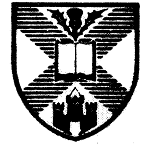
P.D. SCHOFIELD

University of Edinburgh



Department of Computer Science

The IMP-77 Language

by

Peter S. Robertson

Copy 2

INTERNAL REPORT CSR-19-77

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

December, 1977
Revised May, 1979

REFERENCE ONLY

THE IMP-77 LANGUAGE

As implemented by

PETER S. ROBERTSON
DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF EDINBURGH

A REFERENCE MANUAL

first edition: November 1977
second edition: April 1979

+	2-1
-	2-1
->	5-2, 10-1
.	2-2
/	2-1
//	2-1
:	10-1
<	6-1
<-	5-2, 13-5
<<	2-2
<=	6-1
=	5-1, 6-1
==	5-1, 6-1
>	6-1
>=	6-1
>>	2-2
@	1-5
\	2-1
\\	2-1
_	1-4, 3-4
{	1-2
¦	2-1
}	1-2
~	2-2
~=	6-1

INTRODUCTION

IMP-77 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 IMP 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.

This manual describes the language as implemented initially in version 6 of the compiler.

name 3-1, 5-1, 8-7
newline 1-1, 1-2
NL 1-4
not 6-2
null statement 1-2
null statements 4-2
on 12-1
or 3-3, 6-1
own 4-1
parameters 8-6
pointer variables 3-1
precision 13-5
predicate 8-5
quotes 1-1
real 3-1
reals long 13-5
reals normal 13-5
record 3-1, 3-3
record assignment 5-2
recordformat 3-3
recordformats 1-2
repeat 7-1, 8-1
repetition count 4-1
result= 8-4
result== 8-4
return 8-3
routine 8-3
semicolon 1-2
short 13-5
signal 8-5, 12-3
spaces 1-1
spec 9-2, 11-2
start 6-3, 8-1
stop 10-3
string 3-1
switch 10-1
system 11-1
termination 1-2
then 6-3
true 6-1, 8-5
unless 6-3
until 7-2
upper case 1-1
while 7-1
! 1-2, 2-2
!! 2-2
" 1-1, 1-5
6-1
6-1
% 1-1
& 2-2
' 1-1, 1-4
(*) 3-1, 4-1, 10-1
* 2-1

INDEX

alias	11-3
and	6-1, 7-3
array	3-2
array name	3-2
begin	8-1
byte	13-5
c	1-3
character constants	1-4
comment	1-2
comments	1-2
concatenation	2-2
conditions	6-1
const	4-1
constant	4-1
constant expressions	1-4
continue	7-3
cycle	7-1, 8-1
dynamic	11-1
else	6-3
end	8-1, 8-3, 8-4, 8-5
end of file	1-3, 11-1
end of list	1-3
end of program	8-1
error messages	13-2
event	12-1, 12-3
exit	7-3
external	9-2, 11-1
false	6-1, 8-5
finish	6-3, 8-1
FLOAT	2-1
fn	8-4
for	7-1, 10-2
forward references	3-3, 9-2, 11-2
frozen	4-2
function	8-4
identifiers	1-2
if	6-3
include	1-3
initialisation	4-1
instructions	1-3
INT	2-1
integer	3-1
INTPT	2-1
jumps	10-1
labels	1-2, 10-1
list	1-3
long	13-5
loops	7-1
lower case	1-1
map	8-4
modulus	2-1
monitor	10-3

CONTENTS

1-1	Program layout conventions
1-2	Statements
1-4	Constants
2-1	Expressions
3-1	Declarations
4-1	Own variables
4-2	Constant identifiers
4-2	Frozen variables
5-1	Assignment
5-2	Record assignment
5-3	String resolution
6-1	Conditions
6-3	Conditional groups
7-1	Repetition
8-1	Block structure
8-1	Begin blocks
8-2	Local and global variables
8-3	Procedures
8-6	Parameters
8-7	Procedures as parameters
8-7	General type parameter
8-9	Procedure specification
9-1	Control transfer instructions
10-1	External linkage
10-3	Alias
10-3	Predefined procedures
11-1	Events
Appendix 1	A note on the grammar
Appendix 2	Compiler messages
Appendix 3	Sample programs
Appendix 4	Data precision specification
Appendix 5	IMP keywords
Appendix 6	Comparison with EMAS IMP

2. Features not implemented

print text

until .... cycle

array format

implied multiplication

3. Changed Features

'AA' instead of M'AA'

16_1A2 instead of X'1A2'

procedure parameter specification

record(F) R instead of record R(F)

SUBSTRING instead of FROMSTRING

termination of comments

"\" or "\\" instead of "**"

own initialisation

type checking for record operations

external .. spec instead of extrinsic ..

events instead of fault trapping

/ gives a real result

string resolution

APPENDIX 6

COMPARISON WITH EMAS IMP

1. New Features

for

repeat until

continue

predicate

include

frozen

alias

"==" in conditions

integer array (4) name

finish else if ...

optional use of finish .... start

lower case input

{comments}

(*) in owns, switches, records, and strings

constant

constant expressions

function

not

record function

record map

alternative formats for records

embedded comments

PROGRAM LAYOUT CONVENTIONS

An IMP program is a sequence of statements constructed using the ASCII character set extended with an underlined alphabet. Underlined letters, which are used to form keywords, are generated using the shift character percent (%), which is defined as underlining the following letters, the 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)array name P
```

In this manual, keywords are in lower case and underlined.

Newline

The NEWLINE (or LINE-BREAK) character is ASCII character 10 (LF). Newlines are always significant (see Termination).

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 upper case letters.

Quotes

Several language constructions call for one or more characters 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 e.g. 'A'
"   -   string quote    e.g. "FRED"
```

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"" dog" -      a string of 11 characters
```

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

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.

Except in the cases of simple labels and recordformats (q.v.) all identifiers must be declared before they may be used (see Declarations).

STATEMENTS

A STATEMENT is a sequence of atomic elements (keywords, constants, identifiers etc.) arranged according to the syntactic rules of IMP.

Termination

Every statement must be terminated by a newline or a semicolon (however, see Comments).

Null Statements

There are two types of null statement, both of which are ignored by the compiler. They may be used to improve the legibility of the program.

1. Redundant terminators, E.g. blank lines

2. Comments
   A comment is either a statement starting with an exclamation mark (!) or the keyword comment, and terminated by a newline, or any sequence of characters (excluding } and newline) enclosed within a pair of curly brackets. The second form of comment may occur anywhere except between quotes.

```
e.g. comment Deal with exceptional cases
     ! beware of zero
     if X=0 {empty} or x=limit {full} start
```


APPENDIX 5

IMP KEYWORDS

alias	and	array			
begin	byte				
c	comment	const	constant	continue	cycle
dynamic					
else	end	event	exit	external	
false	file	finish	fn	for	format
frozen	function				
if	include	integer			
list	long				
map	monitor				
name	normal	not			
on	of	or	own		
predicate	program				
real	reals	record	repeat	result	return
routine					
short	signal	spec	start	stop	string
switch	system				
then	true				
unless	until				
while					

APPENDIX 4

DATA PRECISION SPECIFICATION

On some machines it is possible to offer a range of precisions for variables of type integer or real. The precision is specified by the use of one of the following prefixes:

short- smaller range than by default

long - larger range than by default

byte - large enough to hold a character (unsigned)

E.g.

byte integer	or	byte
short integer	or	short
long integer	or	long
long real		

If the machine on which the program is to be run cannot support the required precision the prefix will be ignored.

E.g. On the IBM 360 (or ICL 4/75)

byte integer	8-bits unsigned
short integer	16-bits signed
integer	32-bits signed
real	32-bits
long real	64-bits

Note that checks may be applied to ensure that any quantity assigned to a variable is within the correct range of values.

E.g.

```
shortinteger S
integer X
X=16_FFFF
S = X
```

will fail at run-time, as "16_FFFF" is a POSITIVE integer value but a NEGATIVE short integer value.

The assignment operator "<-" may be used to force truncation if required (see Assignment).

The statement reals long will cause all subsequent real keywords to be interpreted as long real. This effect may be terminated by the statement reals normal.

Instructions

The term instruction refers to an assignment, a Routine call or a control transfer.

Continuation

Any statement, excluding comments, may extend over several physical lines provided that each line-break occurs after a comma or a binary operator, or is preceeded by the keyword c. E.g.

```
if X = Y then P = 1 c
              else P = 0
```

which is exactly equivalent to:

```
if X = Y then P = 1 else P = 0
```

Notes

1. The line-break following c causes underlining to be terminated.
2. %C between quotes stands for the two characters percent and C.

Listing Control

During the compilation of a program a line-numbered listing is produced. The statements list and endoflist may be used respectively to enable or disable the listing for selected parts of a program. The default is for listing to be enabled.

Include

A file of statements (terminated by 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. E.g.

include "ECSC17.LISTVARS"

Refer to the relevant system manual for details of system-dependent limitations on the use of include.

Integer Constants (Fixed Point)

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

b) NON-DECIMAL constants
The prefix {decimal constant}"_" may be used to specify the base of the following constant. The letters a, b, .., z are used to represent the 'digits' 10, 11, ..., 35
E.g.

```
2_1010        - BINARY TEN
8_12          - OCTAL TEN
16_A          - HEXADECIMAL TEN
```

c) CHARACTER constants
The ASCII code value of any character may be obtained 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 respectively the code values for single quote, space, and newline.

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

d) MULTI-CHARACTER constants
The previous form may be extended to pack together the codes for several characters to form a single integer constant.

'OVER', 'Max', '1+2', '*@@#'

The exact nature of the packing and the maximum number of characters which may be so packed are both machine dependent.

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


```
   1 %begin
   2    %begin
   3       %realname Q
   4       %integer VALUE, X, X
*                            ! copy
   5       %string(256) S
*  size
   6       %switch SA(1:4), SB(5:4)
*  bounds
   7       %routine %spec CHECK
   8       %integer %functionspec KEY(%integer LOCK)
   9       %if X = 4 %stary
*                         ! atom
  10          VALUE = KEY
*                        ! form
  11          X = VALUW
*                    ! name
  12          X = X+1
  13  sa(5):
*         ! index
  14          VALUE = 0
  15       %finish
*  %start missing
  16       %exit %if X < 0
*  %cycle missing
  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 3

SAMPLE PROGRAM LISTINGS

Computer Science IMP77 Compiler. Version 6.00

```
 1  %begin;                   !program to paginate a file
 2     %constant %integer page size = 64;     !lines per page
 3     %constant %integer form feed = 12;     !FF character
 4
 5     %integer sym
 6     %integer lines left = page size
 7     %integer line number = 0
 8
 9     %on %event 9 %start;              !Input ended
10        newline
11        %stop
12     %finish
13
14     selectinput(1);  selectoutput(1)
15
16     %cycle
17        sym = nextsymbol;              !get input ended
18                                       !before printing
19        line number = line number+1
20        write(line number, 4);  space
21
22        %cycle
23
24           %if sym = form feed %start;        !newpage
25              lines left = 0
26              sym = nl
27           %finish
28           %exit %if sym = nl;                !end of line
29           printsymbol(sym)
30           skipsymbol
31           sym = nextsymbol
32        %repeat
33
34        skipsymbol;                           !skip newline
35        newline
36        lines left = lines left-1
37        %if lines left <= 0 %start;           !end of page
38           lines left = page size
39           printsymbol(form feed)
40        %finish
41     %repeat
42
43  %endofprogram

 36 statements compiled
```


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 @{signed integer constant} meaning "times ten to the power {signed integer constant}". For example, the following real constants all represent the same value (ignoring machine-dependent precision limitations):

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

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

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.

E.g. "STARTING TIME", "x = y*4+x", "a ""red"" hood"

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

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

EXPRESSIONS

1. Arithmetic Expressions

An arithmetic expression is a sequence of arithmetic operands and operators obeying the usual rules of arithmetic expressions. An operand is either a constant, a variable, a function call, a map call, or a numerical expression enclosed in parentheses (see Declarations and Procedures).

a) Integer Expressions

All the operands and operators in an integer expression must yield an integer value. Real values may be converted into integer values using the functions INT and INTPT (refer to the relevant library manual).
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 integer.

b) Real Expressions

All the operands and operators in a real expression must yield real (or integer) results. Where an operator will take either real or integer operands (E.g. +) and the types of the given operands differ, the integer operand will be converted to a real value, otherwise the result of the operation will be of the same type as the original operands. The pre-defined real function FLOAT may be used to force the conversion of an integer expression into a real expression.
The operators available are:

+	addition
-	subtraction or unary minus
*	multiplication
/	division
\	real exponentiation The second operand (the exponent) must be an integer operand.

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

APPENDIX 2

COMPILER MESSAGES

ERRORS

Any errors detected by the compiler will generate messages of the form: * {message}

In most cases a marker (¦) will be output to indicate the position in the statement at which the error was detected.

ATOM -	unknown atomic element. Usually a spelling mistake or % misused.
BOUNDS -	invalid bounds for an array or switch declaration, or wrong number of constants for an array initialization.
CONTEXT -	formally correct statement given in the wrong context. E.g. return inside a function.
COPY -	attempt to redefine a local identifier.
FORM -	incorrectly formed statement. Usually the addition or omission of an atom.
FORMAT -	use of a record with an undefined format.
INDEX -	switch label index out of bounds.
MATCH -	procedure definition does not match a previous spec.
NAME -	undeclared identifier
ORDER -	formally correct statement in wrong sequence. Usually declarations after an on statement.
PROTECTED -	attempt to modify (or unfreeze) a frozen variable.
SIZE -	constant out of range.
TOO COMPLEX -	statement too long or complex to analyse.
TYPE -	object of the wrong type.
%BEGIN MISSING -	too many end statements
%CYCLE MISSING -	a repeat with no matching cycle.
%END MISSING -	unterminated blocks remain at end of program or end of file.
%FINISH MISSING-	outstanding start at end or repeat.
%REPEAT MISSING-	outstanding cycle at end or finish.
RESULT MISSING -	a function, map, or predicate can reach its end.
%START MISSING -	a finish with no matching start.
"{id}" MISSING -	undefined procedure, label, or format.

WARNINGS

? {id} unused -	an identifier has been defined but not used.
? Non-local -	a for statement uses a non-local control variable.
? Access -	this statement can never be reached.
? } missing -	a short comment has not been terminated.

APPENDIX 1

A NOTE ON THE GRAMMAR

?	- indicates a rule is optional
*	- indicates zero or more instances of a rule
,	- separates alternatives
()	- define the scope of the above items
{}	- enclose phrase identifiers
""	- enclose literal strings (keywords are simply 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.
```


Notes

1. Unary minus is treated as 0-...
2. Unary plus (+) is not accepted.
3. An expression may not contain two adjacent operators - they must be separated by parentheses E.g. 23*(-14)
4. 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 pre-defined functions INT or INTPT.

2. 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 possible to mix integer and bit-vector expressions but the full implications of this may be machine dependent.

3. String Expressions

All operands of a string expression must yield values of type string. The only operator available is "." for concatenation (joining together). No sub-expressions in parentheses are permitted.

E.g. "MR ".SURNAME

Precedence of Operators

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

In general, sub-expressions with operators of equal precedence are evaluated from left to right.
The precedence rules may be over-ridden by means of parentheses.

```
Note:    -1\\2 = -1
         (-1)\\2 = 1
         2\\2\\3 = 4\\3 = 64
```

DECLARATIONS

All identifiers (except simple labels and record formats) must be declared at the start of a block before they are 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 four variants:

1. integer
2. real
3. string "(" {max} ")"
4. record "(" {format} ")"

and {max} is an integer constant in the range 1<=max<=255 defining the maximum number of characters which may be held in the string.

{format} 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. Variables

a) Simple Variables

{type}{idlist}

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

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

b) Pointer Variables

{type} name {idlist}

```
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 stated type.

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

signal event {n}{qual}?

where:

```
{n}       ::= an integer in the range 0 <= N <= 15
{qual}    ::= "," {sub event}{extra}?
{extra}   ::= "," {extra info}
```

and {sub event} and {extra info} are integer expressions.

The instruction causes event {n} to be signalled with sub-event (default zero) and extra information (default zero).

```
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 1. In both the on and signal statements the keyword event is optional and may be omitted.

2. 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 block which invoked the current block.

Three functions are provided to give information about the last event to have been signalled.

```
integerfnspec EVENT
integerfnspec SUB EVENT
integerfnspec EVENT INFO
```

If no event has been signalled each of these functions returns zero.

The classes of event and the sub-classes of them are:

EVENT	SUB-CLASS	MEANING (+EXTRA INFORMATION)
0		TERMINATION
	-1	abandon program
	0	normal termination (stop)
	>0	user generated error
1		ARITHMETIC OVERFLOW
	1	integer overflow
	2	real overflow
	3	string overflow
	4	division by zero
2		EXCESS RESOURCE
	1	not enough store
	2	output exceeded
3		DATA ERROR
	1	symbol in data (+symbol)
4		CORRUPT DATA
	1	data transmission error
5		INVALID ARGUMENTS
	1	for cannot terminate
	2	illegal exponent (+exponent)
	3	array inside-out
	4	string inside-out
	5	illegal parameter for READ
6		OUT OF RANGE
	2	array bound fault (+index)
	3	switch bound fault (+index)
	4	Illegal event signal
7		RESOLUTION FAILS
8		UNDEFINED VALUE
	1	unassigned variable
	2	no switch label (+index)
9		STREAM ERROR
	1	input ended
	2	illegal stream (+stream no)
10		LIBRARY PROCEDURE ERROR
11 - 15		GENERAL PURPOSE

Note that some of the events may not be signalled automatically in certain implementations or when the program has been compiled without checks. Refer to the relevant implementation notes for details.

c) Array Pointer Variables

{type} array name {idlist}
{type} name array name {idlist}

```
integer array name AN
real array name VALUES
string (20) array name NAMES, ADDRESSES
record (CARFM) array name MAKE
integer name array name POINTERS
```

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

{type} array "(" {dim} ")" name {idlist}

is provided for declaring pointers to multi-dimensional arrays. E.g.

```
real array (4) name SPACE TIME
```

SPACE TIME may now be pointed at an array of dimension 4.

2. Arrays

{type} array {adefn}("," {adefn})*
{type} name array {adefn} ("," {adefn})*

{adefn} ::= {idlist} "(" {pair}(","{pair})* ")"
{pair} ::= {integer exprn} ":" {integer exprn}

```
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 POINTERS('A':'Z')
```

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

note 1. In each bound pair the value of the first expression (lower bound) must be less than or equal to the value of the second expression (upper bound).

2. 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 variables, arrays and records. The components (elements) of a record may be any of the forms discussed in (1) and (2) above, with the following limitations:

a. Arrays within records must be one dimensional and have constant bounds.
b. A record may not contain a simple record (or a record array) 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.

record format {id} "(" {declaration list} ")"

```
record format F(integer X, record(F)name LINK)
record (F) HEAD
record (F) array CELL(1:15)
```

Note 1. Within a format each identifier must be unique but will not clash with any identifiers outwith that format.

2. 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 appendix for machine dependent alignment considerations.

A format may be defined as an ordered list of formats:

```
recordformat A(............)
recordformat B(............)
recordformat C(............)

recordformat D(A or B or C)
```

Records defined using such a format may be qualified by selectors from any of the component formats (see Record element selection). The amount of storage required for such records is the amount required for a record of the largest format in the list.

The formats of two records are considered equivalent if they contain a common alternative.

Within record format definitions, record pointer variables may be declared using format identifiers that have not yet been defined. The format must be defined before the record is used.

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


EVENTS

During the execution of a program several (synchronous) events may occur, such as failure of resolution, array bound fault etc. (see Faults). 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 representing the events 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 the program terminates.

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

INPUT/OUTPUT

```
routine READSYMBOL(name S)
routine SKIPSYMBOL
integer function NEXTSYMBOL
routine READ(name N)

routine PRINTSYMBOL(integer N)
routine PRINTSTRING(string(*) S)
routine WRITE(integer N, PLACES)
routine NEWLINE
routine NEWLINES(integer N)
routine SPACE
routine SPACES(integer N)

routine SELECTINPUT(integer STREAM)
routine SELECTOUTPUT(integer STREAM)
```

STRING HANDLING

```
string(1) function TOSTRING(integer SYMBOL)
string(*) fn SUBSTRING(string(*)name S, integer F,T)
integer function CHARNO(string(255) S, integer N)
integer function LENGTH(string(255) S)
```

EVENT HANDLING (see Events)

```
integer function EVENT
integer function SUB EVENT
integer function EVENT INFO
```

STORE MAPPING

```
integer function ADDR(name V)
integer map INTEGER(integer ADDRESS)
real map REAL(integer ADDRESS)
string(*)map STRING(integer ADDRESS)
record(*)map RECORD(integer ADDRESS)
```

Refer to the relevant system library manual for detailed specifications of these and other standard procedures.

RECORD ELEMENT SELECTION

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

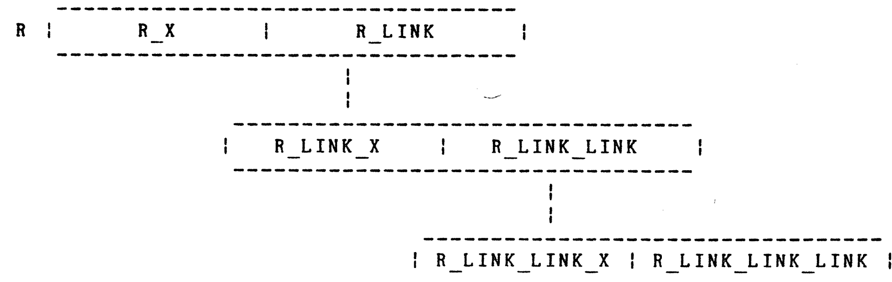
"_"{element id}

E.g. given the declarations:

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

some valid references to variables are:

```
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
```



If the record had been defined using a format declared to be a list of formats, the search for the element identifier would consider each of the formats in the list in the order specified. This means that if two of the formats in the list contain a common element identifier the first will be used. The identifiers of the formats comprising the list may be used as element identifiers to select a particular format as long as the format identifiers do not clash with any of the component element identifiers. e.g.

```
recordformat X(integer Q, R)
recordformat Y(real P, Q)
recordformat Z(X or Y)

record(Z) W

W_Q = 0;        !an integer
W_P = 0;        !a real
W_Y_Q = 1.2;    !a real
```

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.

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 (see Assignment).

```
integer A,B=4, C=-1-B
! the initial value in A is undefined
real R=1.234@-5
string (7) WHO="ANON"
integer name POINT == 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 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.

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. externalrealfnspec SIN alias "MATH$DSIN" (real ARG)

PREDEFINED PROCEDURES

Every separately compiled module, whether a begin-end of program block or a file of external procedures is compiled within a conceptual "outermost block" in which are declared a number of standard procedures such as READ and WRITE. This means that these procedures are global to all parts of a program and so may be used without having to be declared. Note that as these procedures are global they may be redefined within the program.

There are no restrictions on the use of the pre-defined procedures other than those naturally imposed by their definition (routine, function etc.). In particular, predefined procedures may be passed as procedure parameters. Further, own, constant or external identifiers may be declared in this outermost block and will be global to the whole of the file.

```
own integer CALLS = 0

external routine DO SOMETHING
   CALLS = CALLS+1;          ! RECORD TIMES ENTERED
   .
   .
end

external integer function ENTRIES
   result = CALLS
end

end of file
```

Note that the function ENTRIES is used to make the value in CALLS available to other modules without their being able to change that value, even by mistake, other than by calling DO SOMETHING.

For example the following is a complete file of external procedures:

```
integer function SHIFT(integer SYM)
   result = SYM-'a'+'A';    !  LOWER - UPPER CASE
end

external predicate LETTER(integer SYM)
   true if 'A' <= SYM <= 'Z'
   true if 'A' <= SHIFT(SYM) <= 'Z'
   false
end

external predicate DIGIT(integer SYM)
   true if '0' <= SYM <= '9'
   false
end

external predicate ALPHANUM(INTEGER SYM)
   true if LETTER(SYM) or DIGIT(SYM)
   false
end

end of file
```

Note a. The function SHIFT is local to the file - it cannot be called from a different module.

b. The normal scope rules apply within the file, so that ALPHANUM may call both LETTER and DIGIT.

c. External procedures 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.

However, a spec for a procedure contained in the same module as the spec statement must not have the external attribute (because the procedure is not external to the module).

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


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

Any number of null statements may be placed between the lines of constants.

```
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_5D00,
!     A         S         M         D
   16_1A00,  16_1B00,  16_1C00,  16_1D00,
!     AR        SR        MR        DR
   -1(*);                                      !all the rest
```

FROZEN VARIABLES

Variables may be created to contain initial values which, once given, may not be changed. Both values and references may be so frozen. Note that the frozen attribute does not affect the existence of objects; they are created and destroyed in the same way as their non-frozen equivalents.
For example:

```
frozen integer       FI  = X+1
frozen integer name FIN == Y
integer frozen name IFN == Z
```

FI is given the value X+1 and subsequently may not be altered.
FIN may reference any integer variable (initially Y) but the variable it references may not be altered via FIN, i.e. FIN==Z is valid but FIN=0 is not.
IFN is initialised to reference Z and may never reference another integer, however the integer it references may be changed, i.e. IFN==X is not valid but IFN=0 is.

CONSTANT IDENTIFIERS

Named constants may be declared using the prefix constant. A constant integer may be used wherever an integer constant is required.

```
constant integer MAX = 17
constant real PI = 3.14159
constant string (7) VERSION = "vsn:1.6"
constant integer array VAL(1:MAX) = 1,6,9,-1(*)
```

The keyword constant may be abbreviated to const.

constant ..... arrays are effectively own frozen ... arrays.

ASSIGNMENT

There are three forms of assignment:

1. {variable} "=" {expression}

```
X = Y
A(P) = A(P)+1
Y = BIT<<12
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 the corresponding type; in the case of a real variable an integer expression will have its result converted to real before being assigned.

Valid types of assignment are:
{integer variable} "=" {integer expression}
{real variable} "=" {real expression}
{real variable} "=" {integer expression}
{string variable} "=" {string expression}

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} "==" {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 an alternative to the variable.

```
integer name N
integer J
integer array A(1:6)
J = 1
N == A(J);                ! N IS NOW EQUIVALENT TO A(1)
J = 2;                    ! N HAS NOT CHANGED
N = 0;                    ! SAME AS A(1) = 0
```


EXTERNAL LINKAGE

A complete program may be divided into several separately compiled modules which are linked together before (or possibly while) the program is executed. This linkage is achieved by giving the external attribute to relevant indentifiers.
The keyword external may be replaced by system or dynamic but the effect of this is implementation dependent.

1. external DATA OBJECTS

An external variable is declared in the same way as an own variable 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 1. No initialization may be given in an external specification.

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

3. Even though all of the characters in the identifier of an external entity are significant to the compiler, the system loader software might impose constraints on the number of significant characters. Refer to the relevant appendix 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)
```

Such procedures must be compiled in a file comprising only external procedures (and possibly some non-external procedures and own or external declarations). The whole module is terminated by the statement end of file.

OTHER CONTROL TRANSFER INSTRUCTIONS

stop

Execution of the instruction stop causes control to be returned to the program which initiated the execution of the current program. This is also the effect of reaching the statement end of program.

monitor

This instruction causes the run-time diagnostic package to be invoked to produce diagnostic information. If no diagnostic package is available this instruction will be ignored (in some limited implementations the production of diagnostics causes execution of the program to be terminated).

For convenience all other control transfer instructions are gathered here.

return	return from a routine.
result={exp}	return the result of a function
result=={reference}	return the result of a map.
true	return from a predicate.
false	return from a predicate.
exit	jump out of the current cycle to the statement following the matching repeat.
continue	jump to the repeat of the current loop.
signal event	see Events.

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 extra assignments exist for records.

1. {record variable} "=" {record variable}

   The right-hand record is copied bit by bit into the left-hand record. The formats of the two records must be equivalent.

2. {record variable} "=0"

   Each bit of the record is set to zero.

STRING RESOLUTION

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

{string var}"->"{string var}".("{string exp}").\"{string var}

where either the second string variable, the third, or both may be omitted (any dangling full stops also being omitted).

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

The string expression, C, is evaluated and the first variable, A, is searched from left to right to find that string of characters. The fragment of the string to the left of the sub-string so found is assigned to the second variable, B, and the fragment to the right is assigned to the third string variable, D.
The resolution is deemed to have failed if the required sub-string is not found or either of the second or third string variables has been omitted and would have been assigned a non-null string. No assignments are performed if the resolution fails.

For example, the following resolutions all fail if the string variable S contains the string "ABCDEFG"

```
S -> T.("H").U
S -> ("CD").U
S -> T.("EF")
S -> ("ABCDEF")
```

and the following all succeed:

```
S -> T.("CDE").U
S -> ("ABC").U
S -> T.("G")
S -> ("ABCDEFG")
```

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)

   ```
   S -> A.(WANTED).B;  S = A.B
   ```

2. as a simple condition (see Conditions), in which case failure of the resolution deems the simple condition false and success deems it true; in the latter case the resolution is performed and the necessary assignments are made.

   <u>if</u> WHO -> ("Sir ").WHO <u>then</u> KNIGHT = 1

<u>switch</u> LET('A':'Z')
.
.
LET('A'):LET('E'):LET('I'):LET('O'):LET('U'):

! DEAL WITH VOWELS
.
.

LET(*):! ALL THE REST I.E. CONSONANTS

The specific label to which a jump will be made is dependent on the value of an integer expression.

->SW(N) <u>if</u> N > 0
->SW(100+N)
->SW(6)

Note 1. Not all of the declared switch labels need be defined (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.

2. Simple labels may be used before they are defined.

   -> MISSING <u>if</u> HERE = 0
   .
   .
   MISSING:

3. The scope of both types of label is limited to the block in which they are defined, not including any blocks defined therein. Therefore it is not possible to jump into or out of a block.

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

5. The results of entering a <u>for</u> loop with a jump and not through the <u>for</u> statement are undefined.

CONTROL TRANSFER INSTRUCTIONS

LABELS and JUMPS

1. **Simple Labels**

Any statement, excluding declarations, may be given one or more simple labels, where a simple label is of the form: {id} ":"

Each label is written to the left of the statement.

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

Control may be passed to a labelled statement by executing a jump instruction: "->" {id}

```
-> NEXT

-> ERROR1 if DIVISOR = 0
```

2. **Switch Vectors**

A vector of labels may be declared in a similar manner to an array, using the declarator switch.

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

Note a. The vector must be one dimensional.

b. The bounds must be constants.

Once declared, switch labels may be defined 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
```

A star (*) may be used in the definition of a switch label to define any elements within the vector which would otherwise be undefined.

CONDITIONS

Conditional statements are specified using the phrase {condition}, which is defined as:

```
{condition} ::= {simple cond} (and {simple cond})*,
                {simple cond} (or {simple cond})*
```

where {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 true or false depending on the validity of the relation specified by the comparator. Both expressions must yield values of the same type.

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 is only evaluated if the condition specified by the first two expressions is true.
Such a simple condition is frequently used to check for a range of values, E.g.

0 <= VALUE <= 100

3. {reference to variable} "==" {reference to variable}
{reference to variable} "##" {reference to variable}

The two references, which must be of the same 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.

4. {predicate call} - see Procedures

The given predicate is called and the simple condition is true or false depending on whether the exit from the predicate was performed using true or false respectively.

5. {resolution} - see String Resolution

The resolution is attempted. If it fails the simple condition is deemed false, otherwise the resolution is performed and the condition is deemed true.

6. "(" {condition} ")"

This form of simple condition is provided to enable the use of both <u>and</u> and <u>or</u> in a condition, as these connectives are considered to have equal precedence. The connectives <u>and</u> and <u>or</u> may not appear in the same condition except at different levels of parenthesis. E.g.

A=0 <u>or</u> (B=1 <u>and</u> C=2) <u>or</u> D=3

7. <u>not</u> {simple cond}

The given simple condition is evaluated and its truth is negated. E.g. the following simple conditions are exactly equivalent:

A # 0
<u>not</u> A = 0

Evaluation of conditions

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

For example, consider the condition:

A # 0 <u>and</u> B//A # C

If the variable A has the value zero the condition will be deemed false without attempting the evaluation of "B//A # C".

PROCEDURE SPECIFICATION

In several situations it is 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 procedure identifiers must be declared before being used a procedure specification statement is introduced.
This takes the form of the normal procedure heading with the keyword <u>spec</u> inserted before the procedure identifier.

E.g. <u>routine</u> <u>spec</u> MAX(<u>real</u> SIZE)

This has no effect other than declaring the identifier to be a procedure of the specified type which takes the given parameters. Except in the case of <u>external</u> procedure specifications the procedure must be defined later on in the block to which the <u>spec</u> 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 <u>spec</u> 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.

The following is a complete list of formal parameter declarators:

integer	real	string({max})
integer name	real name	string({max})name
integer array name	real array name	string({max})array name
integer fn	real fn	string({max})fn
integer function	real function	string({max})function
integer map	real map	string({max})map

record({fm})

record({fm})name

record({fm})array name

record({fm})fn

record({fm})function

record({fm})map

routine

predicate

name

integer name array name

real name array name

string({max}) name array name

record({fm}) name array name

CONDITIONAL GROUPS

(see Block Structure)

The general form of conditional statements is:

```
if {condition 1} start
   !Statements to be executed if
   !{condition 1} is true.
finish else if {condition 2} start
   !Statements to be executed if {condition 1} is false
   !and {condition 2} is true.
finish else if {condition 3} start
   ..........
   ..........
finish else start
   !statements to be executed if all the previous
   !conditions are false.
finish
```

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

ALTERNATIVE FORMS

1. If the start-finish brackets enclose only one instruction the start-finish group may be replaced by:

   ```
   if {condition} then {instruction} .........      or
   ........... else {instruction}
   ```

2. 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 Y = -1
   unless X = 0 then Y = -1 else Y =  1
   ```

3. In a statement of the form: "finish .... start" both the finish and the start may be omitted e.g.

   ```
   if A = 0 start
      FLAG = 1
   else if A = 12
      FLAG = 3
   else if A < -4
      FLAG = 0
   else
      FLAG = -1
   finish
   ```

4. 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`

REPETITION (LOOPS OR CYCLES)
(see Block Structure)

a. 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.

b. Conditional Repetition

1. while {condition} cycle

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

2. 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; thus 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.

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 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 P(routine Q(integer R)))
```

and the call would be:

```
CHECK(TRY)
```

GENERAL TYPE PARAMETER

In several situations it is useful to be able to pass to a procedure a reference to any type of variable. This is done by specifying an untyped name parameter.

```
E.g. routine WORK(name REF)
```

Such a parameter is intended for system-dependent interface procedures and has severely limited uses. In particular it may only be passed on to another procedure requiring an untyped name parameter.

An example of the use of such a parameter is in the pre-declared READ routine which will accept an integer, real, or string parameter.

```
E.g. integer X
     real Y
     string (15) Z
     READ(X);  READ(Y);  READ(Z)
```

PARAMETERS

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

{param def} ::= "(" {dec list} ")"

where {dec 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, Q)
     integer function MAX(integer array name A, integer F, T)
     predicate EQUIV(record(FM)name LEFT, RIGHT)
```

Parameters are identical to any local variables declared inside the procedure, except that the parameters are initialised each time the procedure is called.
When a procedure is called a list of ACTUAL PARAMETERS must be supplied which must 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. IMP name type parameters are called by reference and not by substitution (c.f. ALGOL 60).

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}.

Note The execution of the cycle body must not alter the value of the control variable - the results of doing so are indeterminate.

3. The final form of conditional cycle is:

```
cycle
   ! CYCLE BODY
repeat until {condition}
```

In this construction the cycle body is executed at least once, terminating when the condition becomes true.

cycle-repeat groups may be nested to any depth.

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 = ' '
```

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.

2. continue - causes control to be passed to the repeat of the current loop.

JOINING INSTRUCTIONS USING and

Several simple instructions may be joined together using and to form a more complex instruction. The execution of such an instruction is achieved by executing each of the component simple instructions in the order given. This construction is used to simplify small start-finish or cycle-repeat groups.

```
E.g. if X = 0 start
        P = 1; Q = 1
     finish
```

may be rewritten:

```
P = 1 and Q = 1 if X = 0
```

or:

```
if X = 0 then P = 1 and Q = 1
```


4. predicate {id}{param def}?

   A predicate is a procedure which tests the validity of an hypothesis and then returns, being either true or false. Predicates may be used wherever a simple condition is required.
   When a predicate is called its statements are executed until either of the instructions true or false is executed. This causes the predicate to terminate accordingly.

   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

a. A routine may terminate by reaching end; all other types of procedure must not be able to reach end, otherwise the compiler will report a fault.

b. Procedure definitions may be nested within any form of block.

c. Procedures may be recursive, that is, a procedure definition may contain a reference to itself.

d. 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.).

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 as the value of the function.

```
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 "==" {variable reference}
```

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
```


BLOCK STRUCTURE

An IMP program is constructed using one or more blocks. Blocks 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.

BEGIN BLOCKS

The simplest type of block is enclosed between the statements begin and end and is referred to as a begin block. If the block is the outermost block of a complete program it must be terminated by the statement
end of program, rather than by a simple end.

For example, a complete program might take the form:

```
begin
   integer COUNT, LIMIT
   .
   begin
      real SUM
      .
   end
   .
end of program
```

A begin block is entered by executing the begin and is left by passing through the end to the following statement. 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.

```
begin
   integer UPPER
   UPPER = ...;     !  CALCULATE VALUE FOR UPPER BOUND
   begin
      integer array CASES(1:UPPER)
      .
   end
end of program

begin
   .
   begin
      integer array TEMP(1:10000)
      .
   end
   begin
      real array WORK AREA(1:11000)
      .
   end
end of program
```

LOCAL AND GLOBAL VARIABLES

An identifier is described as being local to a block if it was declared at the head of 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;                   ! end of inner block
end;                      ! 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
```

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

On entry to a block, storage is allocated to local variables and when the block is left the storage is deallocated (but see Own Variables).

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 end). 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, the previous flow of control is resumed.

There are four forms of procedure, the exact form required being specified by the first statement 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}?

   When a routine is called its statements are executed until either the end 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
end
..
..
CONVERT
..
CONVERT unless X = 0
```