

University of Edinburgh



Department of Computer Science

---

**The IMP-77 Language**

by

Peter S. Robertson

Internal Report

CSR-19-77

James Clerk Maxwell Building,  
The King's Buildings,  
Mayfield Road,  
Edinburgh,  
EH9 3JZ.

December, 1977  
Revised May, 1979  
Revised May, 1983

**THE IMP77 LANGUAGE**

**A Reference Manual**

**Peter S. Robertson  
Lattice Logic Ltd. 1983**

**Reprinted by permission of  
Lattice Logic Ltd.  
Edinburgh  
May 1983**

## Contents

|  |    |
|--|----|
| Introduction . . . . .                           | 1  |
| Character set. . . . .                           | 2  |
| Atoms. . . . .                                   | 3  |
| Compile-time features. . . . .                   | 7  |
| Statements . . . . .                             | 8  |
| Expressions. . . . .                             | 10 |
| Declarations . . . . .                           | 13 |
| Data precision specification . . . . .           | 18 |
| Access to structured variables . . . . .         | 19 |
| Own variables. . . . .                           | 20 |
| Constant arrays. . . . .                         | 20 |
| Initialisation . . . . .                         | 21 |
| Assignment . . . . .                             | 22 |
| String resolution. . . . .                       | 24 |
| Conditions . . . . .                             | 25 |
| Conditional groups . . . . .                     | 27 |
| Repetition . . . . .                             | 29 |
| Block structure. . . . .                         | 32 |
| Begin blocks . . . . .                           | 32 |
| Procedures . . . . .                             | 34 |
| Parameters . . . . .                             | 37 |
| Procedure specification. . . . .                 | 39 |
| External linkage . . . . .                       | 40 |
| Program file structure . . . . .                 | 42 |
| Permanent procedures . . . . .                   | 43 |
| Events . . . . .                                 | 45 |
| Control transfer instructions. . . . .           | 47 |
| Implementation-dependent features. . . . .       | 50 |
| Appendix 1 - A note on the grammar . . . . .     | 52 |
| Appendix 2 - Compiler messages . . . . .         | 53 |
| Appendix 3 - Program listings. . . . .           | 59 |
| Appendix 4 - Standard events . . . . .           | 61 |
| Appendix 5 - Permanent procedures. . . . .       | 62 |
| Appendix 6 - Variant and archaic forms . . . . . | 71 |
| Appendix 7 - ASCII character set . . . . .       | 72 |

## Introduction

IMP77 is an "ALGOL-like" high-level language. Relative to ALGOL 60, the language adds program structuring, data structuring, event signalling, and string handling facilities, but removes (or retains in a modified form) intrinsically inefficient features such as the ALGOL 60 name (substitution) parameter.

The language, based on Atlas Autocode, was originally designed as the implementation language for the Edinburgh Multi-Access System - hence its name - but has since been used successfully for implementing systems, teaching programming and as a general-purpose programming language on many different machines.

Two of the major design aims were:

1. The language should compile to efficient machine code.
2. The syntax of the language should be verbose rather than obscure.

Most IMP77 systems provide comprehensive compile-time and run-time diagnostics, together with an option to suppress generation of run-time checks when compiling tested programs.

Input/output facilities are provided through the external procedure mechanism and are therefore open-ended and can be defined as required, though a standard set of procedures is supported.

It is assumed that the reader is familiar with the more general concepts of high-level programming languages.

The examples of grammar given in the text are simplified in order to show the general features of the syntax.

## Character set

An IMP77 program is a sequence of statements constructed using the ASCII seven bit character set extended with an underlined alphabet.

### Newline

The NEWLINE (or LINE BREAK) character has ASCII code value 10 (NL).

### Quotes

Several language constructions call for one or more characters (text) to be enclosed in quotes; between quotes all characters are significant and stand for themselves.

N.B. Space, newline, and percent characters may appear between quotes and stand for space, newline, and percent.

Two quote characters are used:

|   |                   |
|---|-------------------|
| ' | - character quote |
| " | - string quote    |

If it is required to include the delimiting quote within the text it must be represented by two consecutive quotes: e.g.

|                 |                                 |
|-----------------|---------------------------------|
| ''''            | - the symbol quote              |
| "A ""big"" dug" | - a string of eleven characters |

However, note: ''' and "it's mine"

### Spaces

Except when used to terminate keywords or when between quotes (q.v.) spaces are ignored by the compiler and may be used to improve the legibility of the program.

### Lower Case Letters

Except when enclosed in quotes (q.v.) lower case letters are equivalent to the corresponding upper case letters.

### Control characters

Except for NL (see above) all non-quoted characters whose ASCII codes are outwith the range 32 to 126 inclusive are treated as spaces, but will be sent to the listing unaltered. In particular, the character FF (form feed) may be used to control the pagination of program listing files.

## Atoms

An atom is the basic unit of a program statement and is either a keyword, a special symbol, an identifier, or a constant.

### Keywords

A keyword is a sequence of underlined letters. In source programs underlining is achieved by using the shift character, percent (%), which is defined as underlining the subsequent letters, underlining being terminated by any non-alphabetic character. Hence the following statements are equivalent:

%string(7) %array %name P

%string (7) %arrayname P

and both represent: string(7)arrayname P

In this manual keywords will be written in lower case and underlined.

The following is a list of all of the IMP77 keywords:

|                  |                |                 |                 |                 |               |
|------------------|----------------|-----------------|-----------------|-----------------|---------------|
| <u>alias</u>     | <u>and</u>     | <u>array</u>    |                 |                 |               |
| <u>begin</u>     | <u>byte</u>    |                 |                 |                 |               |
| <u>c</u>         | <u>const</u>   | <u>constant</u> | <u>continue</u> | <u>control</u>  |               |
| <u>cycle</u>     |                |                 |                 |                 |               |
| <u>diagnose</u>  | <u>dynamic</u> |                 |                 |                 |               |
| <u>else</u>      | <u>end</u>     | <u>event</u>    | <u>exit</u>     | <u>external</u> |               |
| <u>false</u>     | <u>file</u>    | <u>finish</u>   | <u>fn</u>       | <u>for</u>      | <u>format</u> |
| <u>function</u>  |                |                 |                 |                 |               |
| <u>if</u>        | <u>include</u> | <u>integer</u>  |                 |                 |               |
| <u>label</u>     | <u>list</u>    | <u>long</u>     |                 |                 |               |
| <u>map</u>       | <u>monitor</u> |                 |                 |                 |               |
| <u>name</u>      | <u>not</u>     |                 |                 |                 |               |
| <u>on</u>        | <u>of</u>      | <u>option</u>   | <u>or</u>       | <u>own</u>      |               |
| <u>predicate</u> | <u>program</u> |                 |                 |                 |               |
| <u>reel</u>      | <u>record</u>  | <u>repeat</u>   | <u>result</u>   | <u>return</u>   |               |
| <u>routine</u>   |                |                 |                 |                 |               |
| <u>section</u>   | <u>short</u>   | <u>signal</u>   | <u>spec</u>     | <u>start</u>    | <u>stop</u>   |
| <u>string</u>    | <u>switch</u>  | <u>system</u>   |                 |                 |               |
| <u>then</u>      | <u>true</u>    |                 |                 |                 |               |
| <u>unless</u>    | <u>until</u>   |                 |                 |                 |               |
| <u>while</u>     |                |                 |                 |                 |               |

### Special symbols

The special symbols are:

|    |    |    |    |    |      |         |
|----|----|----|----|----|------|---------|
| @  | !  | !! | #  | ## | &    | (. . .) |
| *  | *= | +  | -  | -> | .    | /       |
| // | :  | ;  | <  | <- | <<   | <=      |
| =  | == | >  | >= | >> | [..] | [..]    |
| -  | -- | -  |    |    |      |         |

## Identifiers

An identifier is a sequence of any number of letters and digits starting with a letter, e.g. MAX, X, CASE 1, Case 2, case 2b. All letters and digits are significant.

With the exception of labels, all identifiers must be declared before they may be used (see Declarations).

## Constants

### Integer Constants (Fixed Point)

#### a) NUMERICAL constants

A numerical constant is a sequence of decimal digits.  
For example: 7, 43, 2195, 0, 8, 100 000 000

#### c) CHARACTER constants

The ASCII code value of any character may be obtained as an integer value by enclosing the character in single quotes. When the required character is a single quote it must be represented by two consecutive single quotes.  
Examples: 'A', 'a', '+', '0', ''', ''', ' ', ' ', ' ',

Note the last three examples, which represent the code values for single quote, space, and newline respectively.

The predefined named constant NL may be used in place of the rather cumbersome form of a newline character enclosed in quotes.

In general, a character is an integer in the range  $0 \leq \text{character} \leq 255$ .

#### d) MULTI-CHARACTER constants

The ASCII code values for several characters may be packed together to form a single integer constant, by enclosing the characters in single quotes and giving the prefix M.

e.g. M'over', M'Max', M'1+2', M'\*@@#'

The value of the constant is calculated by evaluating the expression:  $\dots [ ( C1 \ll B + C2 ) \ll B + C3 ] \ll B + \dots$  where C1, C2 .. are the characters in the order specified, and B is an implementation-defined constant (commonly 8).

Note that M'?` = `?`

### Constant Integer Expressions

An integer expression with operands which are constants may be used wherever an integer constant is required (see Expressions).

### Real Constants (Floating Point)

A real constant is a sequence of decimal digits optionally including one decimal point. The constant may also be followed by a scaling factor of the form  $\text{@}[\text{signed integer constant}]$  meaning "times ten to the power  $.[\text{signed integer constant}]$ ". For example, ignoring any machine-dependent accuracy problems, the following real constants all have the same value:

120.0, 120, 1.2@2, 12@1, 1200@-1

Note that a decimal integer constant is a special case of a real constant.

### Radix Specification

Integer and real constants may be specified to bases other than ten by adding the prefix "[base]\_" to the constant, where [base] is the base represented to base ten. The letters A, B, ..., Z may be used to represent the 'digits' 10, 11, ..., 35.

E.g. 2\_1010            ten in binary  
      8\_12             ten in octal  
     16\_A             ten in hexadecimal  
      3\_0.1            one third

In the case of real constants any scaling factor will remain in base ten unless a different base is explicitly requested.

E.g.       10 @ 2        one hundred  
      2\_1010 @ 2       one hundred  
      2\_1010 @ 2\_10    one hundred



### String Constants

A string constant is a sequence of not more than 255 characters enclosed in double quote characters - a double quote being represented inside a string constant by two consecutive double quotes. There are no restrictions on which characters may appear within strings.

E.g. "starting time", "x = y\*4+z", "a "red" hood"

Note 1 "a" is a string constant of one character.  
'a' is a character (integer) constant.

ii The null string, a string of no characters, is permitted and is represented by two consecutive double quotes ("").

### EBCDIC Constants

String and character constants may be specified as using the EBCDIC character set rather than ASCII by applying the Prefix E. In the case of multi-character constants the E prefix replaces the M prefix.

E.g. E"Ebcidic string", E'O', E'VOL1'

The particular variant of EBCDIC used is implementation-dependant.

### Named constants

Named constants may be declared using the prefix constant in front of a simple declaration with initialisation (see Declarations). A named constant may be used wherever a literal constant of the same type is required. Note that implementations may restrict the use of named real and string constants as replacements for literal constants.

```
{const} ::= constant [type] [cinit] ( " , " [cinit] ) *  
[cinit] ::= [id] "=" [constant]
```

```
constant integer MAX = 17, MIN = 2
```

```
constant real PI = 3.14159
```

```
constant string (7) VERSION = "Ver:1.6"
```

The keyword constant may be abbreviated to const.

## Compile-time features

### Listing Control

During the compilation of a program a line-numbered listing can be produced. The statements list and endoflist may be used in a nested fashion to control this listing. Following an endoflist, listing is inhibited until either the end of the program is searched or a matching list is encountered. The default is for listing to be enabled.

Along with each line number in the listing file the compiler may add a marker character to provide extra visual information about the nature of the statements being listed. The markers are:

- + this line is a continuation of the previous line.
- & this line is part of a file being included (see include).
- " the compiler is currently searching for a string quote to match one given on a previous line.
- ' the compiler is searching for a character quote to match one given on a previous line.

### Include

A file of statements (terminated by the end of the file or the statement end of file) may be compiled into a program by giving a statement of the form:

include [file specification]

where [file specification] is a string constant representing a system-dependent file name. Refer to the relevant implementation notes for details of implementation-dependent limitations on the use of include.

E.g. include "ECSC17.LISTVARS"

## Statements

A STATEMENT is a sequence of atoms arranged according to the syntactic rules of IMP77.

### Termination

Every statement must be terminated by a newline or, except in the case of comment statements, a semicolon.

### Null Statements

Redundant terminators (newlines or semicolons) effectively generate null statements which are ignored by the compiler and may be used to improve the legibility of the program.

### Continuation

A statement may extend over several physical lines provided that each line break occurs after a comma, or, and, or is preceded by the keyword c (which is otherwise ignored).

E.g. 
$$\begin{array}{l} \text{if } X = Y \text{ then } P = 1 \text{ } \underline{c} \\ \quad \underline{else} \text{ } P = 0 \end{array}$$

is exactly equivalent to: 
$$\text{if } X = Y \text{ then } P = 1 \text{ } \underline{else} \text{ } P = 0$$

Note i The line break following c causes unuerting to be terminated.

ii %c between quotes stands for the two characters percent and c.

iii Comments (q.v.) may not be continued.

### Instructions

An instruction is any imperative statement which may be made conditional, and is either an assignment, a routine call, a control transfer, or a compound instruction.

### Compound instructions

Two or more instructions may be joined using the keyword and to form a compound instruction: e.g.  $A=0$  and  $B=C-1$ . Within a compound instruction a control transfer may only occur as the final instruction. A compound instruction may appear wherever an instruction is required, and results in the component instructions being executed in the order given.

## Comments

A comment is a sequence of characters which is ignored by the compiler, and is intended to permit annotation of programs.

Comments are any sequence of characters, excluding right brace and newline, enclosed in a pair of braces, { and }. A comment may appear between any two atoms, but may not occur within an atom. For convenience the closing brace may be replaced by a newline.

The following is a valid fragment of a program containing comments:

```
LIMIT = 100      (only 100 cases)
MINIMUM = 0      (all positive
PROCESS(X (cases), Y (total cost))
```

and will be seen by the compiler as:

```
LIMIT = 100
MINIMUM = 0
PROCESS(X , Y )
```

For historical reasons, any physical line which starts with an exclamation mark, and is not a continuation line, is considered as a comment and is ignored by the compiler.

## Expressions

### Arithmetic Expressions

An arithmetic expression is a sequence of operators and integer or real operands obeying the elementary rules of algebra. An operand is either a constant, a variable, a function call, a map call, or an arithmetic expression enclosed in parentheses or vertical bars (see Declarations and Procedures).

#### a) Integer Expressions

All the operands and operators in an integer expression must yield integer values.

The operators available are:

- + addition
- subtraction or unary minus
- \* multiplication
- // integer division (the remainder of the division, which is of the same sign as the dividend, is ignored).
- integer exponentiation. The second operand (the exponent) must be a non-negative value.

#### b) Real Expressions

All the operands and operators in a real expression must yield real or integer values. Integer values will automatically be converted into their real equivalents before being used.

The operators available are:

- + addition
- subtraction or unary minus
- \* multiplication
- / division
- real exponentiation

#### c) Ambiguous expressions

Certain operators, such as + and -, may take either integer or real operands. If the two operands are of the same type the result of the operation will be of that type. If the types differ, the integer operand will first be converted to real and the operator will yield a real result. Hence in the expression  $(7.4 + 22 * 6)$ , \* will perform an integer multiplication and + will perform a real addition (see Precedence of operators).

#### d) Modulus

The modulus or absolute value of an expression (integer or real) may be obtained by enclosing that expression between vertical bars. E.g. |X-Y|  
The type of the expression is unchanged.

#### Bit-Vector Expressions

All operands must yield bit-vector (integer) values. The operations are performed on a bit-by-bit basis using the operators:

& and  
| inclusive or  
^ exclusive or  
<< left shift (logical)  
>> right shift (logical)  
~ complement (unary not)

It is permissible to mix integer and bit-vector expressions but the full implications of this may be machine dependent.

The shifting operators (<< and >>) may only be used to shift by a non-negative amount which is less than the number of bits in an integer variable.

All operands are converted to integer precision before use.

#### String Expressions

All operands of a string expression must yield values of type string. The only operator available is "." for concatenation (joining together) and no sub-expressions in parentheses are permitted. The result of the operation is a string value whose actual length is the sum of the actual lengths of the original operands.

E.g. "Mr ".surname

## Precedence of operators

- Highest: 1. `~` (unary not)  
2. `^`, `^^`, `<<`, `>>`  
3. `*`, `/`, `///`, `&`  
Lowest: 4. `+`, `-` (unary and binary), `!`, `!!`

The precedence rules may be overridden by means of parentheses.

Note:  $-1^{**2} = 0-(1^{**2}) = -1$   
 $(-1)^{**2} = 1$   
 $2^{**2^{**3}} = (2^{**2})^{**3} = 4^{**3} = 64$

## Order of evaluation

Excluding the operator precedence rules described above, no assumptions may be made about the order of evaluation of expressions; the compiler is free to use the commutative, associative, and transitive properties of operators to reorder expressions.

Note i Unary minus is treated as 0-...

- ii An expression may not contain two adjacent operators; they must be separated by parentheses E.g. `23*(-14)`
- iii Integer values will be converted to real where necessary, but real values will never be converted to integer unless this is explicitly specified using the predefined functions `INT` or `INTPI`.
- iv Integer (or real) values may be explicitly converted to real values using the predefined function `FLOAT`.
- v byteinteger and shortinteger values will automatically be converted into their integer representations before being used.

## Declarations

All identifiers except labels must be declared at the start of a block before they may be used. The scope of an identifier is the rest of the block in which it is declared, including any blocks subsequently defined therein (see Block Structure and note 3 on Labels and Jumps). In the following discussion the phrase [type] has the definition:

```
[type] ::= integer,  
         real,  
         string "(" [max] ")",  
         record "(" [fm] ")"
```

and [max] is an integer constant in the range  $1 \leq \text{max} \leq 255$  defining the maximum number of characters which may be held in the string.  
[fm] defines the structure of the record (see Records).

When used to define pointer variables or maps(q.v.) ([max]) and ([format]) may be specified as (\*) meaning that the defined object may reference any string variable or any record variable.

### 1. Scalar Variables

#### a) Simple Variables

```
[simple]      ::= [type]  
[simple dec] ::= [simple] [idents]
```

```
integer J,K,COUNT  
real PRESSURE  
string (30) COUNTRY, TOWN  
record (CARFM) MINI, ROVER
```

Each variable is allocated an appropriate (machine dependant) amount of storage to hold a value of the appropriate type.

#### b) Simple Pointer Variables

```
[simple pointer] ::= [type] name  
[simple pointer dec] ::= [simple pointer] [idents]
```

```
integer name P  
real name DATUM  
string (15) name WHO,WHERE  
record (CARFM) name CAR
```

Each variable is allocated enough storage to hold a pointer to (i.e. the address of) a simple variable of the specified type. The use of a simple pointer variable is generally equivalent to the use of the simple variable to which it currently points.



c) General Pointer Variables

```
[general pointer] ::= name  
[general dec]    ::= [general pointer] [idents]
```

name NA, NB

Each variable is allocated enough space to hold a general pointer to a variable of any type. Such pointers may be decomposed into an address, a size and a type by means of the built-in functions ADDR, SIZE OF, and TYPE OF (see Permanent Procedures). General pointer variables may not be used in a context where a value is required.

d) Array Pointer Variables

```
[array pointer]    ::= [atype] [aname]  
[array pointer dec] ::= [array pointer] [idents]
```

```
[atype]           ::= [type]    array,  
                    [type] name array,  
                    name array  
[aname]           ::= array "(" [dim] ")" name,  
                    array name  
[dim]             ::= [integer constant]
```

integer array name AN  
real array name VALUES  
string (20) array name NAMES, ADDRESSES  
record (CARFM) array name MAKE  
integer name array name POINTERS  
name array name GEN POINTERS  
real array (4) name SPACE TIME

Each variable is allocated enough storage to hold a pointer to (i.e. the address of) an array of the specified type.

The three forms of [atype] permit access to arrays of simple variables, simple pointer variables, and general pointer variables.

The first form of [aname] specifies the dimension, [dim], of the sort of array to be accessed; the second form is an abbreviation for the case where [dim] = 1.

## 2. Arrays

```
[array] ::= [etype] [edefn] <" " [edefn]>*  
[edefn] ::= [idlist] "(" [bounds] ")"  
[bounds] ::= [bound pair] <" " [bound pair] >*  
[bound pair] ::= [lower bound] ":" [upper bound]  
[lower bound] ::= [integer expression]  
[upper bound] ::= [integer expression]
```

```
integer array A(1:10),B,C(-4:LIMIT)  
real array Q(1:J+K, 1:J-K)  
string (12) array CLASS(-7:16)  
record (CARFM) array TABLE(LOWER:UPPER)  
integer name array FREQ('A':'Z')  
name array WHAT(0:1)
```

The bound pairs are evaluated and the required amount of storage is allocated to each identifier.

- note 1 In each bound pair the values of the bounds must satisfy the condition:  
Upper bound - Lower bound + 1  $\geq$  0  
This means that arrays may contain zero or more elements.
- ii The number of bound pairs (the dimension of the array) usually may not exceed six, but this is implementation dependent.

### 3. Records

A record is a named collection of data objects. The components (elements) of a record may be any of the forms discussed in (1) and (2) above, with the following limitations:

- i Arrays within records must be one dimensional and have constant bounds.
- ii A record may not contain simple records (or record arrays) of its own format. However it may contain record pointer variables of its own format.

The internal structure of a record is defined using a record format statement:

```
[format]      ::= record format [fm] "(" [format list] ")"
[fm]          ::= [id]
[format list] ::= [alternative] < or [alternative] >*
[alternative] ::= [dec list],
                "(" [format list] ")"
[dec list]   ::= [dec item] < "," [dec item] >*
[dec item]   ::= [simple] [idents],
                [pointer] [idents],
                [general pointer] [idents],
                [array pointer] [idents],
                [array]
```

```
record format F(integer X, record(F)name LINK)
record (F) HEAD
record (F) array CELL(1:15)
record format AS(byte array CHAR(0:12) or string(12) TEXT)
```

Alternatives, as in the definition of AS above, provide a means of imposing different interpretations on parts of a record. Each alternative within a format list will start at the same address within the record and will be padded out with anonymous variables to the size of the longest. The relation between pairs of elements in different alternatives is machine-dependent. Alternatives may be nested to any depth.

Note 1 Each element in a format must have an identifier which is unique within that format; there are no restrictions on the use of identifiers which have been used outwith the format. For example, the following program fragment is valid:

```
integer J, K  
record format FM(integer J, K, L)
```

ii When space is allocated to a record variable the elements are laid out in the order in which they were declared. However see the relevant implementation notes for machine-dependent alignment considerations.

Occasionally it is necessary to be able to refer to a recordformat before it is possible to define it, as in the example below. A statement of the form:

```
recordformatspec (fm)
```

may be used to declare the format identifier. Until the format is declared fully in a recordformat statement the identifier may only be used in the declaration of record pointer variables.

```
recordformatspec Y  
recordformat X(record(Y)name P, real VALUE)  
recordformat Y(record(X)name Q, integer VALUE)
```

### Data precision specification

On some machines it is possible to offer a range of sizes and precisions for variables of type integer or real, and so a mechanism is provided for extending the set of arithmetic data types. The size of integer variables may be changed by adding one of the prefixes byte, short, or long to the keyword integer, and the precision of real variables may be changed by adding the prefix long to the keyword real. The prefix is added immediately in front of the integer or real keyword, and gives rise to constructions such as:

byte integer  
short integer name  
own long real  
external byte integer

The exact meanings of these prefixes are machine-dependent but may be described approximately as:

byte - large enough to hold a character (unsigned)  
short - a signed subset of integer values  
long - a larger range than integer,  
or greater precision than real

Commonly byte gives 8 bits, short 16 bits, and long 64 bits.

Where values are required byte integers and short integers are considered equivalent to normal integers, hence INTEGER=BYTE is a valid instruction. However, where references are concerned the types must be identical, hence INTEGERNAME==BYTE is not a valid instruction. See Assignment.

Before use, byte values will be zero-extended to integer precision and short values will be sign-extended to integer precision.

If the host machine cannot support different data sizes the prefixes will not effect the allocation of variables.

Refer to the relevant implementation notes for details of specific implementations.

The keywords byteinteger and shortinteger may be abbreviated to byte and short respectively.

## Access to structured variables

### Arrays

Access to particular elements of an array is achieved by following the array identifier by a list of subscript expressions enclosed in brackets.

e.g. Q(1, K-3)      A(J)

The number of subscript expressions must equal the number of bound pairs given in the declaration of the array and the value of the expressions must be integers within the range specified by the corresponding bound pairs.

### Record element selection

Selection of a specific element from a record is achieved by following the record by:

"\_"[element id]

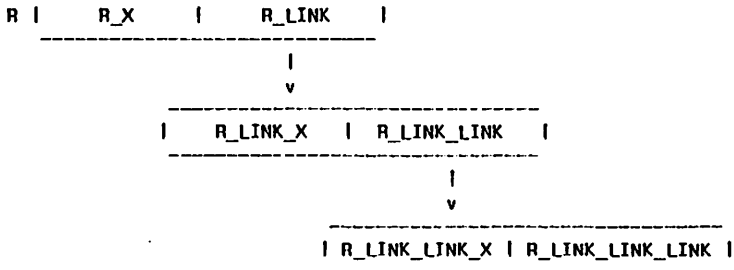
where [element id] refers to an identifier within the format associated with the given record. Clearly, if the record had been declared using \* as a format, no such selection is possible.

Given the declarations:

record format F(integer X, record(F) name LINK)  
record (F) R

some valid references to variables would be:

R                    - a record of format F  
R\_X                  - an integer  
R\_LINK               - a pointer to a record of format F  
R\_LINK\_X             - an integer  
R\_LINK\_LINK         - a pointer to a record of format F  
R\_LINK\_LINK\_X       - an integer



### Own variables

Each variable declared in a block (q.v.) is allocated storage when that block is entered, the storage being released when the block is left. This means that local variables (and the values in them) are lost between traverses of the block.

If, however, the prefix own is applied to a declaration the variables are allocated statically (at load time) and so retain their values when the block is not being executed (see Procedures). The scope of the identifier is unchanged.

Own arrays must be one-dimensional and have constant bounds.

### Constant arrays

The prefix constant may be used in place of own in the declaration of an initialised array (see initialisation) to indicate that the initial values cannot be altered. constant arrays must be one-dimensional and have constant bounds.

A strict definition should prohibit the use of elements of constant arrays wherever there is the possibility of their being assigned new values. Unfortunately this is not convenient in practice as it would prevent passing constant arrays as parameters to routines which never attempt to write to them. Accordingly in the context of == assignments (q.v.) the compiler treats constant arrays as though they were own arrays and leaves checking to hardware protection mechanisms.

## Initialisation

Simple variables and pointer variables may be given initial values when they are created; if no initial value is specified the content of a variable is initially undefined. Note that pointer variables must be assigned using "==" and simple variables using "=" or "<-" (see Assignment).

```
integer A,B=4, C=-1-B (value in A is undefined)
real R=1.234E-5
string (7) WHO="anon"
integer name P == A
```

Own variables are initialised once (effectively before the program begins execution) but ordinary variables are initialised each time the containing block is entered. Arrays may only be initialised if they are own or constant (q.v.). If an own or constant array is to be initialised, every element in the array must be given a value. In order to simplify this, each initial value may be followed by a repetition count in parentheses, and a star, (\*), may be used to represent the number of remaining elements in the array. For convenience a repetition count of zero is permitted and means that the initialising constant is to be ignored. For example the following declarations are all equivalent:

```
own integer array A(2:5) = 7,7,7,7
own integer array A(2:5) = 7(4)
own integer array A(2:5) = 7(*)
```

The list of constants may extend over several physical lines without the need for a continuation mark if each line ends with a comma; a line break is also allowed after the equals sign.

```
constant string (3) array MONTH(1:12) =
    "JAN", "FEB", "MAR",
    "APR", "MAY", "JUN",
    "JUL", "AUG", "SEP",
    "OCT", "NOV", "DEC"
```

```
own integer array OPCODE(0:20) =           (opcode values)
    16_5800, 16_4800, 16_5000, 16_4000,
    (   L      LH      ST      STH      )
    16_5A00, 16_5B00, 16_5C00, 16_5000,
    (   A      S      M      D      )
    16_1A00, 16_1800, 16_1C00, 16_1000,
    (   AR      SR      MR      OR      )
    -1(*)                                     (all the rest)
```



## Assignment

Assignments are instructions which cause the contents of variables to be altered. Note that the compiler is free to choose the order of evaluation of the left and right hand sides of assignments, and so the use of procedures (q.v.) with side-effects is to be discouraged.

There are three forms of assignment:

1. [variable] "=" [expression]

```
X = Y
A(P) = A(P)+1
Y = BIT<<12 !! MODE_FLAGS
PERSON = INITIALS.SURNAME
```

The expression is evaluated and the resulting value is stored in the given variable. The expression may be of type integer, real, or string, and the variable must be of a compatible type; in the case of a real variable an integer expression will have its result converted to real before the assignment. Note that if N and M are (for example) integer name variables, the statement N=M copies the value in the variable pointed at by M into the variable pointed at by N.

2. [pointer variable] "==" [reference to a variable]

The pointer variable is dynamically made equivalent to the given variable; the types of both sides of the assignment must be identical - this includes the formats of records, and the maximum lengths of strings. The assignment may be thought of as the assignment of the address of the variable to the pointer.

Once equivalenced the pointer variable may be used as a synonym for the variable.

```
integer name N
integer X
integer array A(1:6)
X = 1
N == A(X)      {N is now equivalent to A(1)}
X = 2
N = 0          {same effect as A(1) = 0}
```

3. [variable] "<-" [expression]

This is similar to 1. above except that the value of the expression will be truncated if necessary (see Data Precision Specification).

E.g. string(4) S

```
S = "12345"      {fails String Overflow at run-time}
S <- "12345"     {will assign "1234" to S}
```

### Record assignment

Two special assignments exist for records:

1. [record variable] "=" [record variable]

The area of storage associated with the right-hand record is copied into that associated with the left-hand record in a simple-minded fashion, ignoring the structure of the records. The formats of the two records must be identical.

2. [record variable] "=0"

The storage area associated with the record is set to zero, ignoring the structure of the record.

### String resolution

The contents of a string variable may be searched for a sub-string and decomposed accordingly.  
The format of a resolution is:

```
(resolution) ::= [source] "->" [dest]
[dest]       ::= [dest1]? "[" [pattern] "]" [dest2]?
[source]     ::= [string variable]
[pattern]    ::= [string expression]
[dest1]      ::= [string variable] "."
[dest2]      ::= "." [string variable]
```

```
S -> T.( ". ").U
TITLE(J) -> ("Sir").REST
WHO -> WHO.(LETTERS."B.Sc.")
S -> ("HELLO".T)
```

[pattern] is evaluated and [source] is searched from left to right to find the string of characters, [pattern].  
If [pattern] can be found the resolution is deemed to have succeeded otherwise it is deemed to have failed.

If the resolution succeeds, [source] can be considered to be of the form: [left].[pattern].[right], where [left] and [right] are the fragments of [source] respectively to the left and right of the first occurrence of [pattern]. If [dest1] has been specified it is assigned the value [left]. If [dest2] has been specified it is assigned the value [right].

Hence after executing the following statements:

```
string(15) A, B, C, D
A = "123456789456123"
A -> B.("456").C
A -> ("61").D
```

B will contain "123", C will contain "789456123", and D will contain "23".

A resolution may occur in two contexts:

1. as an instruction, in which case failure of the resolution causes an event to be signalled (see Events)

```
WHO -> ("Mr ").WHO; WHO = "Dr ".WHO
```

2. as a simple condition (see Conditions), in which case the simple condition is satisfied if and only if the resolution succeeds, resulting in the resolution being performed and the necessary assignments being made.

```
SAYING = A.****.B while SAYING -> A.(RUDE WORD).B
```

## Conditions

Conditional statements are specified using the phrase [condition], which is defined as:

[condition] ::= [simple cond] &and [simple cond]>\*,  
                  [simple cond] &or [simple cond]>\*

"&and" conditions are satisfied if all of the component simple conditions are satisfied; "&or" conditions are satisfied if any one of the component simple conditions is satisfied.

[simple cond] has seven forms:-

1. [expression] [comp] [expression]

|            |       |                               |
|------------|-------|-------------------------------|
| [comp] ::= | "=",  | - is equal to                 |
|            | "#",  | - is not equal to             |
|            | "<",  | - is less than                |
|            | "<=", | - is less than or equal to    |
|            | ">",  | - is greater than             |
|            | ">=", | - is greater than or equal to |

The given expressions are evaluated and compared. The simple condition is satisfied if the relation specified by the comparator holds. Both expressions must yield values of the same type.

Complete records or arrays may not be compared.

2. [expression] [comp] [expression] [comp] [expression]

This form of simple condition may be thought of as a contraction of the form:

( [x1] [comp1] [x2] &and [x2] [comp2] [x3] )

except that the middle expression [x2] is only evaluated once. Note that the third expression, [x3], is only evaluated if the condition specified by the first two expressions is satisfied.

Such a simple condition is frequently used to check for a range of values, E.g. 17 <= VALUE <= 100

Note that these double-sided conditions are only available for value comparisons.

3. [reference to a variable] "==" [reference to a variable],  
   [reference to a variable] "##" [reference to a variable]

The two variables, which must be of identical type, are compared for equivalence, that is their addresses are compared. Note that the address of a pointer variable is the address of the variable to which it is equivalent.

The simple condition is satisfied if the addresses are equal (== specified) or not equal (## specified).

4. [predicate call] - see Procedures

The given predicate is called and the simple condition is satisfied if and only if the predicate terminates by executing the instruction true.

5. [resolution] - see String Resolution

The resolution is attempted. If it fails the simple condition is not satisfied, otherwise the resolution is performed and the condition is satisfied.

6. "(" [condition] ")"

This form of simple condition is provided to enable the use of both and and or in a condition, as these connectives are considered to have equal precedence. The connectives and and or may not appear in the same condition unless separated by levels of parentheses.

E.g.  $A=0$  or  $(B=1$  and  $C=2)$  or  $D=3$

7. not [simple cond]

This simple condition is satisfied if and only if the simple condition following not is not satisfied. For example, the following simple conditions are exactly equivalent:

$A \neq 0$   
not  $A = 0$

### Testing of conditions

The testing of a condition proceeds from left to right, simple condition by simple condition, terminating as soon as the inevitable outcome of the condition is known.

For example, considering the condition:

$A = 0$  or  $B/A \neq C$

If the variable A has the value zero the whole condition will be satisfied without " $B/A \neq C$ " being tested.

Conditional groups  
(see Block Structure)

The most general form of a conditional group is a sequence of statements of the form:

```
if [condition1] then start  
    (statements to be executed if)  
    ([condition1] is satisfied)  
  
finish else if [condition2] then start  
    (statements to be executed if)  
    ([condition1] is not satisfied and)  
    ([condition2] is satisfied)  
  
finish else if [condition3] then start  
    .....  
    .....  
  
finish else start  
    (statements to be executed if all the)  
    (previous conditions are not satisfied)  
  
finish
```

Note that "if .... start" and "finish else .... start" etc. are complete statements in their own right and as such must be terminated by a newline or semicolon.

Any or all of the else statements may be omitted, and the start-finish groups may be nested to any depth.

## Alternative forms

1. then start may be elided into start.
2. If the start-finish brackets enclose only one instruction the complete start-finish sequence may be replaced by that instruction.

E.g. .. if [condition] then [instruction] .....  
or ..... else [instruction]

3. The keyword if may always be replaced by unless with the effect of negating the whole of the condition. For example, the following two statements are equivalent:

```
    if X = 0 then Y = 1 else Z = -1  
unless X = 0 then Z = -1 else Y = 1
```

4. In a statement of the form: "finish .... start" both of the keywords finish and start may be omitted.

e.g. if A = 0 start  
    FLAG = 1  
    else if A >= 12  
        FLAG = 2  
    else if A < -4  
        FLAG = 0  
    else  
        FLAG = -1  
    finish

5. A statement of the form:

```
    if (condition) then [instruction]
```

may be rewritten in the more natural form:

```
    [instruction] if [condition]
```

E.g. NEWLINE if CHARS >= 60

Note that else is not available in this variant.

Repetition (loops or cycles)  
(see Block Structure)

Indefinite Repetition

A group of statements may be repeated indefinitely by enclosing them between the statements cycle and repeat.

```
    cycle  
        GET DATA  
        PROCESS DATA  
    repeat
```

Subsequently the group of statements between cycle and repeat will be referred to as the 'cycle body'.

cycle-repeat groups may be nested to any depth.

Conditional Repetition

The number of times the cycle body is executed can be controlled by modifying the cycle and repeat statements.

a. while [condition] cycle

Before each execution of the cycle body the specified condition is tested. If the condition is satisfied the cycle body is executed, otherwise control is passed to the statement following the matching repeat.

The cycle body will be executed zero or more times.

b. for [control] "=" [init] ", " [inc] ", " [final] cycle

where

```
[control] ::= [integer variable] - control variable  
[init]    ::= [integer expression] - initial value  
[inc]     ::= [integer expression] - increment  
[final]   ::= [integer expression] - final value
```

On each entry to the cycle the address of the control variable and the values of the three expressions are evaluated and saved; execution of the cycle body cannot change them. The control variable is assigned the value "[init]-[inc]".

At the start of each iteration the value in the control variable is compared with the value [final]. If they are equal control is passed to the statement following the matching repeat, otherwise the value [inc] is added to the control variable and the cycle body is executed.

This definition may be informally described by the following program:

```
integer    Temp Inc = Inc,  
           Temp Final = Final  
???name Temp Control == Control (same type as Control)
```



```

Temp Control = Init-Temp Inc
while Temp Control # Temp Final cycle
    Temp Control = Temp Control+Temp Inc
    (cycle body)
repeat

```

The cycle body will be executed zero or more times.

On exit from the cycle the control variable will contain the value it held immediately prior to the point at which the cycle terminated, usually [final].

The execution of the cycle body must not alter the value of the control variable.

- c. The final form of conditional cycle is:

```

cycle
    (cycle body)
repeat until (condition)

```

After each execution of the cycle body the condition is tested. The loop is repeated if the condition is not satisfied.

until loops always execute the cycle body at least once.

Note that until does not mean while not (.....).

### Simple forms of loop

If the cycle body comprises only one instruction the loop may be rewritten in the form:-

```
[instruction] [loop clause]
```

i.e. [instruction] while [condition]  
 [instruction] for [control="init"], "[inc]", "[final]"  
 [instruction] until [condition]

For example

```

A(J) = 0 for J = 1, 1, 20
READSYMBOL(S) until S = NL
SKIPSYMBOL while NEXTSYMBOL = ' '
B = B+1 and N = N/2 while N # 0

```

## Cycle control instructions

Two instructions are provided to control the execution of a cycle from within the cycle body.

1. exit - causes the cycle to be terminated and control to be passed to the statement following the matching repeat.

The while and until forms of loop may be expressed using exit:

```
cycle      (while)
  exit unless condition
  .....
repeat

cycle      (until)
  .....
  exit if condition
repeat
```

2. continue - causes control to be passed to the repeat of the current loop, where any until conditions will be tested.

### Block structure

An IMP77 program is constructed using one or more blocks, which may be nested one within another; the depth to which this nesting may be performed is implementation dependent.

Note that start - finish (see Conditional Groups) and cycle - repeat (see Repetition) do not define blocks, they merely define the scope of conditions and loops.

When control passes into a block all non-own variables declared in that block (but not in blocks defined within it) are allocated storage, and remain in existence holding their values until control passes out of the block. At this point the variables are destroyed and the storage space is released for later use.

### Begin blocks

The simplest type of block is enclosed between the statements begin and end and is referred to as a begin block.

A begin block is entered by executing the begin and is left by passing through the end to the following statement. They are anonymous routines (q.v.) which have one implied call at the point of definition. The main uses of begin blocks are to declare arrays with bounds calculated at run-time, and to enable the re-use of space taken up by large arrays which are only needed for part of the program.

E.g. begin

```
    integer UPPER
    UPPER = ... (calculate upper bound)
    begin
        integer array CASES(1:UPPER)
        .....
        .....
    end
    begin
        integerarray TEMP(1:1000)
        .....
        .....
    end
    begin
        real array WORK(1:2000)
        .....
        .....
    end
end
```

## Local and Global variables

An identifier is described as being local to a block if it was declared in that block. Any identifiers which are in scope but which were not declared in the block in question are referred to as being global to the block. Clearly, identifiers may be local to only one block but may be global to many.

```
begin                (start of outer block)
  integer X          (X is local to this block)
  begin              (start of inner block)
    integer Y        (Y is local to this block)
    X = 0            (X is global to this block)
  end                (of inner block)
end                (of outer block)
```

Identifiers may always be redeclared in any block to which they are global - the local incarnation taking precedence over the global one.

```
begin
  integer X
  begin
    integer X
    X = 0 (uses the X of the previous line)
  end
end
```

Any attempt to redeclare a local variable will be faulted by the compiler.

## Procedures

A procedure is a block which has an associated identifier; a complete procedure block may be considered as the declaration of the procedure identifier.

Unlike begin blocks, procedures are not entered simply by reaching their first statement; this results in control being transferred to the statement following the matching and. Instead, procedures are activated when they are called by giving the procedure identifier in a context determined by the type of procedure. The effect of a call is to suspend the current flow of control and to pass control to the procedure. When the procedure terminates normally, the previous flow of control is resumed.

There are four forms of procedure, the exact form required being specified by the heading of the block.

The phrase [param def]? stands for the optional parameter definition and will be described later (see Parameters).

### 1. routine [id] [param def]?

A routine call may occur wherever an instruction is required.

When the call is executed, control is transferred to the routine which executes until either the and is reached or the instruction return is executed. This causes the routine to terminate and the previous flow of control to be resumed.

```
integer X, Y

routine CONVERT
  if X < Y start
    X = X+Y
  finish else start
    X = X-Y
  finish
and
..
..
CONVERT
..
CONVERT unless X = 0
```

2. [type] function [id][param def]?

A function is a procedure which calculates a value of the specified type (integer, real, string, or record) and may be used wherever an operand of the specified type is required.

When a function is called its statements are executed until the execution of an instruction of the form:

```
result "=" [expression]
```

This causes the function to terminate, returning the value of the expression.

```
integer X,Y,Z
```

```
integer function SUM
```

```
  result = X+Y
```

```
end
```

```
Z = SUM      {same effect as "Z=X+Y"}
```

The keyword function may be abbreviated to fn.

3. [type] map [id] [param def]?

A map is a procedure which calculates a reference to a variable of the specified type (integer, real, string, or record), and may be used wherever a variable of the specified type is required.

When a map is called its statements are executed until the execution of an instruction of the form:

```
result "==" [reference to a variable]
```

This causes the map to terminate, returning a reference to (i.e. the address of) the given variable.

E.g. integer X,Y

```
integer map MIN
```

```
  if X < Y then result == X else result == Y
```

```
end
```

```
MIN = 0
```

```
{the above statement is exactly equivalent to:}
```

```
{if X < Y then X = 0 else Y = 0}
```

#### 4. predicate [id] [param def]?

A predicate is a procedure which tests the validity of an hypothesis and may be used wherever a simple condition is required. When a predicate is called its statements are executed until either the instruction true is executed, in which case the predicate returns and the simple condition it constitutes is satisfied, or the instruction false is executed, in which case the predicate returns and the simple condition is not satisfied.

Note that a predicate does not return any value.

E.g. integer N

```
predicate SINGLE DIGIT
  true if 0 <= N <= 9
  false
end

N = N//10 unless SINGLE DIGIT
```

#### Notes

- i A routing may terminate by reaching and; all other types of procedure must not be able to reach and, otherwise the compiler will report a fault.
- ii Procedures may be nested within any form of block.
- iii Procedures may be recursive, that is, a procedure definition may contain a reference to itself.
- iv It is not possible to jump out of a block. Similarly a procedure can not be terminated by executing the appropriate statement (return etc.) contained in an inner block. If it is required to force a return from several blocks the signal mechanism should be used (q.v.).
- v Functions, maps, and predicates may alter variables global to themselves, but such side-effects should be avoided or used with caution as, in general, no assumptions may be made about the order in which parts of statements will be executed.

## Parameters

In the previous discussion about procedures the phrase [param def]? was used. This stands for an optional parameter list definition.

[param def] ::= "(" [param list] ")"

where [param list] is a list of declarations defining the 'formal' parameters. The declarations may be of any data type except array ; arrays may only be passed to a procedure as array name parameters.

E.g. routine SWOP(integer name P, 0)  
integer fn MAX(integer array name A, integer F, T)  
predicate EQUIV(record(FM)name LEFT, RIGHT)

Parameters have the same properties as any variables declared inside the procedure, except that the parameters are given values at the time the procedure is called. When a procedure is called 'actual' parameters must be supplied which match the formal parameters exactly in number, order, and type. Parameters are effectively assigned using "==" for those passed by name (E.g. integer name, real array name) and using "=" for those passed by value (E.g. string(10), integer).

For example assuming the declarations:

integer L, M, N  
real R  
integer array V(-7:7)  
record (FM) ONE, TWO

valid calls on the procedures mentioned in the previous example are:

SWOP(L, M)  
SWOP(V(L), V(M))  
N = MAX(V, -1, 0)  
M = MAX(V, L, 7)  
N = M if EQUIV(ONE, TWO)

N.B. IMP77 name type parameters are passed by reference and not by substitution (c.f. ALGOL 60).



### Procedure parameters

In addition to being able to pass variables to procedures it is possible to pass procedures as parameters. This is achieved by using the procedure heading as the 'declaration' of the formal parameter.

```
E.g. routine TRY(routine R(integer X))
      integer J
      R(J) for J = 1, 1, 10
      end
```

The routine TRY may now be called with a single parameter which must be the name of a routine which itself has one integer parameter. In this context the formal parameter names used to specify the parameters of a procedure parameter are otherwise ignored.

Note: If the routine try is itself to be passed as a parameter the heading of the receiving routine would be something like:

```
routine CHECK(routine X(routine Y(integer Z)))
```

and the call would be:

```
CHECK(TRY)
```

### Procedure specification

On occasions it may be necessary to use a procedure before it is possible (or desirable) to define it. For example, where two or more procedures call each other (mutual recursion) or where a procedure is to be defined externally (see External Linkage). As all identifiers must be declared before use, a procedure specification statement is introduced. This takes the form of the normal procedure heading with the keyword spec inserted before the procedure identifier.

E.g. routine spec MAX(real SIZE)

This has no effect other than declaring the identifier to be a procedure of the specified form which takes the given parameters. Except in the case of external procedure specifications the procedure must be defined later on in the block to which the spec is local.

For example:

```
routine spec B(integer X)
.
routine A(integer Y)
.
  B(Y-1)
.
end

routine B(integer X)
.
  A(X+3)
.
end
```

Note that the spec statement and the procedure heading must correspond, that is, the type and form of the statements must match, as must the type, form, order and number of any parameters.

## External linkage

A complete program may be divided into several separately compiled modules which are connected together in some way before (or possibly while) the program is executed. This linkage is requested by giving the prefix external to the relevant declarations. The keywords system and dynamic may be used in place of external; refer to the relevant implementation notes for details of the effect of these keywords.

### 1. external variables

An external variable has all the properties of an own variable, but is declared with the keyword own replaced by external.

```
external integer CHOICE=4, WAIT = -5
```

```
external real array MEAN(-6:6)
```

The identifiers are then available for use by any program that references them. A separately compiled module that requires to use any of these variables must first declare them using an external specification.

```
external integer spec WAIT, CHOICE
```

```
external real array spec MEAN(-6:6)
```

note i No initialisation may be given in an external specification.

ii External arrays must be one-dimensional and have constant bounds.

iii Even though all of the characters in the identifier of an external entity are significant to the compiler, system software might impose constraints on the number of characters significant for linkage purposes. Refer to the relevant implementation notes for system-dependent restrictions.

## 2. external procedures

A procedure may be made available to other modules by prefixing the procedure heading with the keyword external.

external routine TRIAL(string(63) S)

External procedure definitions may not be nested within any blocks.

If a module requires to use an externally defined procedure it must first supply an external procedure specification. For example:

external predicate spec LETTER(integer S)

This is similar to a procedure specification but only requires the specified procedure to have been defined by the time the module is executed.

An external ... spec may be given wherever other declarations would be valid.

### Alias

Any identifier being declared as external may be followed by alias [string const] where the string constant specifies the string to be used for external linkage. From within the module the external object will be identified in the usual way.

E.g. external real fn spec SIN alias "MATH\$DSIN" (real ARG)  
SX = SIN(0.3)

4

## Program file structure

A complete file of statements which may be processed by the compiler comprises a sequence of one or more blocks and is terminated by the statement:

endoffile

There may be no more than one begin block in this sequence (unless nested within other blocks). Such a begin block must be the last block. In this case the final two statements:

end  
endoffile

may be replaced by the single statement:

endofprogram

Declarations may be made global to these blocks with the restriction that variables must be own or external.

Examples of complete program files:

The null program:

endoffile

The most trivial program:

begin  
endofprogram

A more reasonable file:

```
owninteger IN=0, OUT=0
externalroutine GET(integername SYM)
    READSYMBOL(SYM)
    IN = IN+1
end
external routine PUT(integer SYM)
    PRINTSYMBOL(SYM)
    OUT = OUT+1
end
begin
    externalroutinespec PROCESS
    PROCESS
    WRITE(IN, 1)
    PRINTSTRING(" characters in")
    WRITE(OUT, 5)
    PRINTSTRING(" characters out")
    NEWLINE
endofprogram
```

## Permanent procedures

Each file processed by the compiler is conceptually prefixed by a set of declarations, which introduce the commonly used procedures, making them available to every file without any explicit action by the programmer. The compiler treats these declarations as being global to the whole file and hence the identifiers may be redeclared without error.

While the actual declarations may vary from machine to machine, the following are standard and may be assumed present:

|                            |   |
|----------------------------|---|
| <u>constant integer</u>    | NL = 10   |
| <u>routine</u>             | OPEN INPUT( <u>integer</u> STREAM, <u>string</u> (127) FILE)    |
| <u>routine</u>             | CLOSE INPUT   |
| <u>routine</u>             | SELECT INPUT( <u>integer</u> STREAM)                            |
| <u>routine</u>             | READSYMBOL( <u>name</u> S)                                      |
| <u>routine</u>             | SKIPSYMBOL  |
| <u>integer function</u>    | NEXTSYMBOL  |
| <u>routine</u>             | READ( <u>name</u> N)  |
| <u>routine</u>             | PROMPT( <u>string</u> (15) S)                                   |
| <u>routine</u>             | OPEN OUTPUT( <u>integer</u> STREAM, <u>string</u> (127) FILE)   |
| <u>routine</u>             | CLOSE OUTPUT  |
| <u>routine</u>             | SELECT OUTPUT( <u>integer</u> STREAM)                           |
| <u>routine</u>             | PRINTSYMBOL( <u>integer</u> N)                                  |
| <u>routine</u>             | PRINTSTRING( <u>string</u> (255) S)                             |
| <u>routine</u>             | WRITE( <u>integer</u> N, PLACES)                                |
| <u>routine</u>             | NEWLINE   |
| <u>routine</u>             | NEWLINES( <u>integer</u> N)                                     |
| <u>routine</u>             | SPACE   |
| <u>routine</u>             | SPACES( <u>integer</u> N)                                       |
| <u>integer function</u>    | REM( <u>integer</u> A, B)                                       |
| <u>long real function</u>  | FLOAT( <u>long real</u> N)                                      |
| <u>long real function</u>  | FRAC PT( <u>long real</u> L)                                    |
| <u>integer function</u>    | INT PT( <u>long real</u> L)                                     |
| <u>integer function</u>    | INT( <u>long real</u> L)  |
| <u>string(1)function</u>   | TOSTRING( <u>integer</u> SYMBOL)                                |
| <u>integer function</u>    | LENGTH( <u>string</u> (*) <u>name</u> S)                        |
| <u>byte integer map</u>    | CHARNO( <u>string</u> (*) <u>name</u> S, <u>integer</u> N)      |
| <u>string(255)fn</u>       | SUBSTRING( <u>string</u> (*) <u>name</u> S, <u>integer</u> F,T) |
| <u>record format</u>       | EVENT FM( <u>integer</u> EVENT, SUB, EXTRA)                     |
| <u>record(EVENT FM)map</u> | EVENT   |

|                         |                                       |
|-------------------------|---------------------------------------|
| <u>integer function</u> | ADDR( <u>name</u> V)                  |
| <u>integer map</u>      | INTEGER( <u>integer</u> ADDRESS)      |
| <u>byte map</u>         | BYTEINTEGER( <u>integer</u> ADDRESS)  |
| <u>short map</u>        | SHORTINTEGER( <u>integer</u> ADDRESS) |
| <u>real map</u>         | REAL( <u>integer</u> ADDRESS)         |
| <u>long real map</u>    | LONGREAL( <u>integer</u> ADDRESS)     |
| <u>string(*)map</u>     | STRING( <u>integer</u> ADDRESS)       |
| <u>record(*)map</u>     | RECORD( <u>integer</u> ADDRESS)       |
| <u>integer function</u> | SIZE OF( <u>name</u> N)               |
| <u>integer function</u> | TYPE OF( <u>name</u> N)               |

A definition of these procedures can be found in appendix 5.

## Events

During the execution of a program several (synchronous) events may occur, such as arithmetic overflow, array bound fault etc. (see Errors). Normally such events will cause the program to be terminated with an error report and possibly diagnostic information. However events may be trapped and used to control the subsequent execution of the program.

The first non-declarative statements of any block may be of the form:

```
on event [event list] start  
    (on-body statements)  
finish
```

where [event list] is a list of integer constants in the range 0 to 15 inclusive, representing the events to be trapped, or an asterisk (\*) in which case all events are to be trapped.

On entry to the block the on-body is skipped and execution continues from the statements following the finish. If an event specified in the [event list] is signalled during the execution of the statements between the finish of the on event group and the end of the block, control will be passed to the on-body (and may well pass through the finish to the following statements). If the event is not trapped in the current block a 'return' is forced and the event is signalled in the new block at the point from which the old block was entered. The process is repeated until either the event is trapped or the outermost block of the program is reached, in which case the event is reported as a fault and execution terminates.

Note that some events may or may not be signalled automatically in certain implementations or when the program has been compiled with the compile-time checks inhibited. Refer to the relevant implementation notes for details.



## Signalling events

At any time during the execution of a program an event may be signalled by executing an instruction of the form:

signal event [n][sub]?

[n] ::= [integer expression]  
[sub] ::= ", " [integer expression] [extra]?  
[extra] ::= ", " [integer expression]

The instruction causes event [n] to be signalled with sub-event (default zero) and extra information (default zero). The value of [n] must be in the range 0 to 15 inclusive.

signal event 15 (event 15,0,0)

signal event 14,7 if X < 0 (event 14,7,0)

signal event 13,1,Y if Y # 0 (event 13,1,Y)

Note i In both the on and signal statements the keyword event is optional and may be omitted.

ii An event signalled inside an incarnation of an on-body will never be trapped into that incarnation. Instead the search for a trap will start from the previous block.

The pre-defined record map EVENT provides access to a system-defined record containing information about the last event to have been signalled. While the exact definition of the record may vary from implementation to implementation the following fields will always be present:

record format EVENT FM(integer EVENT, SUB, EXTRA)

If no event has been signalled these fields will each contain the value zero.

## Control transfer instructions

### Labels and Jumps

#### 1. Simple Labels

Any statement, excluding declarations, may be given one or more simple labels. Optionally, the labels may be declared at the head of the block in which they are to be used, with the declaration taking the form:

label [idents]

e.g. label NEXT, ERROR1, ERROR2

Each label is located by writing it followed by a colon to the left of the statement to which it refers:

```
NEXT:          P = P+1 if P < 0
ERROR1:ERROR2:FAULTS = FAULTS+1
```

Control is passed to a labelled statement by executing a jump instruction of the form:

"->" [id]

E.g. -> NEXT  
->ERROR1 if DIVISOR = 0

#### 2. Switch Vectors

A vector of labels may be declared in a similar manner to a one-dimensional array, using the declarator switch. The vector must have constant bounds.

```
switch SW(4:9)
switch S1, S2(1:10), S3(11:20)
```

Once declared, switch labels may be located in the same way as simple labels, the particular label required being selected by an integer constant.

```
SW(4):          CHECK VALUE(1)
SW(6):SW(7): ERROR FLAG = 1
LAST: SW(9): (all finished)
```

An asterisk (\*) may be used when locating a switch label to define any elements within the vector which would otherwise be undefined.

```
switch LET('a':'z')
```

LET('e'):LET('e'):LET('i'):LET('o'):LET('u'):

(deal with vowels)

LET(\*):( all the rest i.e. consonants)

Control is passed to one of these statements by executing instructions of the form:

"->" [switch id] "(" [integer expression] ")"

E.g. ->SW(N) if N > 0  
->SW(N+2)  
->SW(6)

Note i) Not all of the declared switch labels need be located (in the previous examples SW(5) and SW(8) are undefined) but an error will occur at run time if an attempt is made to jump to a non-existent switch label.

ii) Labels may be used before they are located.

-> MISSING if HERE = 0

MISSING:

iii) The scope of both types of label is limited to the block in which they are defined, excluding any blocks defined therein. That is labels cannot be global to a block and therefore it is not possible to jump into or out of a block.

iv) The identifiers used for labels must not conflict with other local identifiers.

v) The results of entering a for loop other than via the for statement are undefined.

### Other control instructions

stop This is an abbreviation for:

signal event 0,0,0

and usually results in the normal termination of the program, although the event may be trapped in the usual way.

monitor This instruction passes control to the run-time diagnostic package which should then generate a trace of the state of the program. On implementations without a diagnostic package monitor is a null operation. Following the trace the previous flow of control is resumed.

## Implementation-dependent features

The following features are highly dependent on the particular implementation of the language and the machine on which the programs are to be executed. If used at all they should be used with extreme care.

### Constant pointers

Constant name-type variables may be declared and initialised to point at fixed machine addresses.

e.g. constant integer name CLOCK == 16\_3C .

subsequent reference to CLOCK will be identical to references to INTEGER(16\_3C)

### Address Modifiers

References to simple pointer variables may be followed by an integer expression enclosed in square brackets: e.g. N[2]. The effect of this is effectively to interpret the pointer variable as pointing to the zero'th element of an infinite one-dimensional array of simple objects of the type of the pointer variable. The value of the integer expression is then used to index into this array to select a particular simple variable.

E.g. integerarray A(1:12)  
integername N, M  
N == A[4]  
M == N[3] (same as M == A[7])  
N[-1] = 0 (same as A[3] = 0)

### Option

The statement: Option [string:constant] may be used to select implementation-defined options. Refer to the relevant implementation notes for details.

### Control

The statement: control [integer:constant] may be used to set implementation-dependant compiler options. Refer to the relevant implementation notes for details.

### Diagnose

The statement: diagnose [integer:constant] may be used to control the production of diagnostic information about the operation of the compiler itself. Refer to the relevant compiler documentation for details.

## Machine code

There are two methods of adding in-line machine code sequences to an IMP77 program.

1    "+=" [integer:constant]

Statements of this form plant the given integer constant as an instruction. 4

2    "\*" [machine-code]

Statements of this form enable pseudo assembler statements to be included which reference the program-declared objects. Refer to the relevant implementation notes for details of the syntax of [machine-code].

## Appendix 1

### A note on the grammar

- :=** - introduces the definition of a phrase
- ?** - indicates a rule is optional
- \*** - indicates zero or more instances of a rule
- +** - indicates one or more instances of a rule
- ,** - separates alternatives
- < >** - define the scope of the above items
  
- [ ]** - enclose phrase identifiers
- " "** - enclose literal strings  
keywords are underlined

- E.g.
- "A" <"B" "C">?** -> A  
or ABC
  
  - "A" <"B" "C">\*** -> A  
or ABC  
or ABCBC etc.
  
  - "A" <"B", "C">** -> AB  
or AC
  
  - "A" <"B", "C">\*** -> A  
or AB  
or AC  
or ABB  
or ABC  
or ACB etc.
  
  - "A" <"B", "C">+** -> AB  
or AC  
or ABB  
or ABC  
or ACB etc.

## Appendix 2

### Compiler messages

During the compilation of a program the compiler may generate messages which are generally sent to the listing file and possibly to an interactive report stream. These messages are either error indications or warnings.

#### Errors

An error message indicates that the current statement does not obey the rules of the language or that a necessary statement has been omitted from the previous statement sequence.

Once an error has been detected the compiler ignores the rest of the faulty statement and continues compiling with the next. This may result in consequential errors which will disappear once the original error is corrected. For example the compiler will object to the following declaration:

integer A,B,,C,D

The extra comma will cause the declaration of C and D to be ignored and so subsequent references to them will be faulted (NOT DECLARED). In general it is good practice to correct errors in the order in which they occur in the listing.

Error messages start with an asterisk (\*) and where possible they contain a marker which points into the offending statement at the position at which the compiler detected the error.

The error messages are:

- Atom            An unknown atomic element has been encountered. This is commonly caused by mistyping a keyword.  
E.g. intger, rutine, strat etc.
- Bounds          The size of an array or switch vector is negative.  
E.g. switch S(10:1)  
own integer array X(-1:-10)
- Context        An otherwise correct statement has been given in a context where it is meaningless.  
E.g. exit not contained within a cycle - repeat.  
return not inside a routine.

#### Context [ID]

[ID] is the identifier of a record format which has been used to define a record or record array within the definition of [ID] itself. Note that it is valid to declare record name and record array name variables in this context.

E.g. record format F(integer X, record(F) Y)



- Duplicate** A local identifier is being redeclared.  
E.g. real SUN, MON, TUE, WED, THUR, FRI, SAT, SUN
- Form** An unexpected atom has been encountered. This is usually caused by the omission of an atom or the insertion of an extra atom.  
E.g. integer A, B, , C  
PRINTSTRING("BYE") NEWLINE (semicolon missing)
- Format** Illegal use of a record with a format which is currently undefined.  
E.g. recordformatspec FM  
record(FM)name PT  
PT = 0
- Index** A switch label has been given an index outwith the declared bounds.  
E.g. switch S(1:5)  
S(6):
- Match** The definition of a procedure does not match a previous specification.  
E.g. routinespec PROC(integer X)  
routine PROC(real X)
- Not a variable**  
An attempt has been made to use an object with a constant value in a context where it could be modified. This is commonly caused by using named constants as though they were variables.  
E.g. constant integer TEN = 10  
TEN = TEN+1
- Not declared**  
An undeclared identifier has been used. This error is also commonly generated by omitting the percent from the beginning of certain keywords (usually: if, finish, end repeat).  
E.g. integer SWOP  
SWAP = 0
- Note the following common error:  
string(7)name P  
This declares a simple string variable "namep" instead of what was probably intended: a string pointer variable "P". The reason is that the keyword "name" has not been underlined.

**Order** This is similar to Context but is reserved for statements which are given before they are valid or after other statements which invalidate them. There are three common causes:

1 The declaration of variables other than own or external global to the outermost blocks of a program.

E.g. integer X  
begin  
.....

2 The declaration of an array following a label.

E.g. begin  
LAB: integerarray A(1:5)

3 Declarations following an on statement.

E.g. on event 7 start  
stop  
finish  
integerarray XX(2:7)

**Size** A constant has a value outwith the permitted range.  
E.g. string(300) S

**Too complex**

The statement is too large or complicated to be analysed. This error is quite rare and can invariably be cured by splitting the offending statement into two or more simpler statements. Note that putting redundant continuations (g) at the end of each line of a large list of array initialising constants may provoke this error.

**Type** The type of a given variable or expression does not match the type of object required by the context.

E.g. integer X  
byte integer name P  
P == X

or X = 1.2

**%begin missing**

An and has been found which has no matching begin (or routine etc.).

**%cycle missing**

A repeat has been found which does not have a matching cycle in the current block.

**%end missing**

The end of the program file has been reached before all blocks have been terminated.

**%finish missing**

The end of a block has been reached and it contains a start which has no matching finish.

**%repeat missing**

The end of a block has been reached and it contains a cycle which has no matching repeat.

**result missing**

This occurs at the end of a function, map, or predicate when it is not manifestly evident that control must be passed back from the procedure at run-time.

E.g. integer function F(integer X)  
result = 0 if X <= 0  
end

or predicate EVEN(integer N)  
true if N&1 = 0  
false if N&1 # 0  
(this will give the error as the compiler)  
(is unlikely to be clever enough to detect)  
(the 'completeness' of the conditions)  
end

**%start missing**

The compiler has found a finish for which there is no matching start.

**"[id]" missing**

The object identified by [id] has been specified in the preceding block (by a spec or a label statement) but has not subsequently been defined.

E.g. begin  
routine spec CHECK  
CHECK  
end

## Warnings

A warning indicates that the compiler has detected something which, although not an error in itself, may indicate logical errors elsewhere.

Warning messages start with a question mark (?) and are:

**Access** Control cannot reach the current statement. That is, the previous executable statement was or implied an unconditional transfer of control, and the current statement is not labelled.

**Non-local** The control variable of a for loop is not local to the current block. Such use of globals can lead to unexpected infinite loops:

```
E.g. integer P
      routine R
          for P = 1,1,10 cycle
              .....
          repeat
      end

      R for P = 1,1,20
```

**[id] unused** The given identifier has been declared but never used.

## Catastrophic errors

Under certain circumstances the compiler will be unable to continue after discovering an error, usually because the compiler's tables will have been filled or corrupted.

These errors are:

### Compiler error

There is a fault in the compiler itself.

### Switch vector too large

A switch vector has been declared with a very large number of elements.

### Too many names

The compiler has no room left to describe new named objects.

### Dictionary full

The compiler has no room left to hold the text of new identifiers. This is usually caused by declaring a large number of long identifiers.

### Input ended

The end of an input file has been reached without endoffile or endofprogram being detected. This is most commonly caused by mistyping endofprogram, or leaving out a closing string quote.

Some compilers may choose to treat this as a warning and complete the compilation.

### String constant too long

A string constant has been discovered to contain more than 255 characters. This is commonly caused by leaving out the terminating quote.

### Included file .... does not exist

The compiler cannot gain access to a file specified in an include statement.

### Program too complex

The program is so complex that the compiler has filled its internal tables.

### Too many faults!

This is generated when the compiler discovers a high fault rate in the program. It is used to terminate compilations which would otherwise generate a large number of faults. This is commonly caused by faulty declarations, or by attempting to compile something which is not an IMP77 program.

### Appendix 3

#### Sample program listings

```
1 %begin
2   %constinteger PAGE SIZE = 63,    (lines on a page)
3+   FF = 12                          (ASCII Form Feed)
4   %integer SYM, LINES LEFT = PAGE SIZE, LINE = 0
5   %on %event 9 %start    (end of file)
6     NEWLINE
7     %stop
8   %finish
9
10  %cycle
11    READSYMBOL(SYM)    (provoke input ended before
12                      (printing the line number)
13    LINE = LINE+1
14    WRITE(LINE, 3); SPACE
15    %cycle
16    PRINTSYMBOL(SYM)
17    %exit %if SYM = NL
18    READSYMBOL(SYM)
19    %repeat
20    LINES LEFT = LINES LEFT-1
21    %if LINES LEFT = 0 %start
22    LINES LEFT = PAGE SIZE; PRINTSYMBOL(FF)
23    %finish
24  %repeat
25 %endofprogram

24 Statements compiled
```

```

1 %begin
2   %begin
3     %realname Q
4     %integer VALUE, X, X
*                                     | duplicate
5     %string(256) S
* size
6     %switch SA(1:4), SB(5:2)
* bounds
7     %routine %spec CHECK
8     %integer %functionspec KEY(%integer LOCK)
9     %if X = 4 %stary
*                                     | atom
10    VALUE = KEY
*                                     | form
11    X = VALUW
*                                     | not declared
12    X = X+1
13  ee(5):
* index
14    VALUE = 0
15    %finish
* %start missing
16    %exit %if X < 0
* context
17    %stop
18    X = 0
? access
19    %on %event 4 %start
* order
20    %integerfn KEY(%real LOCK)
* match
21    NEWLINE
22    PRINTSYMBOL('=') %for X = 1, 1, 12
? Non-local
23    %end
* result missing
? LOCK unused
24    Q == VALUE
*                                     | type
25    X = Q&7
*                                     | type
26 %endofprogram
* %end missing
* %finish missing
* CHECK missing

```

Program contains 17 faults

## Appendix 4

### Standard Events

| <u>event</u> | <u>sub-class</u> | <u>meaning (+extra)</u>         |
|--------------|------------------|---------------------------------|
| 0            |                  | <u>TERMINATION</u>              |
|              | -1               | abandon program                 |
|              | 0                | <u>stop</u>                     |
|              | >0               | user generated error            |
| 1            |                  | <u>OVERFLOW</u>                 |
|              | 1                | integer overflow                |
|              | 2                | real overflow                   |
|              | 3                | string overflow                 |
|              | 4                | division by zero                |
|              | 5                | truncation                      |
| 2            |                  | <u>EXCESS RESOURCE</u>          |
|              | 1                | not enough store                |
|              | 2                | output exceeded                 |
|              | 3                | time exceeded                   |
| 3            |                  | <u>DATA ERROR</u>               |
|              | 1                | data transmission error         |
| 4            |                  | <u>INVALID DATA</u>             |
|              | 1                | symbol in data (+symbol)        |
| 5            |                  | <u>INVALID ARGUMENTS</u>        |
|              | 1                | for cannot terminate            |
|              | 2                | illegal parameter type          |
|              | 3                | array inside-out                |
|              | 4                | string inside-out               |
|              | 5                | illegal exponent (+exponent)    |
| 6            |                  | <u>OUT OF RANGE</u>             |
|              | 2                | array bound fault (+index)      |
|              | 3                | switch bound fault (+index)     |
|              | 4                | illegal event signal            |
|              | 5                | CHARNO out of range (+index)    |
|              | 6                | TOSTRING out of range (+symbol) |
|              | 7                | Illegal shift (+shift)          |
| 7            |                  | <u>RESOLUTION FAILS</u>         |
| 8            |                  | <u>UNDEFINED VALUE</u>          |
|              | 1                | unassigned variable             |
|              | 2                | no switch label (+index)        |
|              | 3                | for variable corrupt            |
| 9            |                  | <u>INPUT/OUTPUT ERROR</u>       |
|              | 1                | input ended                     |
|              | 2                | illegal stream (+stream)        |
|              | 3                | file system error (+error code) |
| 10           |                  | <u>LIBRARY PROCEDURE ERROR</u>  |
| 11 - 15      |                  | <u>GENERAL PURPOSE</u>          |



## Appendix 5

### Permanent procedures

In the definition of I/O procedures the term "symbol" is used to represent the unit of transfer. Normally this is eight bits but this may vary on certain machines. Refer to the particular implementation notes for details.

#### routine OPEN INPUT(integer STREAM, string(127) FILE)

**Effect:** The specified stream is first closed (See CLOSE INPUT) and then is associated with the specified file (or device). This routine does not leave the specified stream selected unless it was already selected.

**Errors:** Event 9,2 is signalled if STREAM does not correspond to a valid input stream.  
Event 9,3 is signalled if the specified FILE (or device) cannot be accessed.

#### routine SELECT INPUT(integer STREAM)

**Effect:** The input stream identified by the parameter STREAM is made the currently selected input stream. This may be considered as setting a default parameter which is implied by all other input stream handling procedures, and has no effect on either the old or newly-selected current input stream. The currently selected input stream is unaffected if an error is signalled.

**Errors:** Event 9,2 is signalled if STREAM does not correspond to a valid input stream. Refer to the relevant implementation notes for details.

#### routine READSYMBOL(name N)

**Effect:** The next symbol is taken from the currently selected input stream and is assigned to the parameter.

**Errors:** Event 9,1 is signalled if the end of the currently selected input stream has been reached.  
Event 5,5 is signalled if the parameter does not correspond to a simple variable which may be assigned an integer value.

routine SKIPSYMBOL

Effect: This routine may be considered as:

routine SKIPSYMBOL  
  integer DUMMY  
  READSYMBOL(DUMMY)  
end

Errors: As for READSYMBOL.

integer function NEXTSYMBOL

Effect: A copy of the next symbol in the currently selected input stream is returned as the result. The symbol is not removed from the input stream and can be accessed by another call on NEXTSYMBOL or a call on READSYMBOL.

Errors: Event 9,1 is signalled if the end of the currently selected input stream has been reached.

routine READ(name N)

Effect: This routine is used to input integer, real, or string values from the currently selected input stream. Symbols are input until one is found which is not a space and is not a control character. Depending on the type of the parameter N the following actions are then taken:

N is a string variable.

Symbols are input and built up into a string which is assigned to N. The string is terminated by the first space or newline symbol.

N is an integer variable.

Symbols are input and built up into an integer value. The input is terminated by any character (including spaces and newlines) which does not conform to the informal syntax:

[input] ::= [sign]? [digit]+  
[sign] ::= "+", "-"  
[digit] ::= "0", "1", "2", ..., "9"

Note that some implementations may also allow the BASE\_VALUE form of integer constants to be input using READ.

N is a real variable.

Symbols are input and built up into a real value. The input is terminated by any character (including spaces and newlines) which does not conform to the informal syntax:

[input] ::= [sign]? [digit]\* [frac]? [exp]?  
[frac] ::= "." [digit]+  
[exp] ::= "e" [space]\* [sign]? [digit]+

In all cases the terminating character is not removed from the input stream, but is left available as NEXTSYMBOL.

**Errors:** Event 9,1 is signalled if the end of the current input stream is reached while leading spaces and newlines are being skipped, or when the first significant character of the input is being input. The end of the input stream may be used to terminate the input without causing the event to be signalled. In the case of real constants the exponent following "e" may be considered input using a recursive call of READ.

Event 5,5 will be signalled if the parameter does not correspond to an integer, real, or string variable.

Event 1, x will be signalled if the value input is too large to be held in the parameter (integer: x=1, real: x=2, string: x=3).

Event 4,1 will be signalled if the input does not conform to the required syntax.

#### routine CLOSE INPUT

**Effect:** The currently selected input stream is closed. If the currently selected input stream is not stream 0 the stream is left associated with the null file.

#### routine OPEN OUTPUT(integer STREAM, string(127) FILE)

**Effect:** The specified stream is first closed (see CLOSE OUTPUT). The stream is then associated with the specified file (or device). This in general implies creating or overwriting the file. This routine does not leave the specified stream selected unless it was already selected.

**Errors:** Event 9,2 is signalled if STREAM does not correspond to a valid output stream.

Event 9,3 is signalled if the specified FILE cannot be opened for output.

#### routine SELECT OUTPUT(integer STREAM)

**Effect:** The output stream identified by STREAM is made the currently selected output stream. Refer to SELECT INPUT.

**Errors:** Event 9,2 is signalled if STREAM does not correspond to a valid output stream. Refer to the relevant implementation notes for details.

routine PRINT SYMBOL(integer N)

Effect: The symbol contained in N is sent to the currently selected output stream.

Errors: Event 2,2 is signalled if the output stream is full.

routine PRINT STRING(string(255) S)

Effect: The contents of the parameter are sent to the currently selected output stream. The routine may be considered as:

```
routine PRINT STRING(string(255) S)
      integer N
      PRINTSYMBOL(CHARNO(S, N)) for N = 1,1,LENGTH(S)
end
```

Errors: As for PRINTSYMBOL.

routine WRITE(integer N, PLACES)

Effect: The value contained in the parameter N is sent to the currently selected output stream as a sequence of characters representing its decimal value. PLACES is used to control the size of the output field. The precise effect of WRITE may be defined assuming a hypothetical string function DIGITS which returns a string of decimal digits (with no leading zeros) corresponding to the value of its parameter, e.g. DIGITS(123) = "123".

Then:

```
routine WRITE(integer N, PLACES)
      string(255) S
      S = DIGITS(IN)
      if N < 0 start
        S = "-" . S
      else if PLACES > 0
        S = " " . S
      finish
      if PLACES <= 0 then PLACES = -PLACES end
        else PLACES = PLACES+1
      S = " " . S while LENGTH(S) < PLACES
      PRINTSTRING(S)
end
```

Errors: As for PRINTSYMBOL.

routine NEWLINE

Effect: A newline character (ASCII code 10) is sent to the currently selected output stream:

```
routine NEWLINE  
  PRINT SYMBOL(NL)  
end
```

Errors: As for PRINT SYMBOL.

routine NEWLINES(integer N)

Effect: A number of newlines is sent to the currently selected output stream. This routine has no effect if N is less than or equal to zero.

```
routine NEWLINES(integer N)  
  while N > 0 cycle  
    N = N-1  
    NEWLINE  
  repeat  
end
```

Errors: As for PRINT SYMBOL.

routine SPACE

Effect: A space character is sent to the currently selected output stream:

```
routine SPACE  
  PRINT SYMBOL(' ')  
end
```

Errors: As for PRINT SYMBOL.

routine SPACES(integer N)

Effect: A number of space characters is sent to the currently selected output stream. This routine has no effect if N is less than or equal to zero:

```
routine SPACES(integer N)  
  while N > 0 cycle  
    N = N-1  
    SPACE  
  repeat  
end
```

Errors: As for PRINTSYMBOL.

routine CLOSE OUTPUT

Effect: similar to CLOSE INPUT.

integer function REM(integer A, B)

Effect: The function returns the remainder of dividing A by B. The sign of the remainder is the same as the sign of the dividend (A):

integer function REM(integer A, B)

result = A-A/B\*B

end

Errors: Event 1,4 is signalled if B=0.

long real function FLOAT(long real N)

Effect: The parameter is converted into its floating-point equivalent and returned as the result.

Errors: None

long real function FRAC PT(long real L)

Effect: The fractional part of the parameter L is returned as the result. The fractional part is calculated as L-INTPT(L).

Errors: None

integer function INT PT(long real L)

Effect: The function returns the integer part of L, any truncation being towards minus infinity. Hence INT PT(-1.5)=-2.

Errors: Event 1,1 is signalled if the result cannot be held in an integer variable.

integer function INT(long real L)

Effect: This function returns the nearest integer to the parameter.

integer function INT(long real L)

result = INT PT(L+0.5)

end

Errors: Event 1,1 is signalled if the result cannot be held in an integer variable.

string(1)function TOSTRING(integer SYMBOL)

Effect: The parameter SYMBOL is converted into the equivalent one-character string, which is then returned as the result.

Hence: TOSTRING('a') = "a"

Errors: Event 6,5 is signalled unless  $0 \leq \text{SYMBOL} \leq 255$ .

integer function LENGTH(string(\*)name S)

Effect: The current length of the string is returned.

byte integer map CHARNO(string(\*)name S, integer P)

Effect: A reference to the byte integer holding the P'th character of the string is returned.

Errors: Event 6,5 is signalled unless  $1 \leq P \leq \text{LENGTH}(S)$

string(255)function SUBSTRING(string(\*)name S, integer F,T)

Effect: The sequence of characters from positions F to T inclusive in the string S is returned as the result. The length of this string will be T-F+1.

E.g. if S = "1234567890"  
then SUBSTRING(S, 4, 7) = "4567"

The position of the first character in the string must satisfy the condition:  $1 \leq F \leq \text{LENGTH}(S)+1$ , and the position of the final character must satisfy the condition:  $0 \leq T \leq \text{LENGTH}(S)$ . The length of the result must be greater than or equal to zero. The function may be thought of as:

```
string(255)fn SUBSTRING(string(*)name S, integer F,T)  
  string(255) TEMP  
  integer P  
  TEMP = ""  
  for P = F, 1, T cycle  
    TEMP = TEMP.TOSTRING(CHARNO(S, P))  
  repeat  
  result = TEMP  
end
```

Errors: Event 5,4 is signalled unless:

```
   $1 \leq F \leq \text{LENGTH}(S)+1$   
and  $0 \leq T \leq \text{LENGTH}(S)$   
and  $T-F+1 \geq 0$ 
```

recordformat EVENT FM(integer EVENT, SUB, EXTRA)  
record(EVENT FM)map EVENT

Effect: This map returns a reference to a system-provided record which contains the parameters of the last event to have been signalled. If no event has been signalled all the fields of the record are set to zero. Implementations are free to add extra fields to the record.

Errors: None.

integer function ADDR(name V)

Effect: The machine address of the variable V is returned as an integer value.

Errors: None.

integer map INTEGER(integer ADDRESS)

Effect: This map returns a reference to the integer variable at machine address ADDRESS.

Errors: Machine-specific errors can occur if ADDRESS does not satisfy certain range and alignment considerations.

byte map BYTEINTEGER(integer ADDRESS)

Effect: This map returns a reference to the byteinteger variable at the given address.

Errors: As for INTEGER.

short map SHORTINTEGER(integer ADDRESS)

Effect: This map returns a reference to the shortinteger variable at the given address.

Errors: As for INTEGER.

real map REAL(integer ADDRESS)

Effect: This map returns a reference to the real variable at the given address.

Errors: As for INTEGER.



long real map LONG REAL(integer ADDRESS)

Effect: This map returns a reference to the longreal variable at the given address.

Errors: As for INTEGER.

string(\*)map STRING(integer ADDRESS)

Effect: This map returns a reference to the string variable at the given address.

Errors: As for INTEGER.

record(\*)map RECORD(integer ADDRESS)

Effect: This map returns a reference to the record variable (of indeterminate format) at the given address.

Errors: As for INTEGER.

integer function SIZE OF(name N)

Effect: The number of storage units occupied by the given variable is returned. The unit is machine-dependent but commonly is a byte.

Errors: None.

integer function TYPE OF(name N)

Effect: A code is returned which indicated the type of the parameter. The code values are:

- 0 - unknown type
- 1 - integer
- 2 - real
- 3 - string
- 4 - record
- 5 - byteinteger
- 6 - shortinteger
- 7 - longinteger
- 8 - longreal
- 9 - array
- 10 - label

Appendix 6

Variant and archaic forms

| Standard form       | Variant                             |
|---------------------|-------------------------------------|
| <u>byteinteger</u>  | <u>byte</u>                         |
| <u>constant</u>     | <u>const</u>                        |
| <u>function</u>     | <u>fn</u>                           |
| <u>map</u>          | <u>name function</u> <u>name fn</u> |
| <u>shortinteger</u> | <u>short</u>                        |
| #                   | <>                                  |
| -                   | /                                   |
| ^                   | //                                  |
| **                  | //                                  |

Appendix 7

ASCII character set

|    |         |    |       |    |   |     |     |
|----|---------|----|-------|----|---|-----|-----|
| 0  | NUL     | 32 | space | 64 | @ | 96  | `   |
| 1  | SOH     | 33 | !     | 65 | A | 97  | a   |
| 2  | STX     | 34 | "     | 66 | B | 98  | b   |
| 3  | ETX     | 35 | #     | 67 | C | 99  | c   |
| 4  | EOT     | 36 | \$    | 68 | D | 100 | d   |
| 5  | ENQ     | 37 | %     | 69 | E | 101 | e   |
| 6  | ACK     | 38 | &     | 70 | F | 102 | f   |
| 7  | BEL     | 39 | '     | 71 | G | 103 | g   |
| 8  | BS      | 40 | (     | 72 | H | 104 | h   |
| 9  | HT      | 41 | )     | 73 | I | 105 | i   |
| 10 | LF (NL) | 42 | *     | 74 | J | 106 | j   |
| 11 | VT      | 43 | +     | 75 | K | 107 | k   |
| 12 | FF      | 44 | ,     | 76 | L | 108 | l   |
| 13 | CR      | 45 | -     | 77 | M | 109 | m   |
| 14 | SO      | 46 | .     | 78 | N | 110 | n   |
| 15 | SI      | 47 | /     | 79 | O | 111 | o   |
| 16 | DLE     | 48 | 0     | 80 | P | 112 | p   |
| 17 | DC1     | 49 | 1     | 81 | Q | 113 | q   |
| 18 | DC2     | 50 | 2     | 82 | R | 114 | r   |
| 19 | DC3     | 51 | 3     | 83 | S | 115 | s   |
| 20 | DC4     | 52 | 4     | 84 | T | 116 | t   |
| 21 | NAK     | 53 | 5     | 85 | U | 117 | u   |
| 22 | SYN     | 54 | 6     | 86 | V | 118 | v   |
| 23 | ETB     | 55 | 7     | 87 | W | 119 | w   |
| 24 | CAN     | 56 | 8     | 88 | X | 120 | x   |
| 25 | EM      | 57 | 9     | 89 | Y | 121 | y   |
| 26 | SUB     | 58 | :     | 90 | Z | 122 | z   |
| 27 | ESC     | 59 | ;     | 91 | [ | 123 | {   |
| 28 | FS      | 60 | <     | 92 | \ | 124 |     |
| 29 | GS      | 61 | =     | 93 | ] | 125 | }   |
| 30 | RS      | 62 | >     | 94 | ^ | 126 | ~   |
| 31 | US      | 63 | ?     | 95 | _ | 127 | DEL |