Edinburgh
Regional
Computing
Centre

A Syntactic and Semantic definition of the IMP Language

As implemented by the
Edinburgh Regional Computing Centre

by
P.D. Stephens

First edition
August 1974

A SYNTACTIC AND SEMANTIC

DEFINITION OF THE IMP LANGUAGE

AS IMPLEMENTED IN THE ERCC.

PREFACE

This reference manual describes the IMP language as implemented by the Edinburgh Regional Computing Centre on its service computers, a twin ICL 4-75 interactive system operating under the Edinburgh Multi-Access System (EMAS) and an IBM 370/158 batch system, in a more formal manner than the Edinburgh IMP Language Manual (2nd edition). It replaces Part II, Chapters 8 and 9 of the first edition of the IMP language manual (July 1970). The information has been brought up to date with the current release 8 of the IMP compiler.

The syntax of the IMP language is presented in Section 1 and its semantics in Section 2. Section 3 contains lists of the compile time and run time fault messages generated by the compiler.

The reader is referred to the other two documents which describe facilities of the IMP language; The Edinburgh IMP Language Manual (2nd Edition), and The IMP/FORTRAN System Library Manual.

The editors would like to thank Mrs Anne Tweeddale who typed this manual.

Andrew McKendrick,
Gillian T. Watson,
July 1974.

SECTION 1 - SYNTAX

INTRODUCTION

The syntax describes in phrase-structure notation the structure of the various forms of statement which are allowed in the language. The notation used is as follows. A definition is started by P (for PHRASE), and ended by a semi-colon. The symbols '<' and '>' are used to enclose definable items in the IMP language. They stand for themselves ('less than' and 'greater than' respectively) whenever they enclose any character which is not an upper or lower case letter, or a space or prime. The sequence '::=' separates an item from its definition and may be interpreted as 'is defined as'. The character '|' is used to separate alternative definitions and may be interpreted as 'or'. Symbols used in a definition but not enclosed by '<' and '>' denote actual IMP text, i.e. literals.

For example, the phrase structure definition of 'assignment operator' is:

```
P<assop>::=          ==|=|<-|->;
```

The phrase structure definition of 'name list' is:

```
P<name list>::=          <name><rest of name list>;
P<rest of name list>::= ,<name list>|<null>
```

<null> denotes the null text,

```
i.e.   P<null>::=        ;
```

Spaces are ignored everywhere in a program unless the compiler is operating in text mode. The convention used is that, if there is a null alternative, the words 'rest of' will appear immediately after the '<' in the definition. A prime immediately before the '>' of a phrase also denotes a null alternative.

```
P<plus'>::=          +|-|¬|<null>;
```

<name> is termed a built-in phrase and has not been expanded further in terms of literals or phrases. Such built-in phrases are described in detail at the end of the phrase structure.

IMP PHRASE STRUCTURE

All IMP source statements belong to the class P<SS>, where SS stands for source statement, as defined below.

```
P<SS>::=    <unconditional instrn><separator>|
            <if or unless><cond>%THEN<unconditional instrn><else'><separator>|
            <if or unless><cond><then start><separator>|
            <unconditional instrn><if or unless><cond><separator>|
            <while or until><cond>%THEN<unconditional instrn><separator>|
            <while or until><cond>%CYCLE<separator>|
            <unconditional instrn><while or until><cond><separator>|
            %CYCLE<cycparam'>|
            %REPEAT<separator>|
            <type><qname'><name list><separator>|
            <type>%ARRAY<format'><array list><separator>|
            <xown><type><name list><initial'><rest of own declaration><separator>|
            <xown><type>%ARRAY<name><cbpair><const list>|
            %SWITCH<switch list><separator>|
            <extrn'><rt>%SPEC<name><formal parameter defn'><separator>|
            %SPEC<name><formal parameter defn'><separator>|
            <extrn'><rt><name><formal parameter defn'><separator>|
            %BEGIN<separator>|
            %END<separator>|
            %ENDOFPROGRAM<separator>|
            <label>:|
            <name>(<plus'><iconst>):|
            <comment><comment text><separator>|
            %LIST<separator>|
            %ENDOFLIST<separator>|
            %MCODE<separator>|
            %ENDOFMCODE<separator>|
            *<machine instrn><separator>|
            %FINISH<else'><separator>|
            %RECORDFORMAT<name>(<format element><rest of format defn>)<separator>|
            %RECORD<qname'><name list>(<name>)<separator>|
            %RECORDARRAY<format'><array list>(<name>)<separator>|
            <xown>%RECORD<name list>(<name>)<separator>|
            <xown>%RECORDARRAY<name><cbpair>(<name>)<separator>|
            %RECORDSPEC<name><ename''>(<name>)<separator>|
            %REALS<ln><separator>|
            %CONTROL<iconst><separator>|
            %ENDOFFILE<separator>|
            %FAULT<fault list><separator>|
            <separator>;
```

where:

```
P<unconditional instrn>::=  <name><actual parameters'><ename'><assop><exprn><aui'>|
                            <name><actual parameters'><aui'>|
                            -><label>|
                            -><name>(<exprn>)|
                            %PRINTTEXT'<ptext><aui'>|
                            %EXIT|
                            %RETURN|
                            %RESULT<assop><exprn>|
                            %STOP|
                            %MONITOR|
                            %MONITORSTOP;
```

and:

```
P<plus'>::=                         +|-|¬|<null>;
P<exprn>::=                         <plus'><operand><rest of exprn>;
P<rest of exprn>::=                 <operator><operand><rest of exprn>|<null>;
P<operand>::=                       <name><actual parameters'><ename'>|
                                    <const><name><actual parameters'><ename'>|
                                    <const>|(<exprn>)|!<exprn>!;
P<operator>::=                      +|-|!!!|**|//|!!|*|/|&|<<|>>|.;
P<actual parameters'>::=            (<exprn><rest of actual parameters>)|<null>;
P<rest of actual parameters>::=     ,<exprn><rest of actual parameters>|<null>;
P<comma'>::=                        ,|<null>;
P<if or unless>::=                  %IF|%UNLESS;
P<while or until>::=                %WHILE|%UNTIL;
P<cycparam'>::=                     <name><actual parameter'><ename'>=<exprn>,
                                        <exprn>,<exprn>|<null>;
P<then start>::=                    %THEN%START|%START;
P<type>::=                          %INTEGER|%REAL|%BYTE%INTEGER|
                                    %SHORT%INTEGER|%LONG%REAL|
                                    %STRING<qualifier'>;
P<name list>::=                     <name><rest of name list>;
P<rest of name list>::=             ,<name list>|<null>;
P<array list>::=                    <name list>(<bound pair list>)<rest of array list>;
P<rest of array list>::=            ,<array list>|<null>;
P<bound pair list>::=               <exprn>:<exprn><rest of bp list>;
P<rest of bp list>::=               ,<bound pair list>|<null>;
P<cbpair>::=                        (<plus'><iconst>:<plus'><iconst>);
P<switch list>::=                   <name list><cbpair><rest of switch list>;
P<rest of switch list>::=           ,<switch list>|<null>;
P<rt>::=                            %ROUTINE|<type><fm>;
P<array'>::=                        %ARRAY|<null>;
P<fm>::=                            %FN|%MAP;
```

```
P<formal parameter defn'>::=    (<fp list>)|<null>;
P<fp list>::=                   <fp delimiter><name list><rest of fp list>;
P<rest of fp list>::=           <comma'><fp list>|<null>;
P<fp delimiter>::=              <rt>|<type><qname'>|%NAME|
                                %RECORD<array'>%NAME;

P<format element>::=            <type><qname'><name list>|
                                <type>%ARRAY<name list><cbpair><rest of switch list>|
                                %RECORD<array'>%NAME<name list>|
                                %RECORD<name list>(<name>);

P<rest of format defn>::=       ,<format element><rest of format defn>|<null>;
P<xown>::=                      %OWN|%CONST|%EXTERNAL|%EXTRINSIC;
P<rest of own dec>::=           <name list><initial'><rest of own dec>|<null>;
P<format'>::=                   %FORMAT|<null>;
P<qualifier'>::=                (<unsigned integer>)|<null>;
P<cond>::=                      <simple cond>%AND<and cond>|
                                <simple cond>%OR<or cond>|
                                <simple cond>;

P<and cond>::=                  <simple cond><rest of and cond>;
P<rest of and cond>::=          %AND<and cond>|<null>;
P<or cond>::=                   <simple cond><rest of or cond>;
P<rest of or cond>::=           %OR<or cond>|<null>;
P<simple cond>::=               <exprn><comparator><exprn><rest of simple cond>|
                                (<cond>);

P<rest of simple cond>::=       <comparator><exprn>|<null>;
P<comparator>::=                =|#|¬=|>=|>|<=|<|->;
P<label>::=                     <name>|<unsigned integer>;
P<initial'>::=                  =<plus'><const>|<null>;
P<fault list>::=                <unsigned integer><rest of N list>-><label>
                                    <rest of fault list>;

P<rest of N list>::=            ,<unsigned integer><rest of N list>|<null>;
P<rest of fault list>::=        ,<fault list>|<null>;
P<comment>::=                   %COMMENT|!;
P<assop>::=                     ==|<-|=|->;
P<else'>::=                     %ELSE%START|%ELSE<unconditional instrn>|<null>;
P<extrn'>::=                    %EXTERNAL|%SYSTEM|%DYNAMIC|<null>;
P<ename''>::=                   _<name>|<null>;
P<ename'>::=                    _<name><actual parameters'><ename'>|<null>;
P<qname'>::=                    %ARRAYNAME|%NAME|<null>;
P<ln>::=                        %LONG|%NORMAL;
P<aul'>::=                      %AND<unconditional instrn>|<null>;


(P<machine instrn>              See ICL 4/70 Usercode Reference Manual. Note
                                that only a subset of the 4-70 Usercode is
                                allowed and that its direct use in IMP is not
                                recommended)
```

THE BUILT-IN PHRASES.

A few of the phrases used to define IMP syntax are not themselves defined by literals and/or other phrases. These are the built-in phrases where recognition or rejection of the phrase is by executing a piece of program in the compiler. Built-in phrases are used for speed or because of a need to perform operations other than recognition - a constant must be recognised (which could be performed by a phrase definition) and also evaluated (which could not).

The built in phrases are:

P<null>
P<name>
P<const>
P<iconst>
P<unsigned integer>
P<separator>
P<comment text>
P<const list>
P<ptext>

1. Phrase <null>

This merely directs the syntax analysis to proceed to the next item.

2. Phrase <name>

A name must start with a letter. This may be followed by a string of letters and/or digits. The following are valid names:

A, A3, POINTER 27, A 3 B, A2B37

Once the initial letter is found, phrase name will always find a valid name.

3. Phrase <const>

This recognises and evaluates constants. A constant may be a character constant, a decimal constant, a string constant, or a multicharacter, hexadecimal or binary constant.

Character constant.

Character constants consist of any one character from the extended symbol set between quotes (').

'A', '5', '=', ';'

All characters stand for themselves within quotes (except quote itself which is represented by two quotes), including space and newline. The value of the constant is the internal code of the symbol.

Decimal constant.

Any sequence of digits is permitted which may contain one decimal point (.). Leading zero's are permitted - thus valid decimal constants are:

```
12,    123,    0123,   01.23,  12.30,  0.123,  .123000
```

Further, any decimal constant may be followed by the exponent indication (@), an optional '+' or '-' sign and an integer indicating a scaling factor as a power of ten. Thus valid constants containing an exponent are:

```
12.10,  012@+10,    12@-8,  7@0.
```

Constants > 10**75 cause overflow during compilation;
Constants < 10**-75 are taken as zero.

Constants are evaluated double length. The compiler arranges to store real constants double length if they are used in an expression which is to be evaluated double length. Otherwise constants are stored single length.

String constant.

String constants consist of a string of symbols between quotes, all symbols standing for themselves including space and newline but excepting quote itself which is represented within quotes by two quotes. The compiler checks that the length of the string constant does not exceed the maximum length of 255 symbols, and stores it in string format.
A character constant, 'A' say, is converted to a one-character string if the context so demands.

Multicharacter, Hexadecimal and Binary constants.

Multicharacter constants consist of a string of up to four symbols enclosed in quotes and prefixed by the letter M.

```
M'12MZ'
M'?*'
```

Spaces and newlines are significant and a single quote is represented by two quotes. Each symbol occupies one byte of store.

Hexadecimal constants consist of a string of up to eight hexadecimal digits (0,1,.....9,A,B,.....F) enclosed in quotes and prefixed by the letter X.

```
X'FF'
X'127A4CD2'
```

Each hexadecimal digit occupies a location of four bits in length.

Binary constants consist of a string of up to thirty two binary digits (0 or 1) enclosed in quotes and prefixed by the letter B.

```
B'01011110'
```

Each binary digit occupies one bit.

All three forms may appear in arithmetic expressions but are treated as integer. If one is assigned to a real variable it will be converted to floating point form.
If the number of digits or symbols which appear in a constant is less than the maximum permissible, then the value assumed is the same as if the digits or symbols had been right justified in a location of 32 bits and the remaining bit positions filled with zeros.

4. Phrase <iconst>

This is exactly as for phrase <const> except that real and string constants are not accepted.

5. Phrase <unsigned integer>

Accepts only unsigned integer constants - leading zeros are permitted.

6. Phrase <separator>

Accepts only semi-colon or newline character.

7. Phrase <comment text>

Skips text up to the next valid separator.

8. Phrase <const list>

Accepts a list of constants separated from each other by a comma. The list can continue over several lines provided each line other than the last ends with a comma. No constant list is also a valid alternative. Note that <const list> is affected by the type and precision options of the array it is initialising. An incorrect constant (e.g. too large) gives a syntax fault and terminates the <const list>.

9. Phrase <ptext>

Accepts and stores any character string up to the first occurrence of a single quote that is not immediately followed by another quote. The string is stored as it is written, including newlines, spaces and semi-colons but not including the first or terminating quote. Any occurrence of two single quotes together results in one quote only being stored.

SECTION 2 - SEMANTICS

INTRODUCTION

Not every statement which is syntactically correct is meaningful.

For example

```
%OWNBYTEINTEGER K = 256
```

is a syntactically correct statement, and will be recognised as such by the compiler. However, its meaning is not clear, and that part of the compiler which checks semantics will signal a fault. We therefore give below, corresponding to each source statement, a description of the semantic checks made, and of the effect of the statement both at compile and run time.

General rules are:

COMPILE TIME

1. Only statements described in section UI may be made conditional. (See SS2)
2. Syntax errors recognised by the compiler are signalled by SYNTAX followed by a listing of the offending statement. When possible the compiler will indicate the position of the SYNTAX error by outputting a marker (!) under the character where analysis failed. This marker can be approximate only.

   e.g., if the statement

   ```
   %ROUTINE SPECIAL ACTIONS     is mistyped as
   %ROUTINESPECIAL ACTIONS      the failure message would be

   *   111 SYNTAX
       %ROUTINESPECIALACTIONS
                   !
   ```

   The marker is misplaced to the right, as this erroneous line more nearly corresponds to a '%ROUTINESPEC' statement than the intended '%ROUTINE' statement.
3. The semantic errors recognised by the compiler will be signalled by messages of the type

   ```
   FAULT N
   ```

   where N identifies the fault according to the table in Section 3, COMPILE TIME FAULTS.
4. Each block (See SS18,SS19) is said to be at a different textual level from that of any block containing it.

   If blocks are nested to a depth of more than 9 levels, FAULT 34 will be recorded. After this fault has occurred, the compiler will continue to examine any further statements for errors, but will not compile them.

   If more than 10 levels are defined at any time FAULT 35 will be recorded and compilation will cease completely. Each routine is said to be at a different routine level. FAULT 35 occurs if routine levels are nested greater than 5.
5. The compiler allocates itself space, e.g., for dictionaries, depending on the amount of core store available at compile time. If the store available is not sufficient to compile the source program the compiler perforce must stop and a catastrophic FAULT N (N>100) is recorded.

6. Instructions to enter the system MONITOR routine in the event of various run-time fault conditions occurring are compiled into the program. This action can be varied by including appropriate options in the job control statements.
7. In general, as in the case of the SYNTAX message, compilation continues after any FAULT N message has been recorded until the statement %ENDOFPROGRAM is reached, or a %END is found which corresponds to the opening %BEGIN.

   However faults numbered 101 and upwards are catastrophic and compilation ceases.

RUN TIME

8. The effect of MONITOR entry is cessation of execution, accompanied by output of an explanatory message, as detailed in Section 3, RUN TIME FAULTS.

   A POST MORTEM is also output.

A complete list of all faults recognised by the system is given in Section 3.

SEMANTICS OF SOURCE STATEMENTS

SS1 <unconditional instrn><separator>

where: <unconditional instrn> = an instruction which may be made conditional.
<separator> = semi-colon or newline character.

See Unconditional Instructions U11-U111.

SS2 <if or unless><cond>%THEN<unconditional instrn><else'><separator>

where: <if or unless> = %IF or %UNLESS
<cond> = conditional expression

COMPILE TIME
1. Only instructions of the class <unconditional instrn> may be made conditional. (See U11-U111).
2. If multiple unconditional instructions are provided, the effect is exactly as if the individual unconditional instructions were enclosed within a %START-%FINISH grouping (See SS3).

RUN TIME
3. When <if or unless> = %IF: If the conditional expression is true, then the unconditional instruction is obeyed, otherwise it is skipped, and the unconditional instruction following %ELSE (if any) is obeyed.
4. When <if or unless> = %UNLESS: contrariwise.
5. Because of rounding involved in %REAL arithmetic operations, care should be taken in equality comparisons between two expressions either, or both, of which are of %REAL type.
6. See CP1 and CP2 for evaluation of conditional expressions.

SS3 <if or unless><cond><then start><separator>

where: <then start> = %THEN %START or %START

COMPILE TIME
1. A note is made of the position of %START so that it can be associated with the appropriate %FINISH.
2. %START-%FINISH blocks may be nested to any depth.

RUN TIME
3. When <if or unless> = %IF: If the condition is true, all the statements between %START and the appropriate %FINISH are executed, otherwise all the statements between %START and the appropriate %FINISH are skipped and, if a %ELSE follows the %FINISH, the statement(s) following the %ELSE are executed.
4. When <if or unless> = %UNLESS: contrariwise.
5. If a %START-%FINISH-%ELSE block is entered via a jump and not via the %START then the action taken at the %ELSE is not defined.

SS4 <unconditional instrn><if or unless><cond><separator>

COMPILE TIME

1. Treated as <if or unless><cond>%THEN<unconditional instrn><separator>. (See SS2). Note that %ELSE is not allowed here.

SS5 <while or until><cond>%THEN<unconditional instrn><separator>

where: <while or until> = %WHILE or %UNTIL
<cond> = conditional expression

COMPILE TIME

1. If multiple unconditional instructions (connected by %AND) are provided the effect is exactly as if the individual unconditional instruction were enclosed within a %CYCLE %REPEAT group (See SS6).

RUN TIME

2. When <while or until>=%WHILE: The condition is tested and, if false, the next statement is executed. Otherwise the unconditional instruction is obeyed and a branch made to test the condition again.
3. When <while or until>=%UNTIL: The unconditional instruction is obeyed and the condition tested. If false, a further execution of the unconditional instruction is initiated. Otherwise the next statement is obeyed.
4. If executing the unconditional instruction does not affect any of the variables in the condition, the program is liable to loop indefinitely.
5. Note that with %UNTIL the unconditional instruction will always be executed at least once, but that with %WHILE, the unconditional instruction may not be executed at all.

SS6 <while or until><cond>%CYCLE<separator>

COMPILE TIME

1. A note is made of the position of the %CYCLE so that it can be associated with the appropriate %REPEAT. If at the end of a block, no such %REPEAT has been found, FAULT 13 will be recorded.
2. If <while or until>=%UNTIL the condition is saved so that it can be used at the %REPEAT.

RUN TIME

3. When <while or until>=%WHILE: The condition is tested and, if false, the statement after the appropriate %REPEAT executed. Otherwise the cycle is traversed and a branch is made to test the condition again.
4. When <while or until> = %UNTIL: The cycle is traversed and then the condition is tested. If false, a further traverse of the cycle is initiated.
5. There is no automatic increment of any variable. It is the programmer's responsibility to ensure that executing the cycle is liable to change one of the variables in the condition.
6. As SS5.5.
7. If a jump is made into the middle of a cycle, the effect on executing the %REPEAT is not defined.

SS7 <unconditional instruction><while or until><cond><separator>

COMPILE TIME

1. Treated exactly as <while or until><cond>%THEN<unconditional instrn>. (See SS5).

SS8 %CYCLE <cycparam'>

where: <cycparam'> = <name><actual parameters'><ename'>=<exprn>, <exprn>,<exprn><separator> or <null>

COMPILE TIME

1. A record is made so that a %REPEAT in the same block can be associated uniquely with this %CYCLE. If at the end of a block, no such %REPEAT has been found, FAULT 13 will be recorded. (See SS19 and SS20).
2. If cycle parameters are provided then <name><actual parameters'><ename'> must be a %INTEGER variable (i.e. not %BYTEINTEGER or %SHORTINTEGER), otherwise FAULT 25 is recorded. If the three expressions are not of integer type then FAULT 24 is recorded.
3. Cycles may be nested to any depth, but a %CYCLE and its associated %REPEAT must be in the same block, and the same %START-%FINISH group.
4. No check is made by the compiler whether nested cycles employ the same recorded variable and hence this is entirely the users' responsibility.

e.g.
```
%CYCLE K = 1,1,10
%CYCLE K = 1,1,5
%REPEAT
%REPEAT
```

is syntactically acceptable but will cycle indefinitely.

RUN TIME

5. For open cycles no action is taken. For closed cycles 6-10 apply.
6. The three expressions (p,q,r say) are evaluated and stored upon first entering the cycle. The cycle is monitored, INVALID CYCLE, if,

either (r-p)/q # n, where n is an integer >= 0
or q = 0.

7. The address of the variable <name><actual parameters'><ename'> is recorded.
8. The recorded variable <name><actual parameters'><ename'> is set at the beginning of the first traverse of the cycle to the value p. When the associated %REPEAT is encountered, the current value of <name><actual parameters'><ename'> is tested against the final value r. If these two values are equal, control passes to the next statement after the %REPEAT, otherwise, the current value of <name><actual parameters'><ename'> is incremented by an amount q, and another traverse of the cycle is begun.
9. Transfers of control within the body of a cycle, or out from the body of a cycle, are allowed in the usual way. However, if control is transferred into the body of the cycle without going via the %CYCLE statement, the increment variable will not be assigned and MONITOR will be entered when the %REPEAT is obeyed.

10. Assignments to the recorded variable should not be made within the body of the cycle without full regard to the possible consequences. For instance, in view of 8 above it will be clear that a cycle such as

```
%CYCLE K = 1,1,10                    %CYCLE
    K = 0          is equivalent to
%REPEAT                              %REPEAT
```

SS9 %REPEAT<separator>

COMPILE TIME

1. Each %REPEAT is associated with the last unassociated %CYCLE statement in the same block. If no such %CYCLE statement exists, FAULT 1 is recorded. See SS6 or SS8 as appropriate.

RUN TIME

2. Note that the MONITOR message 'UNASSIGNED VARIABLE' can be obtained from a %REPEAT as a result of SS8.9 or SS6.2.

SS10 <type><qname'><name list><separator>

where: <type> is %REAL, %INTEGER, %BYTEINTEGER, %SHORTINTEGER, %LONGREAL or %STRING <qualifier'>
<qualifier'> = <unsigned integer> or <null>
<qname'> = %ARRAYNAME or %NAME or <null>

COMPILE TIME

1. If any name on <name list> has been declared before in the current block then FAULT 7 is recorded, otherwise arrangements are made for a location in the store to be assigned to each name for use within the current block at run time.
2. The type of each name in the current block is recorded.
3. Variables declared as %BYTE, %SHORT and %LONG are allocated 8, 16 and 64 bits each, respectively. If no length is specified then 32 bits are allocated. However, pointer variables are allocated 32 bits and %ARRAYNAME variables 128 bits.
4. The maximum string length, <unsigned integer>, is checked to ensure it lies between 1 and 255 and,if so, code is compiled to allocate space at run time. Otherwise FAULT 70 is recorded.
5. A check is made that the declarations occur at the head of a block (i.e. before any label or jump instructions occur). If not, FAULT 40 is recorded.

RUN TIME

6. The values of the names are lost on leaving the block, i.e. the declared storage is deallocated and made available for other use.
7. Pointer variables are specialised and should not be used unless the stack mechanism is well understood. Unpredictable and dire consequences will follow if a pointer variable is left pointing to a variable when it is deallocated as in SS10.6 or SS11.9.

SS11 <type>%ARRAY<format'><array list><separator>

where: <format'> = %FORMAT or <null>

COMPILE TIME

1. If any name on <array list> has been declared before in the block then FAULT 7 is recorded, otherwise the name and the type of the name in the current block are recorded.
2. If expressions defining the array bounds are not %INTEGER expressions then FAULT 24 is recorded.
3. A check is made for string arrays that <unsigned integer> lies between 1 and 255. If not, FAULT 70 is recorded.
4. A check is made that the declarations occur at the head of a block (i.e. before any label or jump instruction occurs). If not, FAULT 40 is recorded.
5. If no faults have been found in the declaration, instructions for calculating the bounds at run time are compiled into the program.
6. If <format'> = %FORMAT, no space is allocated. Otherwise instructions for allocating space at run time are compiled into the program. Array formats are required for the special array mapping function ARRAY only. If they are used in any other context, FAULT 29 will be recorded.

RUN TIME

7. The bounds of each array are computed, and (subject to 6) space is allocated. (See AE1-AE4 for semantics of <exprn>).
8. However, MONITOR is entered if any lower bound exceeds the corresponding upper bound, 'ARRAY INSIDE OUT'. If there is insufficient space available, the program will be monitored 'NOT ENOUGH STORE'.
9. This space is made available for other use at the end of the block.
10. The program is monitored if an attempt is made to refer to an array element outside the declared bounds. This checking can be suppressed by job control options.
11. The array will always start on a double word boundary.
12. Note that no initialisation takes place.

SS12 <xown><type><name list><initial'><rest of own declaration>

where: <xown> = %OWN or %CONST or %EXTERNAL or %EXTRINSIC
<initial'> = <plus'><const> or <null>

COMPILE TIME

1. This statement declares static variables where space is allocated at compile time rather than run time. They make less efficient use of space than dynamic variables, but possess additional attributes. %OWN and %CONST variables are allocated in the non-sharable (GLAP) and sharable (code) areas of the program file respectively. %CONST variables are therefore 'read only' and any attempt to assign to them results in FAULT 29 being recorded. %EXTERNAL variables are similar to %OWN but their position is recorded in the LOAD DATA so that other routines may use them. %EXTRINSIC variables are assumed to exist in some other program file which will be present at run time. No space is allocated and the variables cannot be initialised or FAULT 46 will be recorded.
2. The constants provided are checked to be suitable for the variable and, in the case of strings, that the length of string constant is <= declared maximum length. If this check fails, FAULT 44 is recorded. If <initial'> = <null> then zero (or null string) is pre-assigned.

3. For %STRING variables a check is made that the declared maximum length lies between 1 and 255.
4. The required amount of space in the appropriate area is reserved and initialised.
5. Static variables may be declared global to, and used to communicate between, a file of external routines.

RUN TIME

6. Since the space for static variables is allocated at compile time, if a new value is assigned to a static variable, the new value will be found in the variable on re-entry to the block (cf OWN variables in ALGOL 60).
7. %EXTERNAL and %EXTRINSIC variables enable IMP routines to access COMMON areas produced by the Edinburgh Fortran Compiler. The restrictions involved are outside the scope of this document.

SS13 <xown><type>%ARRAY<name><cbpair><const list>

where: <xown> = %OWN or %CONST or %EXTERNAL or %EXTRINSIC

COMPILE TIME

1. As SS12.1
2. The array bounds are checked - if the lower exceeds the upper, FAULT 43 is recorded.
3. If a <const list> is provided, the number of constants is checked against the declared bounds. If there is a discrepancy, FAULT 45 is recorded.
4. If no <const list> is given, the appropriate number of zeros (or null strings) are pre-assigned.
5. Variables declared as %BYTE, %SHORT and %LONG are allocated 8, 16 and 64 bits each, respectively. If no length is specified, then 32 bits are allocated.
6. For string arrays a check is made that <unsigned integer> is present and lies between 1 and 255. If not, FAULT 70 is recorded.
7. Note that phrase constlist is primed to accept constants of the size and type defined by <type>. Any incorrect constants will give a SYNTAX fault.

RUN TIME

8. Although static arrays are declared in a block and can only be referred to within that block, the actual array variables are not held in the dynamically allocated storage for that block. Consequently, if a new value is assigned to a static array element, the NEW value will be found on re-entry (cf ALGOL).

SS14 %SWITCH<switch list><separator>

COMPILE TIME

1. If any name in <switch list> has been declared before in this block then FAULT 7 is recorded, otherwise the name and type of name in this block are recorded.
2. The bounds are recorded and space is allocated for the storage of the addresses corresponding to the switch labels. (See SS22).
3. If the lower bound exceeds the upper bound then FAULT 43 is recorded and compilation continues, both bounds being set to the greater bound.
4. If either of the bounds is outside the range -32768:32767 then FAULT 18 will be recorded.

RUN TIME - No action.

SS15 <extrn'><rt>%SPEC<name><formal parameter defn'><separator>

where: <rt> = %ROUTINE, <type>%FN or <type>%MAP
<extrn'> = %SYSTEM or %EXTERNAL or %DYNAMIC or <null>

COMPILE TIME

1. The <name> is declared as an <rt> type name of the current block. FAULT 7 is recorded if the name has been set before in the current block.
2. A record is made of the type of each formal parameter. The names are not recorded, and it is only the order of the parameter types which is made use of.
3. %SYSTEM, %EXTERNAL or %DYNAMIC routine specs are used to set up a reference so that the routine can be found at program loading.
4. Any internal <rt> specified must be described somewhere later in the same block, otherwise FAULT 28 is recorded at the end of this block (except as in 5 below).
5. Within a <rt> description which uses routine-type parameters there must be, corresponding to each such parameter, a routine type %SPEC which appears before the first reference to that parameter. However, in the case of routine-type formal parameters, there must be no description corresponding to the required %SPEC.
6. The reduced form %SPEC <name><formal parameter defn'><separator> may only be used as an alternative to the above where the specification is of <rt> type parameter, in which case the semantics applicable are those detailed in SS16.
7. It is the programmer's responsibility to make certain that the number and type of parameters in %EXTERNALROUTINESPEC correspond to those in the library %ROUTINE. The compiler cannot check.

RUN TIME

8. Any %EXTERNAL routine must be available in the library files declared in the job head or the program will not be loaded. %SYSTEM routines are not for general use.
9. No attempt to load a routine referenced by a %DYNAMICROUTINESPEC is made until the routine is called. If the routine cannot then be found, MONITOR is entered. This form of declaration is useful for routines that are not necessarily called during the execution of a program.

SS16 %SPEC <name><formal parameter defn'><separator>

COMPILE TIME
1. This statement may only appear within a %ROUTINE, %FN or %MAP block, otherwise FAULT 53 is recorded.
2. If the <name> is not that of an <rt> type parameter in the %ROUTINE, %FN or %MAP parameter list then FAULT 3 is recorded, otherwise the <name> is declared as an <rt> type name within the current %ROUTINE, %FN or %MAP block.
3. No <rt> description corresponding to this %SPEC may appear in the current %ROUTINE, %FN or %MAP block, otherwise FAULT 7 will be recorded.
4. A record is made etc as in SS15.2 above.
5. Corresponding to each <rt> type parameter of the current %ROUTINE, %FN or %MAP block, there must be a %SPEC of this type, or that above (SS15), which appears before the first reference to that parameter. If not, FAULT 21 will be recorded.

RUN TIME - No action.

SS17 <extrn'><rt><name><formal parameter defn'><separator>

where: <rt> = %ROUTINE, <type> %FN or <type> %MAP

COMPILE TIME
1. If the <name> has not been specified (see SS15) in the current block, then this source statement is first treated exactly as
   <rt> %SPEC <name><formal parameter defn'><separator>.
2. This statement marks the beginning of a new block.
3. The formal parameter names are declared in the new block in the appropriate way. FAULT 9 is recorded if the type of the first formal parameter in the heading is not the same as the type of the first parameter given in the corresponding %SPEC if any; similarly for the second parameter, and so on. FAULT 8 or FAULT 10 will also be recorded if the number of parameters differs from that of the spec.
4. An <rt> can only be entered by an <rt> call, and therefore the compiler inserts a jump around the description.
5. Compilation of the <rt> follows. Instructions are planted to store the current diagnostic pointers and to reset the pointers to describe the new block.
6. The form %EXTERNALROUTINE may occur only in a library file terminated by %ENDOFFILE.
7. The form %DYNAMICROUTINE is treated exactly as %EXTERNALROUTINE.
8. Parameters of type %REAL %ARRAY %NAME, %INTEGER %ARRAY %NAME etc. have no dimensions specified. The compiler takes the dimensions from the number of parameters at the first reference encountered.

RUN TIME
9. The complete block is skipped.

SS18 %BEGIN<separator>

COMPILE TIME
1. This statement marks the beginning of a new block.
2. The first statement of a program will normally, but not necessarily, be this statement.
3. Instructions are planted to store the current diagnostic pointers and to reset the pointers to describe the new block.

RUN TIME
4. Control passes into the block.

SS19 %END<separator>

COMPILE TIME
1. Denotes the textual end of a block (%BEGIN, %ROUTINE, %FN or %MAP).
2. Denotes the dynamic end of a %BEGIN block and may denote the dynamic end of a %ROUTINE.
3. Labels of the block which have been used but not set are recorded with FAULT 11.
4. A check is made to ensure that each %CYCLE statement has been associated uniquely with a %REPEAT. If not, FAULT 13 is recorded.
5. A check is made to ensure that all %START statements have been associated uniquely with a %FINISH. If not, FAULT 53 is recorded.
6. All the names declared in this block are unset. The pointers stored on entry are restored.
7. If the block is a %ROUTINE then %END is treated as %RETURN; %END. If the block is a %FN or %MAP, %END is treated as %MONITOR; %END.
8. If the %END corresponds to the first %BEGIN at the head of the program, FAULT 14 is recorded and compilation ceases.
9. Any names or labels declared in this block but not used are listed. Such names and labels result in a less efficient object program.

RUN TIME
10. All local working space is lost except that declared statically (See SS12 or SS13).

SS20 %ENDOFPROGRAM<separator>

COMPILE TIME
1. Exactly as %END<separator> (See SS19).
2. The unconditional instruction %STOP is compiled.
3. FAULT 15 is recorded if this statement is not the %END corresponding to the first %BEGIN of the program.
4. Compilation ceases. Thus this must be the last statement of the program.
5. If no faults have been recorded, the object program is completed.

RUN TIME
6. Execution ceases.

SS21 <label>:

where: <label> = <name> or <unsigned integer>

COMPILE TIME
1. If the label is already on the label list associated with this block, then FAULT 2 is recorded.
2. Otherwise the label and the address of the next compiled instruction are added to the label list.
3. <unsigned integer> must be in the range 1 to 16383.
4. Labels set within any textual level apply to that level only, see 1 above. Thus, control may not be transferred from one textual level to another by jumps to labels. Note that in the following situation

```
1: %BEGIN
    .
    .
1: %END
```

the first label 1 is outside, while the second is inside, the textual level considered, so no fault is recorded.

RUN TIME - No action.

SS22 <name>(<plus'><iconst>):

COMPILE TIME
1. <name> must have been declared in the current block as a %SWITCH variable (otherwise FAULT 4 will be recorded).
2. The signed quantity <plus'><iconst> must be within the bounds declared for this switch variable, otherwise FAULT 5 is recorded.
3. This switch label must not have been set before in this block, otherwise FAULT 6 is recorded.
4. If a fault has not been recorded the address of the next instruction is recorded as the address of this switch label.

RUN TIME - No action.

SS23 <comment><comment text><separator>

where: <comment> = %COMMENT or !

COMPILE TIME
1. The statement is ignored.

RUN TIME - No action.

SS24 %LIST<separator>

COMPILE TIME
1. Source statement %LIST causes the input program to be listed on the line printer, together with line numbers, during compilation. The first line of output is the first after the %LIST statement.
2. No code is compiled in response to this statement.
3. A %LIST statement during listing has no effect.

RUN TIME - No action.

SS25 %ENDOFLIST<separator>

COMPILE TIME
1. Source statement %ENDOFLIST causes the listing begun by %LIST to cease. The last statement listed is %ENDOFLIST.
2. No code is compiled in response to this statement.
3. An %ENDOFLIST elsewhere in a program has no effect if no listing is being effected.

RUN TIME - No action.

SS26 %MCODE<separator>

This source statement no longer has any effect, and is retained only for compatibility.

SS27 %ENDOFMCODE<separator>

This source statement no longer has any effect, and is retained only for compatibility.

SS28 *<machine instrn><separator>

Most machine instructions are permitted. See System 4 Usercode Reference Manual for the valid forms of machine instruction.

The use of Usercode in general purpose programs is unnecessary and is not recommended.

This statement is only valid in the IMP system compiler.

SS29 %FINISH<else'><separator>

where: <else'> = %ELSE %START or %ELSE <unconditional instrn> or <null>

COMPILE TIME
1. The %FINISH is associated with the last unassociated %START in the same block. If no such %START exists, FAULT 51 is recorded.
2. A check is made that %CYCLE and %REPEAT have corresponded within the %START-%FINISH pair. If not, FAULT 52 is recorded.
3. The jump instruction around the %START...%FINISH is filled.
4. If %ELSE follows the %FINISH a check is made that the occurrence of %ELSE is legal. If not, FAULT 47 is recorded.

RUN TIME - No action.

SS30 %RECORDFORMAT<name>(<format element><rest of format defn>)<separator>

COMPILE TIME
1. A check is made that the name of the record format has not been previously declared at this level.
2. The names given in <record format defn> are used to identify elements of the record. However these names are not 'set' and may be redeclared in the same block without causing a fault.
3. No code is compiled for this statement.

RUN TIME - No action.

SS31 %RECORD<qname'><name list>(<name>)<separator>

COMPILE TIME
1. A check is made that <name> is a valid %RECORDFORMAT name. If not, FAULT 62 is recorded.
2. A check is made that none of the names in <name list> have been previously declared at this level. If any have, FAULT 7 is recorded.
3. Instructions are compiled to allocate space.
4. The declaration must occur at the head of a block or FAULT 40 is recorded.

RUN TIME
5. Space is allocated - each record starts on a double word boundary.

SS32 %RECORDARRAY<format'><array list>(<name>)<separator>

COMPILE TIME
1. A check is made that <name> is a valid %RECORDFORMAT name. If not, FAULT 62 is recorded.
2. A check is made that none of the names in <array list> have been previously declared at this level. If any have, FAULT 7 is recorded.
3. As for ordinary arrays SS11.2-6.

RUN TIME
4. As for ordinary arrays SS11.7-10.
5. Record arrays start on a double word boundary but the records are packed together as closely as alignment permits.

SS33 <xown>%RECORD<name list>(<name>)<separator>

COMPILE TIME
1. A check is made that (<name>) is a valid %RECORDFORMAT name. If not, FAULT 62 is recorded.
2. Exactly as for SS12.1,3-5 for ordinary static variables.
3. Note that no initialisation is permitted. The static records will start on a double word boundary and be cleared to zero.

RUN TIME
4. As SS12.6-7

SS34 <xown>%RECORDARRAY<name><cbpair>(<name>)<separator>

COMPILE TIME
1. A check is made that (<name>) is a valid %RECORDFORMAT name. If not, FAULT 62 is recorded.
2. As SS13.2 for ordinary static arrays.
3. No initialisation is permitted. Each static record array will start on a double word boundary and will be cleared to zero.

RUN TIME
4. As SS13.8

SS35 %RECORDSPEC<name><ename''>(<name>)<separator>

COMPILE TIME
1. A check is made that (<name>) is a valid %RECORDFORMAT name. If not, FAULT 62 is recorded.
2. <name><ename''> must represent either
   a record <array'> name in the formal parameter list of the current routine, fn or map
   or
   a record <array'> name in a %RECORDFORMAT declared at this level.
   If neither applies, FAULT 63 occurs.
3. This statement is currently used to assign a format to a %RECORDNAME. This must be done before any reference is made to the name or FAULT 21 occurs.
4. No code is compiled for this statement.

RUN TIME - No action

SS36 %REALS<ln><separator>

where: <ln> = %LONG or %NORMAL

COMPILE TIME
1. If <ln> is %LONG then every occurrence of %REAL thereafter is compiled as %LONGREAL, until a %REALSNORMAL is met.
2. %REALSNORMAL cancels %REALSLONG.

RUN TIME - No action.

SS37 %CONTROL<Iconst><separator>

COMPILE TIME
1. This is a generalised flag setting statement to control compile time options. The exact actions are liable to change and it is envisaged that the statement will only be used via the macro scheme. However %CONTROL 0 will always turn off all checking and diagnostics.
2. This statement is only valid in the IMP system compiler.

RUN TIME - No action

SS38 %ENDOFFILE<separator>

COMPILE TIME
1. Terminates the compilation of a file of external routines.
2. Compilation ceases.
3. If no faults have been recorded then the object program is completed.

RUN TIME - No action.

SS39 %FAULT<fault list><separator>

COMPILE TIME
1. The labels referred to within the jump instruction following each <rest of N list> must obey the normal rules for jump labels. (See UI3.)
2. This statement may only appear at the basic textual level of the program, i.e. within %BEGIN...%ENDOFPROGRAM, but not within any other blocks or routines, otherwise it will be faulted, (FAULT 26).
3. If the faults in <fault list> include non trappable faults then FAULT 36 is recorded.
4. This statement counts as a branch and any declaration occurring after it will be faulted.
5. For the faults which may be trapped and their corresponding numbers, which appear in the <rest of N list>s, see Section 3.

RUN TIME
6. Certain information concerning the current state of the program is stored so that if a fault, which has been allowed for, occurs subsequently, this information may be used to restart the program from the relevant label; i.e. the statement must be executed at some time during the normal flow of control of the program in order for any faults to be trapped thereafter.
7. The label to be jumped to when any particular fault is trapped may be changed dynamically by executing a further fault statement in which the new label appears.

SS40 <separator>

COMPILE TIME
1. No action. This represents merely a redundant separator between statements.

SEMANTICS OF UNCONDITIONAL INSTRUCTIONS

UI1 <name><actual parameters'><ename'><assop><exprn><aui'>

where: <assop> is == or = or <- or ->
<aui'> is %AND<unconditional instrn> or <null>

A INTEGER OR REAL ASSIGNMENT

COMPILE TIME

1. If the <name> is of integer type and the RHS is of real type then FAULT 24 is recorded.
2. <name> must be capable of being a destination for a value. Thus a function, a routine name, switch or a %CONST variable is not allowed, otherwise FAULT 29 is recorded.
3. For details of <exprn>, see AE1-4.

RUN TIME

4. If <assop> is '=', the RHS is evaluated and the value is assigned to the destination given by the LHS provided that the LENGTH associated with the LHS is great enough to hold the value calculated. If the value is too great the program is monitored.
5. If <assop> is '<-', the least significant 8 bits (if the LHS is a %BYTEINTEGER) or 16 bits (if the LHS is a %SHORTINTEGER) are assigned to the LHS. The remaining bits of the RHS value are ignored. In %REAL expressions '<-' is exactly equivalent to '='.

B STRING ASSIGNMENT

COMPILE TIME

6. If <assop> is '->', then a resolution (see AE4) is compiled.
7. <name> must be capable of being the address of a string. If not, FAULT 29 is recorded.
8. See AE3 for details of string expressions.

RUN TIME

9. If <assop> is '=', then the RHS is evaluated and assigned to the LHS provided the declared length N of the LHS is sufficient to hold the RHS. If not, the program is monitored.
10. If <assop> is '<-', then the first N characters of the RHS are assigned and the remainder lost.

C RECORD ASSIGNMENT

COMPILE TIME

11. The <name> on the left hand side of the assignment must be a record variable.
12. The <exprn> on the right hand side must be either zero or a single record variable.
13. If the operator is '=', then both records must be of the same size, otherwise FAULT 66 is recorded.

RUN TIME

14. If the <exprn> is zero then the record on the left hand side is cleared to all zeros.
15. If the operator is '=', then the record on the right hand side is copied to the record on the left hand side.
16. If the operator is '<-', then a copy is done as in 2, but restricted in extent to the size of the smaller record.

D ADDRESS ASSIGNMENT (i.e. <assop> is ==)

COMPILE TIME

17. The <name> must be capable of receiving an address (i.e. of type %ARRAYNAME or %NAME). If not, FAULT 82 is recorded.
18. The <exprn> must evaluate to a single variable which has an address, otherwise FAULT 81 is recorded.
19. The variable on the right hand side must have the same type and precision as the left hand side or FAULT 83 occurs.
20. Care is required to ensure that a pointer variable is not left pointing to a variable when the latter is undeclared.

RUN TIME

21. The address of the right hand side is evaluated and stored in the space reserved for the pointer variable.

UI2 <name><actual parameters'><aui'>

COMPILE TIME

1. A check is made to ensure that <name> has been declared as a %ROUTINE. If not, FAULT 17 is recorded.
2. Checks are made to ensure that the actual parameters in <actual parameters'> are consistent with the formal parameters of the %ROUTINESPEC for <name>, both in type and number. If the check fails FAULTS 22 and/or 19 are recorded.
3. The LENGTHs of formal and actual parameters are checked to match if the parameters are %NAME type.
4. If no faults have been found, the %ROUTINE call is compiled.
5. For rules regarding formal-actual parameter correspondence see edition 2 of the Edinburgh IMP Language Manual, Section 6.
6. Similar rules apply to the passing of parameters to other <rt> types.

RUN TIME

7. The actual parameters are evaluated and passed on for use by the %ROUTINE body. The mode of passing on is analogous to the assignment operator '=', i.e. an assignment to a %BYTEINTEGER or a %SHORTINTEGER is checked to be of a sufficiently small value.
8. The %ROUTINE body is entered.
9. For parameters of type %REALARRAYNAME and %INTEGERARRAYNAME, a check is made that the actual parameter has the correct number of dimensions. If not, a non-trappable fault occurs.
10. When the %ROUTINE body is left, control is returned to the instruction following the one in which <name><actual parameters'> appears.

UI3 -> <label>

COMPILE TIME

1. As the label corresponding to this jump may not yet be set in this block, no attempt is made to compile the instruction until the end of the block is encountered. If the label is omitted, FAULT 11 followed by the offending <label> is recorded at the end of the block.

RUN TIME

2. The normal sequence of instructions is broken and the next instruction to be obeyed is taken at label <label> of the block in which this jump instruction appears.

UI4 -> <name>(<exprn>)

COMPILE TIME

1. Unless <name> is a switch variable declared in the current block, FAULT 4 is recorded.
2. Unless <exprn> is an integer expression, FAULT 24 is recorded.
3. A jump to a switch label has the property of an ordinary jump in that control can only be transferred within the same block level. The scope of a switch declaration extends over the block, unlike the scope of other declarations. Consequently a jump may not be made to a switch label in an enclosed or enclosing block.

RUN TIME

4. The expression is evaluated and, in checking mode, a check is made to see that the value is within the bounds of the switch vector, as declared, and that a switch label exists corresponding to this value of the argument. If the label exists, control is transferred to that label, otherwise the program is monitored, 'SWITCH VARIABLE NOT SET'.
5. If an attempt is made to transfer control to a non-existent switch label in non-checking mode, the result is not defined.

UI5 %PRINTTEXT'<ptext><aul'>

COMPILE TIME

1. <ptext> terminates with any single quote that is not followed immediately by another quote.
2. <ptext> is stored exactly as it is written, including newlines and semi-colons, but not including the first quote or the terminating quote.
3. A single quote is represented by two quotes.
4. The preferred form of this statement is the more general PRINTSTRING ('...'). This old form is maintained for compatibility with other IMP compilers without type %STRING.

RUN TIME

5. <ptext> is output as stored.

U16 %EXIT

COMPILE TIME
1. A check is made that this statement occurs in a %CYCLE-%REPEAT group at the current level. If not, FAULT 54 occurs.

RUN TIME
2. Control is transferred unconditionally to the statement following the %REPEAT of the enclosing %CYCLE-%REPEAT group.

U17 %RETURN

COMPILE TIME
1. FAULT 30 is recorded unless this instruction appears in a block delimited by %ROUTINE...%END.
2. This instruction may appear more than once in such a block.
3. The %END corresponding to the %ROUTINE heading is regarded as %RETURN; %END so that an exit from a %ROUTINE can be made by running on to its %END.

RUN TIME
4. This instruction is the dynamic end of a %ROUTINE block and control passes back to the instruction following the routine call which caused the entry.
5. Diagnostic pointers are restored.
6. The %ROUTINE is regarded as a block and, consequently, on exit from this block all of the local working space is deallocated in the usual way.

U18 %RESULT<assop><exprn>

COMPILE TIME
1. May appear only in a %FN or %MAP block, otherwise FAULT 31 is recorded.
2. For functions the expression must be of the same type as that of the %FN.
3. The form %RESULT== is provided for, and restricted to, %MAP blocks, otherwise FAULT 31 is recorded. If any other form of expression is used for %MAP blocks the programmer is responsible for validating the address produced.

RUN TIME
4. The value of the expression is presented as the result of the %FN or %MAP (in the case of %MAP it is an address) in the same way that values are assigned to variables. (See U11).
5. Control passes back to the block containing the %FN or %MAP block, to complete evaluation of the expression within which the %FN or %MAP was invoked.

U19 %STOP

COMPILE TIME
1. May appear any number of times anywhere in the program.

RUN TIME
2. Execution ceases.

U110 %MONITOR

COMPILE TIME
1. Code is planted to enter the system MONITOR routine.

RUN TIME
2. MONITOR is entered to produce diagnostic information and then execution continues.

U111 %MONITORSTOP

COMPILE TIME
1. Treated exactly as %MONITOR; %STOP

RUN TIME
2. MONITOR is entered to produce a post mortem, and execution ceases.

<exprn>

This phrase is treated as either an %INTEGER expression, a %REAL expression, or a %STRING expression depending upon the context.

AE1 INTEGER EXPRESSION

COMPILE TIME

1. A check is made that all operands are integer operands (i.e. integer variables or integer functions valid at the current level, or integer constants).
2. If constant precedes a name without an operator, then a multiplication is implied.
3. Word, short and byte integers may be mixed at will in an expression. All integer expressions are calculated in a 32 bit field. The sign bit of %SHORTINTEGER variables is propagated to maintain the same (positive or negative) value. Zeros are propagated for %BYTEINTEGER variables, so that they may hold only positive integers.
4. Valid constant forms of operand are
   A decimal integer e.g. 117
   A binary constant e.g. B'10101'.
   A multi character ISO constant e.g. M'XY'.
   A hexadecimal constant e.g. X'FF11'.
5. Constants are assembled at the lower end of a 32 bit word. Each item of a binary constant takes 1 bit, each item of a hexadecimal constant takes 4 bits and each item of a multi character constant takes 8 bits. A fault is recorded if the assembled constant is more than 32 bits long. (N.B. newlines and spaces are not ignored between quotes and to represent ' in a multi character constant it must be written as '').
6. If an operator is found to be acting on two constant operands, then the operation is interpreted at compile time. If overflow occurs during interpretation, FAULT 38 is recorded.

RUN TIME

7. %INTEGER variables are held in 32-bit twos-complement fixed point form and must therefore lie in the range -2**31 (-2147483648) to 2**31-1. %SHORTINTEGER variables are held in 16-bit twos-complement fixed point form and must therefore lie in the range -2**15 (-32768) to 2**15-1. %BYTEINTEGER variables are held in 8-bit fixed point form and must lie in the range 0 to 255 (2**8-1).
8. Sub-expressions in brackets (or modulus signs) are evaluated first. After this, the order of precedence between operators is (highest precedence first):

   ¬
   ** << >>
   * / // &
   + - ! !!

   Where two adjacent operators are of equal precedence, operations are carried out from left to right.
9. An initial minus sign is treated exactly as '0-'.
10. An initial 'not' has highest precedence and has the effect of 'exclusive or' with X'FFFFFFFF'. Thus ¬A**B is exactly equivalent to (X'FFFFFFFF'!!A)**B.

11. The program is monitored 'INTEGER OVERFLOW' if at any point in the calculation the 32 bit capacity is exceeded.
12. Exponentiation is carried out by repeated multiplication. The program is monitored immediately 'ILLEGAL EXPONENT' if the exponent n lies outside the range 0<=n<=63. (Note that smaller exponents may also cause a monitor signal by overflowing while attempting to execute, for example, 2**50).
13. 0**0 gives the result 1 (as, of course, does any other quantity raised to the power zero).
14. The operators '&', '!', '!!' have the effects of logical AND, OR and EXCLUSIVE OR respectively.
15. There are two forms of division, represented by '/' and '//'. For the former, the program is monitored 'NON INTEGER QUOTIENT' whenever a division results in a non-integral quotient. In the case of the latter the remainder is ignored. Rounding in both forms of division occurs so that the remainder has the same sign as the dividend. Thus:

```
 5//2  = 2          5//-2 = -2
-5//-2 = 2         -5//2  = -2
```

    Note the significance of the order in which operations are carried out in the following examples in which I is an integer:-

```
I=(I+1)/2*I     (fails if I is even)
I=I*(I+1)/2     (valid for all values of I)
```

16. An attempt to divide by zero causes the program to be monitored 'DIVIDE ERROR'.
17. The operators '<<' and '>>' denote LOGICAL LEFT shift and LOGICAL RIGHT shift respectively. No overflow can be caused; bits shifted out are lost; spaces created are filled with ZEROS. The bottom six bits only of the second operand are considered. Thus, the effect of a<<b is the same as that of a<<(b&X'3F'). Thus a<<b is NOT the same as a>>(-b) places. Care should be exercised with arithmetic e.g. (-1)>>1 = 2**31-1, but -1>>1 = 0.

AE2 REAL EXPRESSION

COMPILE TIME

1. Operands can be either real or integer, except that the operand immediately following the operator '**' must satisfy the conditions for an integer expression.
2. If a constant precedes a name without an operation, then a multiplication is implied.
3. A real expression is evaluated double length if the LHS is double length or any of the operands are double length. Otherwise single length arithmetic is used. A sub expression containing only single length operands is evaluated single length.
4. Conversions of single length to double length are slow. Mixed length expressions should be avoided where possible.
5. If an operator is found to be acting on two constant operands, then the operation is interpreted at compile time. If overflow occurs during interpretation, FAULT 38 is recorded.

RUN TIME

6. Real numbers are held in floating point form (one sign bit, 7-bit hexadecimal exponent, 24-bit mantissa for %REAL or 56 bit mantissa for %LONGREAL). Integer quantities occurring in a real expression are immediately converted to floating point form, except where a sub-expression in brackets or modulus signs consists wholly of integer operands and the operators '+', '-' and '*'. In this case the sub-expression is evaluated fixed point, and then converted. The range of possible values of real numbers is approximately -7*10**75 to 7*10**75.
7. Sub-expressions in brackets (or modulus signs) are evaluated first. After this, the order of precedence between operators is (highest precedence first):

```
**
* /
+ -
```

   Where two adjacent operators are of equal precedence, operations are carried out from left to right.
8. An initial minus sign is treated as '0-'.
9. The program is monitored 'REAL OVERFLOW' if at any stage in the calculation a number with a hexadecimal exponent outside the range 0-127 is produced.
10. Exponentiation is carried out by repeated multiplication. The program is monitored immediately 'ILLEGAL EXPONENT' if the exponent n lies outside the range -255<=n<=255. If n is negative, x**n is evaluated as 1/x**n.
11. 0**0 gives the result 1 (as, of course, does any other quantity raised to the power zero).
12. An attempt to divide by zero causes the program to be monitored, 'DIVIDE ERROR'.

AE3 STRING EXPRESSIONS

COMPILE TIME

1. All names and constants are checked to be of type string.
2. Bracketed sub expressions are not allowed.
3. Only the concatenation operator '.' is permitted.

RUN TIME

4. The concatenations are performed from left to right but the program is monitored immediately if the resultant string exceeds 255 characters.

AE4 STRING RESOLUTIONS

COMPILE TIME

1. All names must be of type string and all the resolution expressions must be enclosed in brackets and satisfy all the conditions for string expressions (q.v.).

RUN TIME

2. In the simple resolution

<name1> -> <name2>.(<exprn>).<name3>

the following sequence of events occur:

<exprn> is evaluated as a string expression.
<name1> is searched to find this expression and the program is monitored if it cannot be found.
The part of <name1> before the resolution expression is assigned to <name2> as if by <assop> =.
The part of <name2> after the resolution expression is assigned to <name3> as if by <assop> =.
Either or both of <name2> and <name3> can be set as a null string by this operation.
The program will be monitored if either <name2> or <name3> is not long enough to hold their respective components.

3. The general resolution

<name1> -> <name2>.(<exprn>).<name3>.(<exprn2>).<name4> etc.,

is treated exactly as

<name1> -> <name2>.(<exprn1>).<private nameA>.
<private nameA> -> <name3>.(<exprn2>).<private nameB>
etc.,

where the private names are strings of length 255 declared by the compiler.

4. The resolution may be the subject of a condition. The condition is true if the resolution can be compiled and false if failure occurs. (N.B. the program will still be monitored as in 2 above where relevant). As a result of 3, some variables may be assigned if a multi-stage conditional resolution fails in the second or subsequent stage.

THE SEMANTICS OF CONDITIONAL PHRASES

<cond>

CP1 ARITHMETIC CONDITIONS

COMPILE TIME

1. Any constituent expression which contains
   a real variable, real function or real constant
   or
   either of the operators '/' or '**'
   is compiled as a real expression. All other expressions are compiled as integer expressions.

RUN TIME

2. <exprn><comparator><exprn>
   The left-hand expression is evaluated first, followed by the right-hand one. If one is an integer expression and the other real, the former is converted to floating point form before comparison. If one is %REAL and the other %LONGREAL then the former is stretched before comparison.
3. <exprn><comparator><exprn><comparator><exprn>
   The first two expressions are evaluated and compared. The third expression is only evaluated if the required condition between the first two is satisfied. Conversion is carried out if necessary as in 1.
4. <simple cond> %AND <rest of and cond>
   <simple cond> %OR <rest of or cond>
   The conditions are tested from left to right, stopping as soon as sufficient information is obtained to give the overall verdict true or false.

CP2 STRING CONDITIONS

COMPILE TIME

1. A check is made that all operands are of type string and that double sided conditions do not include resolutions.
2. The effect of using '<' and '>' comparators is to perform lexicographic comparisons based on the ISO character set and its internal codes. Thus WHISKEY > WATER is true.

RUN TIME

3. Expressions and conditions are evaluated and tested from left to right as for arithmetic conditions.
4. A resolution is treated as true if it can be completed and false otherwise.
5. Note that variables can be assigned by a compound resolution condition that subsequently fails (see AE4).

SECTION 3 - FAULT LISTS

Compile Time Faults

1. Too many %REPEATs
2. Label set twice or out of range
3. %SPEC faulty
4. Switch vector not declared
5. Switch label error
6. Switch label set twice
7. Name set twice
8. Too many parameters in Routine type declaration
9. Parameter fault in Routine type declaration
10. Too few parameters in Routine type declaration
11. Label not set
12. Type general parameter misused
13. %REPEAT missing
14. Too many %ENDs
15. Too few %ENDs
16. Name not set
17. Not a routine name
18. Switch vector error
19. Wrong number of parameters or subscripts
20. Switch vector or record format name in expression
21. Routine type or record name not yet specified
22. Actual parameter fault
23. Routine name in expression
24. Real quantity in integer expression
25. Cycle variable not integer type
26. %FAULT statement not at basic textual level
27. (Unassigned)
28. Routine body not described
29. LHS not a destination or name is not an address
30. %RETURN out of context
31. %RESULT out of context
32. (Machine code fault)
33. (Machine code fault)
34. Textual level > 8
35. Routine level > 5
36. Attempt to trap an untrappable fault
37. Array has too many dimensions
38. Constant overflow
39. Real quantity as exponent
40. Declarations misplaced
41. (Unassigned)
42. String variable in arithmetic expression
43. Bound pair inside out
44. Const error
45. Own array error
46. Attempt to initialise an extrinsic
47. Dangling %ELSE
48. Substitute character in program
49. (Unassigned)
50. Incorrect use of %DEFINE
51. Spurious %FINISH
52. Missing %REPEAT inside %START-%FINISH block
53. %FINISH missing

54. %EXIT out of context
55. %EXTERNALROUTINE in program
56. %ENDOFFILE out of context
57. Level 0 used illegally
58. (Machine code status fault)
59. (Machine code status fault)
60. (Unassigned)
61. (Unassigned)
62. Format wrong
63. %RECORDSPEC in error
64. Subname omitted
65. Wrong subname
66. Faulty record assignment
67. Invalid LHS of equivalence statement
68. (Unassigned)
69. Subname out of context
70. Invalid length in string declaration
71. String expression contains a variable
72. String expression contains invalid operator
73. Resolution comparator out of context
74. Resolution format incorrect
75. String expression contains subexpression
76. (Unassigned)
77. (Unassigned)
78. (Unassigned)
79. (Unassigned)
80. (Unassigned)
81. Variable equivalenced to expression
82. LHS not an address
83. Equivalenced operands not of same type
84. RECORD or ARRAY misused
85. (Unassigned)
86. (Unassigned
87. (Unassigned)
88. (Unassigned)
89. (Unassigned)
90. (Unassigned)
91. (Unassigned)
92. (Unassigned)
93. (Unassigned)
94. (Unassigned)
95. (Unassigned)
96. (Unassigned)
97. (Unassigned)
98. Addressability limit exceeded
99. No base register cover

Catastrophic Compile Time Faults

101. Source line too long
102. Analysis record too long
103. Dictionary overflow
104. Too many names
105. Too many levels
106. String constant > 255 symbols
107. ASL Empty
108. End Message character in program

A Fault number greater than 200 indicates a compiler error.

Trappable Run Time Faults

1. INTEGER OVERFLOW
2. REAL OVERFLOW
3. INVALID CYCLE
4. NOT ENOUGH STORE
5. SQRT NEGATIVE
6. LOG NEGATIVE
7. SWITCH VARIABLE NOT SET
9. INPUT FILE ENDED
10. NON-INTEGER QUOTIENT
11. RESULT NOT SPECIFIED
14. SYMBOL IN DATA s
16. REAL INSTEAD OF INTEGER IN DATA
17. DIVIDE ERROR
18. SUBSTITUTE CHARACTER IN DATA
19. (GENERAL GRAPH PLOTTER FAULT)
21. ILLEGAL EXPONENT
22. TRIG FN INACCURATE
23. TAN TOO LARGE
24. EXP TOO LARGE
25. LIBRARY FN FAULT n
26. RESOLUTION FAILS
27. INTPT TOO LARGE
28. ARRAY INSIDE OUT
30. CAPACITY EXCEEDED
31. UNASSIGNED VARIABLE
32. ARRAY BOUND FAULT n

Non Trappable Run Time Faults

ADDRESS ERROR
CORRUPT DOPEVECTOR n
ILLEGAL OPCODE
OPERATOR TERMINATION
OUTPUT EXCEEDED
TIME EXCEEDED
UNEXPLAINED INTERRUPT