS(A4),A0                ;Get base addr. of /TIMER/

* (A4 contains the address of .ATAB at run-time)
    MOVE.B HOURS(A0),D0                        ; Get contents of HOURS

```

4.10.1.2. Preservation of registers

The generated Pascal code depends upon the contents of registers A3 to A6 being unchanged on return from a procedure. Assembler-coded procedures or functions must conform with these requirements. On return, the link and all parameters must have been removed from the stack.

4.10.1.3. Parameters

When a procedure or function has parameters, these are pushed onto 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

:

--+-----+---(increasing addresses -->)

·Sp

On return, parameters as well as link must have been removed.

4.10.1.3.1. 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 low-addressed byte of the pair). In the case of 4-or 8-byte values, the high-order word is pushed first followed by the lower-order word(s).

Set and dynamic-string values are passed adjusted to the length of the formal parameter. Such values consist of a length word followed by the actual set or string bytes. An odd-length set is increased in length by one byte by appending a zero byte.

Structure value parameters (arrays and records) are passed by address. (A copy of the array/record is then made in its local stack frame by the called procedure.)

4.10.1.3.2. VAR parameters

In all cases, the 4-byte address of the "first" (i.e. lowest-addressed) byte is pushed onto the stack.

4.10.1.4. Function results

A function call passes an additional "hidden" parameter on the stack before the actual parameters are passed. The extra parameter is the address of a location in the caller's data space that is to receive the function result.

4.10.1.5. Reserved section names

The section names .CODE and .ATAB and public symbol .ATABS are reserved and should only be used as shown by the examples above. The section names .IN1T, .NAME, .ENTRY and .LWT are also reserved and must not be used by any assembler routine under any circumstances.

5. Pro Pascal Operation

5.1. Installation Details

Hardware requirements

The hardware required to run the Pascal compiler is a Sinclair QL computer running QDOS with at least 60K bytes of user RAM, and with the PRL ROM cartridge fitted in the QL's ROM slot.

N.B. The PRL ROM cartridge must only be fitted or removed with the QL's power supply disconnected.

The minimum requirement *for* user programs is a Sinclair QL computer running QDOS. Memory and peripheral requirements depend on the program.

Delivery and installation

The Pascal software is delivered mainly on microdrive cartridges, containing the following files:

- **PAS** Pascal compiler control program
- **PROPAS1** Pascal compiler pass 1

- **PROPAS2** Pascal compiler pass 2
- **PROPAS_ERR** Pascal compile-time error messages
- **LINK** Linker
- **PASLIB_REL** Pascal run-time library
-
- **PLINIT_REL** Pascal library initialisation module
- **PLEND_REL** Pascal library end module
- **PAS_LINK** Standard Pascal linker command file
- **BOOT** SuperBASIC program to install PRL
- **PRL** Prospero Resident Library -see section 4.1
- **PROLIB** Librarian program
- **XREF** Cross-reference generator
- **SETDDEV**
- **NOQNS** Pascal configuration programs
- **SETSTACK**
- **TRAPREG_PAS** "QTRAP" include file -see Part II, 9.3
- **GRAPHICS_PAS** "WOPEN" etc. include file -see Part II, 9.3

Also supplied on microdrive are a few example source program (**_PAS**) files. If there are any special comments relating to the software, for instance descriptions of extra files, they are placed in a file called **READ_ME**. If this file is present, consult it before using the software.

The remaining Pascal software is supplied in a ROM cartridge. This contains the Prospero Resident Library (PRL), and it must be installed in the QL's ROM slot before using the Pascal compiler and also before running a Pascal program. Disconnect the QL's power supply before installing or removing the PRL cartridge.

Before use, it is essential to create working copies of the compiler and other files, and to use these rather than the supplied files, which should be kept safely as master copies.

The files required for compilation of Pascal source programs are **PAS**, **PROPAS1**, **PROPAS2** and, optionally, **PROPAS_ERR** (if the latter is not present, error messages will identify the error type by number without the text). The disposition of these files will depend on the user machine configuration:

For users with a basic QL, it is recommended that two "compiling" cartridges be created, the first containing **PAS**, **PROPAS1** and **PROPAS_ERR**, and the second containing **PROPAS2** and **PROPAS_ERR**. Then the software is immediately usable when loaded from microdrive 1.

For users with a disc system, a "compiling" disc may be set up containing all the above four files. In addition, a configuration file **PAS_CONFIG** must be set up and program **PAS** configured to use this file (see section 5).

It is recommended that source files be kept on their own separate cartridges or discs. Note that source file names must end in **_PAS**. There must also be sufficient space on a source file medium for the **_REL** files produced by the compiler. By default, there must also be room on the source file medium for the compiler's work file, which typically occupies about as much space as the source file; this

default arrangement can, however, be changed by reconfiguring (see section 5).

For microdrive users, it is recommended that a "linking" cartridge be created with the Linker and the files PASLIB_REL, PLINIT_REL, PLEND_REL, PAS_LINK and PROLIB on it. For disc users, these files can probably all fit on the "compiling" disc.

In the descriptions which follow, it is assumed that the "compiling" media are in MDV1 and the user's source media in MDV2.

5.1.1. Simple compile. link and execute

To prove that the software is installed and functioning correctly, make working media (see above) containing the compiler and linker files and copy the sample program source PRIME_PAS to a source file medium.

5.1.1.1. Compile

With the first compiler cartridge in MDV1 and the source cartridge in MDV2, type

```
exec pas
```

and the following output is generated in the compiler's window (the user must type the underlined information):

```
Pro Pascal Compiler -Version mmq 1.1
Copyright (C) 1965 Prospero Software
Source filename -MDV2_PRIME<ENTER>
Default options: G -console output to LOG file? (Y/N/.) .(ENTER>
Unit PRIME
Load compiler pass 2 in MDV1 and press ENTER
Name: PRIME Lines: 35 Code: 446 Data: 8
```

Note that the prompt for the second compiler pass only occurs for microdrive configurations. At this point, the source file PRIME_PAS on MDV2 has been compiled into a "relocatable binary" file PRIME_REL also on MDV2.

In the report output at the end of pass 2, the "name" is the module name of the relocatable module produced by the compilation. This name will be displayed when the linked program is executed. The "code" and "data" figures are in bytes, and represent the total generated code (+ constants), and data (excluding COMMON data items), respectively.

5.1.1.2. Link

To "link edit" the file PRIME_REL and produce an executable program, place the linker cartridge in mdv1 and type

```
EXEC MDV1_LINK
```

and enter the Linker command

The filename MDV1_PAS is converted by the Linker into the name MDV1_PAS_LINK. It uses this "template" file to link PRIME_REL with selected library subroutines from the file PASLIB_REL. (Further details are in section 3.)

The result of linking is the file PRIME_BIN on mdv2.

The command DIR MDV2_ shows three PRIME files: the source _PAS, the executable program _BIN, and also the relocatable version _REL which is generated by the compiler and read by the Linker.

5.1.1.3. Execute

Simply enter

EXEC MDV1_PRIME_BIN

Program PRIME reads from the standard file "input" and writes to the standard file "output", both of which are by default assigned to the console window. It repeatedly asks for a number and prints the smallest factor (or else prime'). For example (with input underlined) :

```
Input an integer up to a thousand million (0 to finish): 999999989
Smallest factor of 999999989 is : 4327
Input an integer up to a thousand million (0 to finish): 999999937
Smallest factor of 999999937 is : Prime
```

and so on.

The extra facilities of the compiler are explained in the next two sections. In particular, it is shown how to compile and link programs consisting of more than one source file.

5.2. Operation of the Compiler

Each invocation of the QL Pascal compiler processes one source file. A source file must contain one Pascal PROGRAM or SEGMENT, and the compiler converts this into a binary output file in Sinclair relocatable format, consisting of a single relocatable module.

Compilation is a 2-pass process under the overall control of program PAS. Pass 1 (the program PROPAS1) reads the source file and generates a temporary work file, with a name of the form Tm1\$_<job id>_IL, which contains a semi-compiled "intermediate-language" representation of the source program. When processing of Pass 1 is complete, PAS gives control to Pass 2 (the program PROPAS2). This program reads the work file, and generates the relocatable _REL file. When compilation is complete, Pass 2 deletes the work file and gives control back to PAS, which then terminates.

The previous section has described the Simple form of operation of the compiler. In this section, the various options and messages are explained.

5.2.1. Forms of invocation

The Pascal compiler may always be invoked and run interactively, and, if the QL Toolkit is installed, in batch mode as well, although the latter method may not be possible in an unexpanded QL owing to memory limitations. The following text assumes that the compiler is installed on microdrive, but it will equally apply to disc-based systems as well.

5.2.1.1. Interactive mode

With the first compiler cartridge in MDV1, enter:

```
EXEC MDV1_PAS
```

or use EXEC_W if desired. After opening its window and signing on, the Pascal compiler asks for the name of the _PAS source file to be compiled. It outputs:

```
Source filename
```

and the user should enter, e.g. "MDV2_CALC", it being assumed that there is a file called CALC_PAS on mdv2. If this file cannot be opened, the prompt is repeated.

The compiler then displays the currently configured default compilation options as a reminder (by default, no options will be configured, but the user may configure options as described in section

5):

```
Default options: <defaults>
```

The compiler then displays a series of prompts, one for each option, each of which may be replied to with "Y" (or "y") to select an option, or "N" (or "n") to not select an option, or "." to terminate prompting and use the defaults from then on:

```
G -console output to LOG file? (Y/N/.)
I -range checks on index bounds? (Y/N/.)
A -range checks on assignments? (Y/N/.)
P -checks on pointers? (Y/N/.)
N -track source names & line numbers at run time? (Y/N/.)
L -source listing? (Y/N/.)
D -real constants to double precision? (Y/N/.)
C -compact object code? (Y/N/.)
S -accept only strict Standard Pascal? (Y/N/.)
```

By default, the compiler simply reads a `_PAS` source file and produces a `_REL` file, which is always on the same device as the source. If the log file option is selected, a `_LOG` file is produced. If the listing option is selected, a `_PRN` file is produced. The contents of these optional files, which are also produced on the same device as the source, are described in later sections.

5.2.1.2. Batch mode

With the QL Toolkit installed, a single command suffices to perform one compilation run. Assuming the use of floppy discs, enter:

```
EX FDK1_PAS; '<source>/<options>'
```

(or use EW if desired), where

<source> might be, e.g. "FDK2_CALC", there being a source file `CALC_PAS` on FDK2, and

<options> gives the desired compilation option letters, e.g. "GIA", separated from <source> by "/".

Examples:

EX FDK1_PAS; 'FDK1_TABLE/GLN' causes the source file `TABLE_PAS` on FDK2 to be compiled with options G, L and N (see below).

EW FDK1_PAS; 'FDK2_CALC' causes the source file `CALC_PAS` on FDK2 to be compiled with the currently configured default options.

5.2.2. Compile-time options

The various compile-time options are described in the following sub-sections. The default setting for each option is "off", or N, when the software is shipped, but this can be altered by introducing a configuration file (see section 5.1).

5.2.2.1. G -console output to LOG file

When this option is specified, the messages output by the compiler to the console during compilation are written also to a file. The name of the file is the same as that of the source, with `"_LOG"` added. 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.).

5.2.2.2. I -range checks on index bounds

Range checks determine whether or not index expressions are within the correct limits. The checks

are carried out just before an index value is to be used, and have the effect of generating more code.

Index bound checks can be valuable in the early stages of program testing. If code size or speed is at a premium, they may be switched off once the program has been tested.

5.2.2.3. A -range checks on assignments

Assignment checks determine whether or not the values assigned to ordinal-type variables are within the range allowed for such quantities. The checks are carried out just before a value is to be assigned, and have the effect of generating more code.

5.2.2.4. P -checks on pointers

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.

5.2.2.5. N -track source names & line numbers at run time

This option instructs the compiler to insert extra code into the object program to maintain during execution a record of the source file name and line number corresponding to the code currently being obeyed. This information will be displayed in the event of any run-time error, and if the error is within a subprogram then the calling stack which is printed out (see section 4.3) will contain these file names and line numbers (for all calls which occurred in program units compiled with this option).

5.2.2.6. L -source listing

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 hexadecimal address of the start of that line within the object code. The listing is output to a file with the name of the source but ending in "_PRN" on the same device as the source. After the compilation it may be printed or displayed as desired.

5.2.2.7. D -double precision floating-point constants

With this option, the compiler is instructed to treat every unsigned-real constant (see Part 11, 1.1.5.2) as the corresponding unsigned-longreal constant (see Part 11, 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

1.2DO

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. (Note, however, that the construct integer / integer is always evaluated to single precision only.)

5.2.2.8. C -compact object code

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 (and is anyway rather small). Use of the option would only be recommended

for particularly large programs.

5.2.2.9. S -accept only strict Standard Pascal

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.

5.2.3. Compiler messages

When the compilation process proper begins, messages are output to the console to report progress and any irregularities.

5.2.3.1. Normal messages

In the main, these are self-explanatory.

If any use is made of the source file insertion facility (see Part 11, 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 win1_pascal_test_make 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).

At the end of Pass 2, the sizes of the code and data areas generated, and the total number of source lines, are printed. These are all decimal values. The data sizes do not include any COMMON variables.

Error messages

If the specified source file does not exist, the request for a file name is repeated.

If there is insufficient memory for compiler workspace or stack, one or other of the compiler passes will fail to run, and the compiler will terminate the compilation. Some possible corrective actions are given below.

Errors in the source program may be detected during either of the passes, though the majority generally appear in pass 1. The format in each case is: source line number and error code, with an explanatory sentence if the file PROPAS_ERR is present, followed by the text of the source line in error (pass 1 only). In Appendix B is a list of the error codes, with somewhat fuller descriptions where appropriate.

A single error, as the programmer sees it, may sometimes give rise to a number of reported errors. 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 a device (e.g. insufficient room for the `_REL` file). Such events give rise to error messages in the normal run-time error format (see section 4.3).

If the compiler indicates that its stack space has become full, this is most likely due to an unusually complicated expression in the source being compiled, e.g. the use of many levels of nested parentheses. The normal remedy for this would be to simplify the source expression.

If run-time error H is encountered during compilation, the normal remedy would be to repeat the compilation with a larger RAM area for the compiler to run in. (Resetting the machine prior to compiling may help, in this respect.) If no more memory is available, however, the only solution is to reduce the size of the compilation input.

5.3. Operation of the Linker

The Linker combines the output from one or more executions of the Pascal compiler with modules from the supplied run-time library to construct an executable program file. Linking is a 2-pass process, converting a collection of `_REL` files into an executable `_BIN` file. By default, a report is produced in a file with the same name as the `_BIN` file but with the extension `_MAP`. This report can be suppressed by including `"-NOLIST"` in the linker command line.

As an example, if the `_REL` file to be link-edited is `CALC_REL`, on drive `mdv2`, then a suitable linker command line is:

```
MDV1_CALC MDV1_PAS -NOLIST
```

The linker appends `"-LINK"` to the second filename and treats it as a command file. The contents of the supplied file `PAS_LINK` is:

```
INPUT MDV1_PLINIT
INPUT *
LIBRARY MDV1_PASLIB
INPUT MDV1_PLEND
DATA 4K
COMMON DUMMY
```

Each `INPUT` command directs the linker to include the specified relocatable file, the extension `_REL` being automatically supplied. The special form `"INPUT"` causes the first filename in the linker command line to be included, again after appending `_REL`. (In the present case, this will cause the file `MDV2_CALC_REL` to be read .)

A `LIBRARY` command, on the other hand, instructs the linker to include from the named file only those modules required, in the sense of having been referenced from module(s) already included. (So in this case, the library file `MDV1_PASLIB_REL` will be selectively scanned.)

This form of linker command is adequate for the case of a Pascal program consisting of a single source file (i.e. a `PROGRAM` and no `SEGMENTS` or `EXTERNAL` directives). If an executable program is to be constructed from more than one relocatable file - the output from two separate compilations, perhaps, or a Pascal compilation and some Assembler-coded modules - there are two ways to proceed. Either the separate relocatable files can be combined into one, by using the `PROLIB` utility (see section 6), or a new `LINK` template file can be created, by editing extra `INPUT` command lines into a copy of `PAS_LINK`.

When editing this file, it is essential to note that:

1. The file `PLINIT_REL` must be the first `INPUT` file.
2. Additional `INPUT` lines should be positioned before the `LIBRARY` line(s).
3. Additional user libraries must come before the `PASLIB` `LIBRARY` line.

4. *PLEND_REL* must be INPUT after all other *_REL* files and LIBRARY files.
5. The COMMON DUMMY option must be used.
6. If SECTION commands are introduced, the first such must be: SECTION * ENTRY.
7. The Linker's OFFSET command must not be used.

You may also wish to alter the size of the data space allocated by the DATA command. This value must be large enough to cover the following requirements:

- Two FCA (File Control Area) control blocks for standard i/o, the size of each being about 110 bytes.
- Stack requirements, being 16 bytes for each procedure called, 4 bytes for each VAR parameter, the size of the data item itself for each value parameter or local data item, and an extra 4 bytes for function calls. The calculation of stack space needs to allow for the worst case, i.e. the deepest nesting of procedure calls, and should include an allowance for use by the run-time library, as well as some margin of safety.

As an example, suppose it is required to combine an Assembler-coded module ASS_REL with a Pascal-coded module PASC_REL, and that a run-time stack requirement of 6K is required. Then one method of linking such a program is to create a linker template file called PASC_LINK, say:

```
INPUT MDV1_PLINIT
INPUT *
INPUT MDV2_ASS
LIBRARY MDV1_PASLIB
INPUT MDV1_PLEND
DATA 6K
COMMON DUMMY
```

The link operation can be performed by:

```
EX LINK
```

and then entering, as the linker command line:

```
-WITH MDV2_PASC
```

The linker uses the first name in this command (MDV2_PASC) to fill out the * place holder in the second INPUT line of PASC_LINK, and also as the "root" name for constructing the names of the executable file (by appending _BIN) and the storage-allocation report file (by appending _MAP). An executable file called MDV2_PASC_BIN is therefore produced, and can be executed by (for example):

```
EX MDV2_PASC_BIN
```

6. Operation of Object Programs

The Prospero Resident Library (PRL) is a collection of machine-code routines required by all Pascal programs, and also by the compiler. Before running a Pascal program, or the compiler, PRL must be installed. PRL is supplied in the ROM cartridge issued with the compiler, and also as a separate program on a microdrive cartridge. The ROM cartridge must be used for the compiler, but compiled Pascal programs can be run using either the ROM cartridge or the "software" PRL loaded from a file. The ROM PRL is installed by plugging it in to the ROM socket at the rear of the QL be-

fore applying power. The "software" PRL is installed by means of the SuperBASIC command:

```
LRUN MDV1_BOOT
```

(This command is, of course, invoked automatically at startup by the QL computer if there is a cartridge present in MDV1 holding the file MDV1_BOOT.

Once PRL is installed, it does not need to be re-installed except when the QL is reset or powered down with no ROM present. The SuperBASIC command

```
PRL
```

can be used to check the presence and correctness of an installed PRL.

The BOOT + PRL files can be copied along with linked Pascal object programs for use on machines other than the one used for compilation.

The fact that about 16K of common run-time code is held, separately, in the PRL substantially reduces both the size of object programs, and the time required for linking and loading programs, because the routines in PRL would otherwise be linked into every compiled program.

6.1. Execution of Pascal object programs

The normal way of executing a Pascal program that has been linked is by means of the SuperBASIC EXEC command (see below).

When executed, Pascal programs locate PRL and open a CON_ window for the display of run-time error messages and so on. The standard Pascal files input and output are also connected to this window, unless otherwise specified (see below).

6.1.1. Invocation by EXEC or EXEC_W

In order to execute a compiled and linked Pascal program, the SuperBASIC command EXEC or EXEC_W is used (or else the Toolkit command EX or EW). For example:

After opening a window and signing on, the Pascal program prompts the user as follows :

```
Standard input file? <>
Standard output file? <>
Option string? <>
```

with <> showing where the cursor is positioned awaiting user response. In simple cases, the ENTER key can be pressed for each question.

The first response associates a data file with the standard file "input". If a file name is entered, it is opened for exclusive input. If the open fails, the prompt is repeated. If only ENTER is pressed, any use of the file `input` in read etc. statements will be directed to the standard window that is being used for this initial dialogue.

The second response associates a data file with the standard file `output`. If a file name is entered, the file is opened for new overwrite. If only ENTER is pressed, any use of `output` in `writeln` etc. statements will be directed to the standard window.

The last response specifies a line of up to 80 characters to be made available as an option string to the running program. If the line is too long, the prompt is repeated. If only ENTER is pressed, the option string is of zero length. (The option string is obtained within the user program by calling the `getcomm` procedure -see part 11, section 9.2).

The above dialogue can be suppressed in a program by means of the NOQNS program issued with the compiler - see section 5.3 below.

6.1.1.1. Invocation by EX or EW Toolkit commands

In this case, there is a choice between running interactively as in 4.2.1 above, and running autonomously. For example (assuming suitable default file devices):

```
EX programname
```

or

```
EX programname;option-string
```

or

```
EX programname,infile_spec,outfile_spec;option-string
```

where zero or two data files may be specified (but not just one), and "option-string" is a SuperBASIC format string of characters to be passed to the initiated program.

In the first format, the program runs just as if EXEC had been used, that is, interactively, as described in the previous section. But in the second and third examples, the program starts with no further user interaction.

In the second example, standard input and output will be assigned by the program to its own window, whereas in the latter, the specified data files will be made available to the user program for standard input and output. It is an error to give only one datafile an extra dummy input or output file must be specified. (If more than two datafiles are specified, only the first and last files are made available to the user program as standard input and output: the others will not be accessible.)

In the second and third examples, the option-string is optional and, if omitted, a zero length string is passed to the initiated program.

Under EX control, there is no limit on the size of option-string as there is when EXEC program control is used.

6.1.1.2. Handling of pre-connected files

The previous two sections have shown how the user may specify one or two files to be pre-connected to a Pascal program, with the connection defaulting to the standard window. Then user program references to files input and output will be routed to the specified or defaulted files.

6.1.2. Invocation using execprog

Programs can also be executed from within Pascal programs using the execprog procedure (see Part 11, section 9.3). In this case, errors are reported to the initiating program by means of return codes. These return codes are listed alongside the corresponding messages for normal program initiation in section ~.~ below.

If a program uses the execprog facility to run a child program, its own pre-connected files are automatically made available to the child program (through any number of parent-child levels).

6.1.3. Run-time errors

The only aspect of program operation not determined from the program itself arises if an error is detected by the run-time software.

Errors can occur during the initialisation process before the program has fully started, and which are therefore not reported via the normal run-time error reporting method. A complete list of error messages produced during this initialisation process appears in section 4.4 below.

Once program execution proper has commenced, errors may be detected in a number of situa-

tions: file handling, 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 section 2.2). In all cases a report is made on the console, giving error type identified by a letter -and the hexadecimal (base 16) machine address relative to the start of the code:

```
Error x at address aaaaaa
```

A list of the run-time error codes is given in Appendix C. The address aaaaaa is directly comparable with the addresses provided in the MAP file generated by the Linker. For input/output errors, and some other cases, additional information appears with the standard message.

The standard error message is followed by trace information showing how the point in error was reached. This takes the form of a list of addresses at which procedure and function calls occurred. All addresses are relative to the start of the code. The first address given corresponds to the point where the main program called the next lower level of procedure, and so on up to the actual procedure or function in error.

For each source file which was compiled with the /N ("track source line numbers") option, the addresses in the error report are made more intelligible (without recourse to the linker's MAP file) by the addition of the source file name, the procedure name and the line number.

Finally, many classes of error allow continuation, and this choice is offered as a console option with (Y/N) response: press Y or y to continue, N or n to terminate the program.

6.2. Miscellaneous error messages

Most of these messages appear in the initiated program's standard window, with departures from this rule noted under particular messages. Where a program was initiated using execprog, the error is passed back as a return code to the initiating program.

```
Too long: <prompt>
```

The reply to the previous <prompt> was too long, e.g. a filename exceeded 36 characters. The correct reply should now be given.

```
Embedded blanks not allowed
```

This can appear in the standard window during the running of a user program. The reply to the previous prompt contained an embedded space character. In particular this can occur when a program opens a workfile and no default device has been configured. The user is prompted for a device name, which may not contain spaces. A corrected reply should be given to enable the program to continue.

```
Execution error: <error text>
```

where <error text> is one of the following:

- „bad command": (execprog return code -2) An invalid SuperBASIC command string.
- "pgm not opened": (execprog return code -3) Only possible when using execprog. The specified user program file could not be opened (can be due to the filename exceeding 36 characters).
- "not executable": (execprog return code -4) The specified program file was opened successfully but the file header showed that it was not an executable program file.
- "load failure": (execprog return code -5) The specified program file could not be loaded into memory, due to a failure of the QDOS "load-file" operation.
- "wrong version": (execprog return code -6) There is an inconsistency between the version of PRL loaded and the version of the run-time software used when linking the user program. Most

likely to be caused by upgrading to a new Pascal release without re-linking the user program with the new run-time software.

- "out of memory": (execprog return code -7) Insufficient memory is available for loading and/or running the user program.
- "init. failure": (execprog return code -8) The program has successfully been loaded, but then an error occurred in processing the relocatable items in section .ATAB.

If the program consists purely of Pascal code, this error implies a problem with the Pascal software, and it should be reported. If user-provided assembler-language routines were included in the link of the user program, they should be checked to ensure that they only use section .ATAB, if at all, as described in Part 11 section 9.5, namely for achieving the relocation of common block addresses and of JMP instructions having 4-byte absolute operands. In particular, this error can be caused quite easily by not preceding the assembler instructions by a suitable SECTION directive, so that they become part of .ATAB instead.

It can quickly be verified whether or not assembler routines are the cause of this error, by linking the user program without them, then loading it. The program will then fail during execution, rather than during initialisation.

- "no/corrupt PRL" (execprog return code -9) PRL (the Prospero Resident Library) is not loaded, or alternatively has been corrupted and is no longer usable.
- "window failure" (execprog return code -10) An attempt to open read or write to the standard window has failed.
- "only one chan" (execprog return code -11) If either of standard input and standard output is specified using EX or EW commands, then they must both be specified.
- "linking order" (execprog return code -12) Unlikely to occur, but indicates an erroneous attempt to link with _REL files generated by other compilers or assemblers.
- "no job created" (execprog return code -13) QDOS failed to create a job for the program to execute under.

6.3. The Configuration Programs

6.3.1. Configuring the compiler

The Pascal compiler consists of two main passes (PROPAS1 and PROPAS2) which execute under control of a small "parent" program PAS. There is also an error message file PROPAS_ERR. During compilation, an intermediate file is produced. Together, these files are too large to fit onto a single microdrive cartridge. The default arrangements are that PROPAS1, PROPAS2 and PROPAS_ERR are on MDV1, and the work file is on MDV2. The compiler prompts for the PROPAS1 cartridge in drive 1 to be replaced with the PROPAS2 cartridge during the compilation. For users with other devices, the arrangements can be altered, as follows.

Use SETDDEV (see 5.2.1) so that the PAS compiler program has a default device specified for it. The compiler will then look on this device for a configuration file with the name PAS_CONFIG. This is a five line text file, which may be prepared with a text editor, as follows:

line 1: default compile time options (e.g. /GL)

line 2: device holding PROPAS1 (e.g. FDK1_)

line 3: device for PROPAS2

line 4: device for PROPAS_ERR

line 5: device for intermediate file

Example:

```
/GI  
FDK1_  
FDK1_  
FDK1_  
FDK1_
```

A second aspect of the compiler's operation which can be adjusted by the user is the size of its stack work space (which is only made use of to a significant extent when processing very complicated nested expressions). The compiler subprograms PROPAS1 and PROPAS2 are issued with a stack size suitable for an unexpanded machine. For some source programs, this may be insufficient, and a compile time error will indicate that it has been exceeded. In this event the compiler's capacity can be increased by running the SETSTACK program (see 5.2.3 below) on PROPAS1. The user with more than the basic 128K RAM may well choose anyway to modify his working copy of PROPAS1 to have a stack size of 8K, or even larger.

6.3.2. Configuring object programs

The configuration programs enable object programs to be tailored to suit the user's own requirements. Three utilities are provided on the issue cartridge. They are all machine code programs which require PRL to be installed.

The aspects which may be configured are:

- the default device on which anonymous files are to reside;
- whether the initial dialogue (described in section 4.2 above) takes place or not;
- the stack size allocated for the running program.

Each utility reads an existing program file (which may or may not itself be configured), applies the changes specified and writes a new version of the program file. The question and answer session to specify the options required is largely self-explanatory.

6.3.2.1. Default device -SETDDEV

Each Pascal object program contains a field in which may be specified a "default device". This device is used by compiled programs for the placement of anonymous files (those for which no name was specified by a call of the "assign" procedure.)

The initial value for this option is "no device". In this state, when a compiled program creates a work file, the user is prompted for the name of a device to use.

The default device may be altered using the program SETDDEV. It is invoked by:

```
EXEC xxxx_SETDDEV
```

(where xxxx is the name of the device holding the SETDDEV program), and the user is then prompted for the name of the program file to be modified, and the name of the device to be installed as the default for that program. A 4-character device name should be specified, or 4 blanks for "no device".

6.3.2.2. Initial dialogue -NOQNS

A dialogue normally takes place with the user when a Pascal object program is executed (as described in section 4 above). In many cases, this dialogue is not required and execution of the program can proceed immediately with default values for the initial responses. The initial dialogue (and associated messages) can be suppressed with the NOQNS program.

NOQNS is initiated by the SuperBASIC command:

```
EXEC xxxx_NOQNS
```

(where xxxx is the name of the device holding the NOQNS program). The user is prompted for the name of the program file to be modified, and the appropriate change is made.

```
Stack size -SETSTACK
```

The instructions to the linker contained in PAS_LINK or any individual link job derived from it include a specification of the "DATA" size. In Prospero object programs the space so defined is used mainly for the stack, as detailed in section 3 above.

The stack size for a program is modified using the SETSTACK program. SETSTACK is run by the SuperBASIC command;

```
EXEC xxxx_SETSTACK
```

(where xxxx is the name of the device holding the SETSTACK program). SETSTACK then gives instructions for modifying the stack size.

6.4. Operation of the Librarian

The purpose of the PROLIB librarian utility program is to administer files which are in Sinclair relocatable object format such as those produced by Prospero's Pro Pascal or Pro Fortran-77 compilers, or by GST's macro Assembler. Individual modules may be extracted, and/or files may be merged together into libraries. A number of report options are also available.

A file created by PROLIB will be in Sinclair relocatable format, and so suitable for processing by linkers capable of handling this format, such as GST's LINK.

6.4.1. Forms of invocation

There are three ways of operating the librarian: the "one-line", the "conversational" and the "indirect" mode. All the options are available in each mode.

6.4.1.1. The one-line command

With the QL Toolkit installed, a one-line execution of PROLIB is possible, as in:

```
EX PROLIB; options
```

(or use EW if desired).

The option string (supplied in quotes after the ';' character) must be constructed as follows. First must come the name of the "library" file. This may optionally be followed by the character "I" together with one or more letters from the set M, X, U, N, D.

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; they may be 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:

```
EX FDK1_PROLIB; 'MDV1_NEWLIB/M = MDV1_OD1, MDV1_OD2'
```

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. Any of the input filenames may be immediately followed by a "module selector" (see 6.3).

No filename extension may be given (whether for the library or the component input file names): the extension *_REL* is supplied automatically by the librarian.

6.4.1.2. Conversational mode

By entering the command (assuming a floppy-disk system):

```
EX FLP1_PROLIB
```

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 just <ENTER> on its own.

Again, no filename extension must be given: PROLIB automatically appends *_REL* to all filenames.

6.4.1.3. Indirect mode

The QL Toolkit must be installed for this mode to be used.

The indirect mode of operating the librarian combines the features of the first two modes: a one-line command is given containing the name of a "command file" (preceded by the character @), this command file containing the answers to the questions which would be asked in the "conversational" mode.

For example, typing

```
EX FDK1_PROLIB; '@ FDK2_MLIB ·
```

where the text file MLIB contains the lines

```
MDV2_MLIB
N
N
N
Y
MDV1_M1LIB, MDV1_M2LIB, MDV1_M3LIB
```

causes PROLIB to combine the modules from the 3 files M2LIB_REL and M3LIB_REL into the composite library file. Note that if the command file name has no extension, none is supplied by PROLIB.

6.4.2. 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.

6.4.2.1. 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, the Sections it contains, and all Public symbols defined and External symbols referenced within it. The "Sections" are pieces of the code or data which go to make up an executable program; their sizes (in decimal) are printed.

6.4.2.2. Cross-reference listing (X)

The report consists of two parts. The first part gives, for each Public/External name in the library file (in alphabetical order), the name of the module in which it is defined (i.e. is a Public 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 Sections (in alphabetical order) together with the names of the modules which reference them.

6.4.2.3. Unsatisfied references listing (U)

This report is concerned with the requirement imposed by many linkers that, for a library which is to be "selectively" searched (cf. the /S option described in section 6.3), 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. 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.

6.4.2.4. Suppress .names (N)

(This option is only meaningful if at least one of H, U or X has been selected .)

In order to avoid conflict with user-defined names, most Public and Section names in the Pascal library begin with '!'. Since these 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.

6.4.2.5. Listings to disc (D)

(This option is only meaningful if at least one of H, U or X has been selected.)

The default destination for reports is the console. If this option is chosen, the reports are written instead to a disc or microdrive file. The file is given the same name as the library file, but with the extension _PRN.

6.4.2.6. 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:

```
FDK1_NAME/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:

```
FDK1_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:

6.4.3. Librarian messages

6.4.3.1. Normal messages

If in the "create" mode, when it starts to process each input file the librarian writes the full filename to the console.

6.4.3.2. Error messages

6.4.3.2.1. Non-fatal errors

If an input file is empty, this is reported and the next file is processed.

If an input file cannot be found (perhaps because its name has been misspelt), the librarian reports this 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.

6.4.3.2.2. Fatal errors

If any other error situation occurs, execution is aborted immediately, after outputting a message to the console.

The first group of such messages are caused by driving the librarian incorrectly. There are 4 such.

```
Command line improperly terminated
```

In the one-line command mode, the library filename and switches have been read, followed by a character other than ";".

```
Command file not found
```

In the indirect mode, the filename after the @ character is illegal or the file does not exist.

```
No library filename supplied
```


In the indirect mode, the first line in the command file should contain a valid QDOS filename.

```
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.

The other group of errors should never occur. The most probable cause is that an input file is not in Sinclair relocatable format at all. The error messages are:

```
Cannot find subsection
End of input file encountered
Illegal directive encountered
Illegal id encountered
Inconsistent section id
Input file relocatable format incorrect
Name in input file exceeds 32 characters
SECTION/Common inconsistency
```

6.5. Cross-Reference Generator

A cross-reference generator XREF is provided as part of the Pro Pascal package. It reads a source file and produces a report listing all the identifiers used, in alphabetical order, and the source lines on which these identifiers occur.

A cross-reference generator is a very useful facility when developing programs of any size, and XREF is tailored to the Pro Pascal syntax (extra reserved words, hexadecimal and longreal constants, underscore within identifiers, source file insertion).

6.5.1. Forms of invocation

There are two ways of operating the cross-referencer: the "one_line" and the "conversational" modes.

One-line mode

With the QL Toolkit installed, a one-line execution of XREF is possible, as in:

```
EX FLP1_XREF; 'MDV1_PRIME'
```

(or use EW if desired).

The option string (supplied in quotes after the ';' character) must consist of the source file name, excluding the final '_PAS', which is supplied automatically by XREF, as by the compiler.

The source file is read, and the cross-reference listing is output direct to the screen, with a page width of approx. 100 characters.

Conversational mode

By entering the SuperBASIC command (assuming a floppy-disk system):

```
EXEC FDK1_XREF
```

the conversational mode of operation is entered.

The program first requests the source filename. As with the compiler (and the one-line mode described in 7.1.1) the _PAS suffix is supplied automatically.

Next, the line width can be set. Replying to this question with just <ENTER> 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.)

Finally, the destination for the listing is specified. This may be any device or file name. Examples are: *SER* (for the printer), or *MDV1_PRIME_PRN*. No suffix is added by XREF. If simply <ENTER> is pressed, the default destination is used.

6.5.2. Messages

If use is made of the source file insertion facility (see Part 11, 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 2.3.1 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.

Appendix A: Language Summary

A.I: 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: anyone 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 'string-character', which stands for any 8-bit-code character. (The characters in the 7-bit-code ASCII set are printed 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.II: Lexical Aspects

```
compilation-unit = {token {separator} }
```

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

```
separator = space | end-or-line | comment
```

```
special-symbol = "+" | "-" | "W" [ "I" I "=" [ "(" I ">" I "[" J '1'  
"." f "," I ":" f "j" I ".... n I "C" I ")n J ">" "(=" I "(=" I  
":=" I" "J
```

```
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} "'"  
space = " "  
comment = "{ " | "("*) character-sequence "}" | "*)" "  
word-symbol = "AND" | "ARRAY" | "BEGIN" | "CASE" | "COMMON" |  
"CONST" | "DIV" | "DO" | "DOWNT0" | "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 = {string-character}
```

```
decimal-integer = digit-sequence  
hexadecimal-integer = digit {hexdigit} "H"  
scale-factor = [sign] decimal-integer  
apostrophe-image = "'"
```

```
hexdigit = digit "A" "B" "C" "D" "E" "F"
```

A.III: Syntactic Aspects

```
compilation-unit = program | segment
```

```
program = program-heading block "."  
segment =segment-heading segment-declarations "BEGIN" "END".
```

```
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
```

```
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
```

```
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"
```

```
constant-identifier = identifier  
constant =[sign] [unsigned-number | constant-identifier] |
```



```

                                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

```

```

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}

```

```

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

```

```

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}

```

```

factor = variable-access | unsigned-constant | function-designator
        set-constructor "(" expression ")" | "NOT" factor
multiplying-operator = "*" | "/" | "DIV" | "MOD" | "AND"
case-constant = constant

```

```

unsigned-constant = unsigned-number | character-string |
                    constant-identifier | "NIL"
function-designator = function-identifier [actual-parameter-list]
set-constructor = "[" [member-designator{"," member-designator}] "]"

```

```

member-designator = expression [".." expression]

```

Appendix B: Compile-Time Errors

For each error number, the text (provided PROPAS_ERR is present) which is printed at compile-time plus, where necessary extra explanation is given,

Number	Meaning	Explanation
1	Simple type expected	
2	Identifier expected	

Number	Meaning	Explanation
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	
22	.. expected	
24	Illegal source character	
25	One identifier may not follow another	
49	No cases in case statement	
50	Error in constant	

Number	Meaning	Explanation
51	:: expected	(in an assignment statement or FOR)
52	THEN expected	
53	UNTIL expected	
54	DO expected	
55	TO or DOWNT0 expected	
58	Error in factor	
59	Error in variable	
101	Identifier declared twice	The identifier is printed
102	Low bound exceeds high bound	
103	Identifier is not of appropriate class	The identifier is printed
104	Identifier has not been declared	The identifier is 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 or set is outside the (ordinal) range 0.. 262127
115	Base type of set must be ordinal type	
116	Bad parameter type for standard procedure	

Number	Meaning	Explanation
117	Unsatisfied forward reference	The name of the undefined type (which will have occurred after A 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/sub-range/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	

Number	Meaning	Explanation
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
141	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)

Number	Meaning	Explanation
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 variable of a FOR-loop may not be modified in the body of an inner-level procedure or function
160	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

Number	Meaning	Explanation
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
263	Too many COMMON names	More than 128
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.. 262127
305	Warning: FOR loop will never be executed	Final-value < initial-value (if TO), or > initial-value (if DOWNT0). 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
309	Case-constant outside range of tag-type	
310	String size must be integer constant	(in the declaration string[n])

Number	Meaning	Explanation
311	String length exceeded	Attempt to assign packed array of char hold it, or array longer than 32767 string value parameter
312	Set length exceeded	
320	Incompatible parameter-lists	to string too short to characters passed as
322	Undeclared FORWARD procedure or function	The name is 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 identifier is 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	
331	End of source file encountered	Probably due to incorrect block structure (a missing END, etc.), or to an unclosed comment (a missing "!")
333		
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
370	Code for procedure exceeds 32K bytes	
380	Integer overflow (in a compile-time integer expression)	
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 a null string

Number	Meaning	Explanation
403	Compiler stack size insufficient	May be due to a particularly complex source expression
404	Compiler workfile contents invalid	Should not normally occur, and may indicate a compiler malfunction
405	Compiler workspace insufficient	Insufficient memory available to the compiler. Memory may have been 'lost' to the system by programs run before the compiler. Re-booting the machine may recover such memory. Otherwise, a smaller source program or more memory are the only solutions

B.I: 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: $\text{abs}(\text{value}) > 32768.0$ Longreal argument: $\text{abs}(\text{value}) > 4.29509$
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.	<ul style="list-style-type: none"> • Unable to open input file (filename displayed). • Disc or directory space insufficient for output. • Attempted reset of an output device or rewrite of an input device. • Attempt to read as FILE OF t1 a file created as FILE OF t2 (different from t1).
E Stack error on exit.	The stack pointer is not correct on exit from a procedure or function. Possibly because an incorrect Assembler-coded subroutine has been called.
F File programming error.	Incorrect sequence of operations on a file (e.g. attempted read before reset).

Code	Meaning
G OWNERR error.	The user has attempted to install a non-outer-level procedure as the error handler (see Part 11, section 9.3.6).
H Heap overflow.	Insufficient free space for "new" operation
J Divide error (integers).	In "i DIV j" j is zero; in "1 MOD j" j is zero or negative. Continuation possible, but results not prediotable
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" or "rename" operation is not in the correct format for a file or device name.
O Overflow during integer arithmetic.	From 32-bit add or subtract (when option A or I invoked) or from 32-bit multiply (always checked). 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 (with compile-time checking option P).
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, either the item does not start with an integer or the value is too large or incorrectly constructed (e.g. 1.e5).
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 32767 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 file size beyond the maximum
V Set construction error.	A set expression contains element(s) outside the range 0 to 262127.
W Write error on textfile.	Fieldwidth parameter is outside the range 1 to 32161 .
X Overflow during real or long-real arithmetic.	Exponent out of range. Continuation possible but results not predictable.

Code	Meaning
Y Call of missing procedure or function.	
Z Divide by zero (reals or long-reals).	Continuation possible, but results not predictable.

The Pascal Standard (ISO 1185) 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, DoIO, Do11, D.III, Do15, 0016, D023, D032, D.33, D. 3!!, D035, D036, D040, Do!!1, D.42, Do!!4, D045, D046, D05I, 0.54, D.56, DoS7, D.56

If the A compile-time option is specified, the following additional errors are detected at run time: D07, D08, D.17, 0018, D.37, Do!!7, D.49, 0.50, 0.52, D.53, D.S5

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 I compile-time option is specified, error D.3 is, additionally, detected.

The following errors are not, in general, reported: D.2, D.3, 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.)

Appendix C: Mixed Language Programming

Program construction

Mixed language programs may be constructed by amalgamating Pro Fortran-77 and Pro Pascal components. Pascal segments can be included in a Fortran program, or Fortran subprograms can be included in a Pascal program.

Input/output may be performed in both languages independently. Only one special rule applies here: in the case of a Fortran main program with Pascal procedures, the standard files "input" and "output" will not have been implicitly assigned to the console, and hence they must be assigned explicitly and a reset or rewrite given before they are used.

When a mixed language program terminates, either normally, or through a run-time error, all open Pascal and Fortran files are closed.

After compilation by the appropriate compiler, the components are link-edited together (refer to Part III). In the linker command, the ~EL files are listed in sequence (with the main program module typically coming first), followed by the names of the two libraries to be selectively scanned, with the library for the main program language coming first.

Correspondence of data types

The table below shows the correspondence between Fortran and Pascal data types.

Fortran	Pascal
INTEGER	integer
INTEGER' 2	-32768..32767
INTEGER' 1	-128..127
REAL	real
DOUBLE PRECISION	longreal
COMPLEX	RECORD realpart,imagpart: real; END
LOGICAL' 1	boolean
CHARACTER • n	PACKED ARRAY [1..n] OF char

Single variables of equivalent type may be associated (for example by the COMMON facility) and referenced from either language.

Arrays may also be referenced from either language, but if an array has more than one dimension it is important to notice that Fortran stores in "column major" order (as given by the "subscript value" function) and Pascal in "row major" order. The sequence of subscripts/indices must therefore be reversed when changing from one language to the other.

The COMMON facility can be used within mixed language programs with the understanding that when a Pro Pascal variable is declared in COMMON the variable name becomes a common block name. The significance of this may be seen in the example below.

Fortran	Pascal
COMMON CB A,B,C A = 5.5	COMMON cb: RECORD a,b,c: real; END cb.a := 5.5;

Here the two declarations each describe a common block containing three real variables. The suggested form of declaration of cb as a record provides the equivalent separation of the common block name from the names of the variables within it. However, if the block contains just one component another method is to give the block and the contents the same name in Fortran:

Fortran	Pascal
COMMON X X(120)	COMMON x: ARRAY [1..120] OF real;

Parameters

The name of a Pascal procedure at the outer level can be quoted in a Fortran CALL statement, or a Fortran subroutine can be given an EXTERNAL declaration within Pascal and then referenced. Pascal and Fortran functions are similarly equivalent.

The Pascal parameters and the Fortran arguments must match in number, order, and type (see above). All Pascal parameters must be VAR parameters.

Note that in the case of a CHARACTER argument, Pro Fortran-77 does not pass the address of the start of the data (as would be done in Pro Pascal for a PACKED ARRAY [] OF char), but the address of a 6-byte Character Variable Descriptor (CVD). This is structured like a Pascal record consisting of a 4-byte address field followed by a 2-byte length field.

A Fortran subroutine or function which has a subroutine or function as a dummy argument can be called from Pascal, but because Pascal passes two addresses in such cases (the entry address and the static link), the Fortran must include an extra dummy argument to match the latter. For example

Pascal

```
FUNCTION area (PROCEDURE calc): real; EXTERNAL;
```

Fortran

```
REAL FUNCTION AREA (DUMMY , CALC)
```

The Pascal procedure passed as an actual corresponding to calc must be declared at the outer level.

Interchange of files

A Pro Fortran-77 file of variable-length formatted records has the same layout as a Pro Pascal file of type text. (Both, in fact, are normal QDOS text files.)

A Pro Fortran-77 file of fixed-length records has the same layout as a non-text file in Pro Pascal. To exchange binary files, it is necessary to know the file element size implied by the Pascal file declaration, since this must be quoted explicitly in the RECL= parameter of the Fortran OPEN statement. Refer to the user manual for sizes of the various data types. Correspondence between data layouts within the file elements is similar to that for COMMON variables (see above), bearing in mind that a Fortran unformatted READ or WRITE simply copies the data items in the io-list between memory and file. Thus for example a file written by a Pascal program as

```
FILE OF RECORD
  item: integer;
  vmax,vmin: real;
END;
```

would be read in Fortran by

```
OPEN (5, RECL=12, ACCESS='DIRECT', *** )
READ (5, REC= *.* ) ITEM,VHAX,VHIN
```

There is no equivalent in Pro Pascal to a Pro Fortran-77 file of variable-length unformatted records.