THE IMP-77 LANGUAGE

As implemented by

PETER S. ROBERTSON
DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF EDINBURGH

A REFERENCE MANUAL

first edition: DECEMBER 1977
this edition reproduced: FEBRUARY 2003

INTRODUCTION

IMP 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.

The main disadvantage of IMP is that it is not currently in widespread use.

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.

INDEX

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 all subsequent 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).

Spaces

Except when used to terminate keywords or when between quotes (see Quotes) 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 (see Quotes) 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:

' - symbol 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 eleven 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 case of simple labels (see Control Transfer Instructions) 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 any sequence of characters (the "comment text") preceded by a comment mark and ending with a newline character (note: a semicolon does not terminate a comment; it is included as part of the comment text).

   The comment marks are either the keyword comment or an exclamation mark (!); e.g.

```
comment main loop starts here
! return here on error
```

   As a semicolon does not terminate a comment it is a simple matter to 'comment out' sections of code.

```
COMPUTE CASES
! if CASES < 0 start
!     DISPLAY DATA; NEWLINE; TIDY UP
!     stop
!finish
HANDLE CASES
```

   (Also see note on continuation)

Instructions

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

Continuation

A statement may extend over several physical lines provided that each line break occurs after a comma, or is preceded by the keyword c. E.g.

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

is exactly equivalent to:

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

Notes

1. A statement may have an unlimited number of continuations.
2. The line break following c causes underlining to be terminated.
3. c is only permitted between complete atoms of the language; that is, it may not split keywords, constants, etc.
4. %C between quotes stands for the two characters percent and C.
5. The effect of a comment mark is limited to one physical line (see Null statements).

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 "end of file") may be compiled into a program by giving the statement

```
"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 appendix for details of system-dependent limitations on the use of include.

CONSTANTS

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 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 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', 'l+2', '*@@£'
```

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

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 "@"{signed integer constant} meaning "times ten to the power {signed integer constant}". For example, 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.

String Constants

A string constant is a sequence of not more than 255 character 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+z", "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

Arithmetic Expressions

An arithmetic expression is a sequence of operators and integer or real operands obeying the usual rules of algebra. 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.

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 modulus or absolute value of an expression (integer or real) may be obtained by enclosing that expression between vertical bars.

E.g.

```
|X-Y|
```

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.

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.

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) 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.

{fmt} defines the structure of the record (see Records).

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 appropriate 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 specified type.

c) Array Pointer Variables

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

or

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

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

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

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

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

```
real array (4) name SPACE TIME
integer name array (2) name LISTS
```

Arrays

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

or

{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(1:12)
```

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.

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 must be one-dimensional and have constant bounds.

b) A record may not contain a simple record (or a record arrays) of its own format. However it may contain record pointer variables of its own format.

There are three ways to specify formats:

1. Explicit definition

record "(" {declaration list} ") ...

```
record (integer X, Y, Z) R
record (real P, real name Q) name S, T
record (real array A(1:5), real V) array X(1:4)
```

2. Using a format identifier

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

record "(" {id} ")" ...

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

3. Using a previously declared record as a format definition.

```
record (integer ONE, TWO, THREE) R
record (like R) S, T
```

Note

1. Within a format each identifier must be unique but will not clash with any identifiers outwith that format (see Block Structure for a discussion of local and global identifiers).

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 implementation notes for machine-dependent alignment considerations.

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

OWN VARIABLES

Each variable declared in a block is allocated storage when that block is entered, the storage being returned (released) when the block is left. This means that 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 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

Own variables may be given initial values (effectively before the program starts execution); if no initial value is specified the content of an own variable is undefined.

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

In the case of own name and own array name variables the initial value (if present) represents the absolute address of respectively the initial variable to be pointed at or the (possibly hypothetical) 'zeroth element' of an array.

```
own integer name CLOCK==72
own integer array name SAVE AREA == 16
```

This is highly machine dependent.

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.

```
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 VALUE(1:50) =;   ! TAG VALUES
    1, 2, 3, 0(7),
! ------type 1------
   11, 22, 33, 4(3), 55(4),
!------type 2------------
   111, 222, 3, -1(5),
! ------types 3 & 4-----------
    -2(*);              ! ALL THE REST
```

CONSTANT IDENTIFIERS

The prefix constant may replace own to indicate that the initial value can never change. 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(*)
```

Constant pointer variables may be declared but are highly machine and system dependent.

```
constant integer name STATUS REG == 160
constant integer array name WORD == 0
```

Note: constant pointers are effectively simple variables of the appropriate type located at the specified (absolute) address.

The keyword constant may be abbreviated to const.

ALIAS

In many programs it is convenient to be able to interpret a variable in several different ways. This may be achieved by declaring a new variable to be an alias of an existing variable. Any identifier in the identifier list of a declaration may be followed by "alias" {var} where {var} is a variable of any type. The effect is that the new identifier is allocated storage at the same address in memory as the variable of which it is an alias.

```
real VALUE
integer REAL BIT PATTERN alias VALUE

record (F1) name N1
record (F2) name N2 alias N1

integer X, Y, Z
integer array A alias X (1:3)
```

Note:

```
integer name N
integer X alias N
integer Y alias INTEGER(ADDR(N))
integer J
J = X;  ! same as J = ADDR(N)
J = Y;  ! same as J = N
```

An array can be mapped onto an absolutely addressed area of store by means of the built-in map INTEGER (see Predefined Procedures)

```
integer array SEGMENT alias INTEGER(SEGAD) (1:LIMIT)
```

Note that this feature is highly machine dependent.

[ABD note: as far as I understand it, much of the preceding is simply not true for real IMP-77 compilers. The real use of alias is in renaming external routines]

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 the assignment.

Valid types of assignment are:

{integer variable} "=" {integer expression}
{real variable} "=" {real expression}
{real variable} "=" {integer expression}
{string variable} "=" {string expression}

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.

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)
integer name array PT(2:12)
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
PT(J) == A(4)
```

N.B. Extreme care should be taken if variables declared in different blocks are to be equivalenced as it is possible to leave a pointer referencing a variable which no longer exists (see Block Structure).

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 CAPACITY EXCEEDED 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 the same.

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.

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

The string expression is evaluated and the first variable is searched from left to right to find that string of characters. The string to the left of the sub-string so found is assigned to the second variable and the string to the right is assigned to the third.

The resolution is deemed to have failed if the required sub-string is not found or either of the second or third variables has been omitted and would have been assigned a non-null string.

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 deems the simple condition false and success deems it true; in the latter case the resolution is performed and the necessary assignments are made.

```
if WHO -> ("SIR ").WHO then KNIGHT = 1
```

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 not evaluated unless 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 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.

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.

Note that this form of simple condition has a side effect if the simple condition is true!

6. "(" [condition] ")"

This form of simple condition is provided to enable the use of both and and or in a condition. 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}

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

```
A # 0
not 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, considering the condition:

```
A # 0 and 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".

CONDITIONAL STATEMENTS

The general form of a conditional statement is:

```
if {condition} start
   ! STATEMENTS TO BE EXECUTED IF
   ! {condition} IS TRUE
finish else start
   ! STATEMENTS TO BE EXECUTED IF
   ! {condition} IS FALSE
finish
```

If start-finish brackets enclose one instruction only, that part may be reduced to:

```
if {condition} then {instruction} else start
```

or

```
finish else {instruction}
```

or in the simplest case:

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

If nothing is to be done specifically when the condition is false the else part may be omitted.

```
if {condition} start
   ! STATEMENTS TO BE EXECUTED IF
   ! {condition} IS TRUE
finish
```

or

```
if {condition} then {instruction}
```

start-finish groups may be nested to any depth.

ALTERNATIVE FORMS

1. A conditional statement of the form:

```
if {condition} then {instruction}
```

has the same effect if rewritten in the more natural form:

```
{instruction} if {condition}
```

```
X = ERROR if X > LIMIT
```

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. The statement "finish else start" may be abbreviated to "else".

```
if X = 0 start
   FLAG = 1; COUNT = 0
else
   FLAG = 2; COUNT = -1
finish
```

4. The else part of any conditional group may be replaced by another complete conditional group, treated as though it were a single instruction.

For example:

```
if A = 0 start
   P = 1; Q = 2
finish else start
   if A < 0 start
      P = -1; Q = 2
   finish else start
      P = 1; Q = -2
   finish
finish
```

may be rewritten:

```
if A = 0 start
   P = 1; Q = 2
finish else if A < 0 start
   P = -1; Q = 2
finish else start
   P = 1; Q = -2
finish
```

REPETITION (LOOPS OR CYCLES)

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 value of the three expression are evaluated and saved; thus the cycle body cannot change them. The control variable is assigned the value "{init}-{inc}".

The value in the control variable is compared with the value of {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.

On normal exit from the cycle the control variable will contain the value {final}, however see exit.

Note: the effects of altering the control variable within the cycle body are undefined.

3. The final form of conditional cycle is:

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

In this construction, the cycle body is always executed at least once. The loop may also be qualified by a while or for as defined above. For example:

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

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. In the case of a for loop the control variable will retain the value it contained immediately prior to the exit.
2. continue - causes control to be passed to the repeat (and any associated until condition) 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
```

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.

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".

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-tine, 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, space from the stack is allocated to any local variables, and when the block is left the space is returned to the stack (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 t h e 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
```

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 an instruction of the form:

```
result "=" {expression}
```

is executed. 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 an instruction of the form:

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

is executed. 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 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 their end, otherwise the compiler will report a fault.

b. Procedures may be nested within any form of block.

c. Procedures may be recursive, that is, they may call themselves.

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).

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 PARAMETERS

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 Y
real Y
string (15) Z
READ(X);  READ(Y);  READ(Z)
```

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

integer name array name

real name array name

string({max})name array name

record({fm})

record({fm})name

record({fm})array name

record({fm})name array name

record({fm})fn

record({fm})function

record({fm})map

routine

predicate

name

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 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 type which takes the given parameters. Except in the case of external procedure specifications the procedure must be defined later on in the same block (but not any blocks defined therein).

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.

CONTROL TRANSFER INSTRUCTIONS

LABELS AND JUMPS

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
```

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 used in the same way as simple labels.

```
SW(4):          CHECK VALUE(1)
SW(5):SW(6):    ERROR FLAG = 1
LAST: SW(9):    ! ALL FINISHED
```

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

```
switch 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) if N > 0
->SW(100+N)
->SW(6)
```

Note:

1. Not all of the declared switch labels need be defined (in the previous example SW(7) 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 are the only identifiers which may be used before they are declared/defined.

3. The scope of both types of label is limited to the block in which they are defined, not including 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.
4. The identifiers used for labels must not conflict with other local identifiers.
5. The results of entering a for loop with a jump and not through the for statement are undefined.

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.

Control is transferred by signalling event zero (see Events).

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 top of the current cycle.
signal event	see Events.

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 identifiers.

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 initialisation may be specified 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 load 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

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.

The prefix external may be replaced by system or dynamic, the exact significance of which may vary from machine to machine.

PREDEFINED PROCEDURES

Every separately compiled module, whether a begin-end 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.

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 of CALLS available to other modules without their being able to change that value, even by mistake.

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

routine PRINTSYMBOL(integer N)
routine PRINTSTRING(string(255) 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(255) fn SUBSTRING(string(255)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(255)map STRING(integer ADDRESS)
```

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

EVENTS

During the execution of a program several (synchronous) events may occur, such as resolution fails, array bound faults 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 further 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 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.

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

4. integer function EVENT - returns the class of the last event.
5. integer function SUB EVENT - returns the sub-class of the last event.
6. integer function EVENT INFO - returns any extra information passed with the event.

If no event has occurred each of these functions will return the value zero.

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

EVENT	SUB-CLASS	MEANING (+EXTRA INFORMATION)
0		TERMINATION
	-1	ABANDON PROGRAM WITHOUT DIAGNOSTICS
	0	NORMAL TERMINATION (stop)
	>0	USER GENERATED ERROR
1		ARITHMETIC OVERFLOW
	1	INTEGER OVERFLOW
	2	REAL OVERFLOW
2		EXCESS RESOURCE
	1	NOT ENOUGH STORE
3		DATA ERROR
	1	SYMBOL IN DATA (+SYMBOL)
4		CORRUPT DATA
	1	DATA TRANSMISSION ERROR
5		INVALID ARGUMENTS
	1	ILLEGAL CYCLE
	2	ILLEGAL EXPONENT (+EXPONENT)
	3	ARRAY INSIDE-OUT
6		OUT OF RANGE
	1	CAPACITY EXCEEDED
	2	ARRAY BOUND FAULT (+INDEX)
	3	NO SWITCH LABEL (+INDEX)
7		RESOLUTION FAILS
8		UNASSIGNED VARIABLE
9		INPUT ENDED
10		LIBRARY PROCEDURE ERROR
11-15		GENERAL PURPOSE

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

```
signal event {N}{QUAL}?
```

where:

```
{N}       ::= an integer constant 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 in the previous block.

APPENDIX 1

COMPILER ERROR MESSAGES

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.

ACCESS	-	the statement cannot be reached. This is not treated as an error but may indicate another fault.
ATOM	-	unknown atomic element.
BOUNDS	-	invalid bounds for an array or switch declaration, or wrong number of constants for an array initialisation.
CONTEXT	-	formally correct statement given in the wrong context.
COPY	-	attempt to redefine a local identifier.
FORM	-	incorrectly formed statement.
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.
SIZE	-	constant out of range.
TOO COMPLEX	-	statement too long or complex to analyse.
TYPE	-	variable of wrong type.
TYPE FOR {op}	-	operator {op} out of context.
%BEGIN MISSING	-	too many end statements
%CYCLE MISSING	-	a repeat has been given 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 has been given with no matching start.
"{ID}" MISSING	-	undefined procedure or label.

APPENDIX 2

SAMPLE PROGRAM LISTING

```
   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           %routinespec CHECK
   8           %integerfnspec KEY(%integer X)
   9           %if X = 4 %STARY
*                                ! ATOM
  10                 VALUE = KEY
*                                ! FORM
  11                 VALUR = 0
*                    ! NAME
  12     SA(5):
*              ! INDEX
  13                 %exit
*%CYCLE MISSING
  14                 %stop
  15                 X = 0
*ACCESS
  16           %finish
*%START MISSING
  17           %on %event 4 %start
*ORDER
  18                 %integerfunction KEY(%real X)
*MATCH
  19                 %end
*RESULT MISSING
  20           Q == VALUE
*                   ! TYPE
  21           X = Q&7
* TYPE FOR "&"
  22     %endofprogram
*%END MISSING
*%FINISH MISSING
*"CHECK" MISSING
```

APPENDIX 3

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
short integer
long integer
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).

APPENDIX 4

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
if	include	integer			
like	list	long			
map	monitor				
name	not				
on	of	or	own		
predicate	program				
real	record	repeat	result	return	routine
short	signal	spec	start	stop	string
switch	system				
then	true				
unless	until				
while					

APPENDIX 5

COMPARISON WITH EMAS IMP

1. New Features

   for

   repeat until

   continue

   predicate

   include

   "==" in conditions

   integer array (4) name

   finish else if ...

   else

   lower case input

   like

   (*) in owns and switches

   constant

   function

   alias

   not

   record function

   record map

   constant expressions

2. Features not implemented

   print text

   until ... cycle

   array format

   reals long

   reals normal

   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 operation

external .. spec instead of extri

events instead of fault trapping

/ gives a real result

string resolution