USING IMP

An informal introduction
Issue 1.1

Peter S. Robertson
Lattice Logic Ltd. 1986

This document is intended as an informal introduction to the IMP language for people with a general understanding of the ideas and concepts of programming. It introduces sample programs or program fragments and discusses various features of the language which have been used. Also included are comments on why things are done the way they are and the benefits and disadvantages of the choices which were made during the development of the language. For a more formal and detailed description of the language please refer to "The IMP Language".

Throughout the text the IMP language will be referred to simply as IMP.

Part of the philosophy of IMP is to provide convenience forms of the standard constructions which, if used with care, can greatly improve the readability of programs. These convenience forms need never be used as the standard forms will always work.

First of all, here is a trivial program which prints out:

Hello there.

```
      %begin                            {simple first program}
         Printstring("Hello there.")
         newline
      %end
      %end %of %file
```

It would be a good idea at this stage simply to get this program into your machine, exactly as written above, and attempt to compile and run it. This will give you some feeling for how easy (or more likely how difficult) it is to generate the necessary incantations to get the operating system to do anything.
If the compiler produces error messages then you have probably mistyped the program. If the program is exactly the same as given here then the fault lies outwith the scope of this document.

Even though this program is so simple, it illustrates many of the features which give IMP its 'flavour'.

In the example above the text between braces (the curly brackets { and }) is considered to be a comment and along with the braces will be ignored by the compiler. If the closing brace is omitted the compiler will assume that there should have been one immediately before the end of the line, so if you have a comment which extends over several lines, each line must start with an opening brace.
While it is considered good programming practice to include apposite comments, their use will be minimised in this document as they would probably distract the reader, especially as the programs will be described in the text for it is the program statements themselves and not the algorithms they implement which are of interest.

Next, some 'words' start with a % character and some do not. The reason for this is to divide the 'words' of a program into two totally distinct categories: KEYWORDS and IDENTIFIERS. KEYWORDS start with a % and 'belong' to the compiler, while IDENTIFIERS do not start with a % and 'belong' to the programmer. When writing in IMP, if an identifier is more easily understood with spaces in it, put them in.
For example, our original program could be rewritten:

```
%begin
   print string("Hello there.")
   new line
%end
%end %of %file
```

The % character in keywords is often thought of as underlining the keyword, because when writing programs on paper it is much faster to underline than to write a percent sign. The exact definition of the effect of percent is that it underlines everything following it stopping at the first character which is not a letter. Hence %end %of %file could equally well be written %endoffile.

There are absolutely no restrictions on what characters can appear (or not appear if they are invisible) inside quotes; if your editor or operating system will let you put characters in the program the compiler will accept them. The only problem might be the effect of these characters on output devices if listings are generated from the compiler. In particular there is no need for the obscure 'escape sequences' of some languages to include spaces, newlines, tabs or whatever into a program.
Some programmers may prefer to rewrite the program as:

```
%begin
   printstring("Hello there.
")
%end
%endoffile
```

The disadvantage of allowing newlines inside quotes is that if the closing quote is omitted the compiler will remain in text mode and suck in the rest of the program, eventually giving a fault such as 'String too long' or 'Input ended'.
In practice this is never much of a problem, especially as the compiler will mark line numbers in the listing file to show whenever it is still in text mode at the start of a line:

```
1   %begin
2      printstring("Hello there.
3"   ")
4   %end
5   %endoffile
```

IMP requires that statements are terminated by a newline or a semicolon; newlines are not ignored. The only time a semicolon is needed is if you want two or more statements on the same line:

```
      %begin
         printstring("Hello there.");  newline
      %end
      %end %of %file
```

If you want to break a statement over several lines each line break must be preceded by a hyphen (which is otherwise ignored), or must come after a comma or the keywords %AND or %OR.

```
      %begin
         print string   -
         ("Hello there.")
         newline
      %end
      %end -
      %offile
```

The question of where statements start and finish is one of the more obscure parts of IMP and does cause some difficulty for beginners, but this is learned as one becomes more familiar with the language.

Now for a program which is a little more adventurous:

```
      %begin                    {program to add two numbers}
         %integer First, Second, Sum
         read(first)            {input first number}
         read(second)           {input second number}
         sum = first+second
         write(sum, 0)          {output the sum}
         newline
      %end
      %endoffile
```

This program is made up of one <u>block</u> (the bit between %begin and %end).
Within the block is a declaration (%integer) and imperative statements (read ..., sum = ..., etc.). Within each block in a program declarations must come before imperative statements.

When a block is executed the declarations cause various objects (integers, reals etc.) to be created and given identifiers by means of which they can be referenced. When the block is left any such objects are destroyed and the identifiers loose their meaning.

The program above creates three objects which can hold integer (whole number) values and calls them: first, second, and sum. Such objects are commonly called <u>variables</u>. These variables may be given values and subsequently the values they have been given may be retrieved for furthur use. Note that if an attempt is made to take the value of a variable before any value has been given to it the program will signal an error (unassigned variable). This check that variables have been given values catches one of the most frequent programming bugs (at least in the experience of the author), but sadly very few languages bother with it. Some languages give every new variable the value zero (or its equivalent). This often lets programs stagger on and fail long after the point at which a particular variable should have been initialised, or worse, the program just produces a credible but wrong answer. The more common approach is for new variables to be left with whatever rubbish is lying about in the memory, giving rise to programs which run sometimes and fail at other times depending what was happening previously.

The statement:

```
        Sum = First + Second
```

is an <u>assignment</u> which computes the value of the expression on the right hand side of the equals sign and assigns that value to the variable on the left hand side of the equals sign. The expression can be as simple or as complex as you like but it must give a result which is the same type (sort of thing) as the final destination. For example as First and Second are integers their sum must be an integer, and hence may be assigned to the integer variable Sum.
However, the division operator (/) always gives an answer which is real (fractional) and hence the assignment:

```
        Sum = First / Second
```

would be faulted by the compiler as Sum cannot hold real values. If an integer result is required as the result of dividing two integer values, the integer division operator (//) must be used. This performs the division and then discards any remainder.

The definition of IMP gives and number of operators which may be used to form expressions (add, ,subtract, multiply, etc.) and also defines how the expressions are to be evaluated, hence A+B*C means 'multiply B by C and then add in A' but not 'add A to B and multiply the answer by C'. It is to be strongly recommended that wherever there might be the slightest confusion don't be clever; use brackets to make the meaning obvious, i.e. write A+(B*C) or (A+B)*C. A very common mistake is to write A/2*B intending to get A/(2*B) but in fact getting (A/2)*B.

READ, WRITE, and NEWLINE are examples of <u>routine calls</u>, and the things in brackets after READ and WRITE are <u>parameters</u>. NEWLINE has no parameters and so is not followed by any brackets (some languages would insist that the call be followed by empty brackets: NEWLINE(), perhaps consistent but definitely irritating).

Note the difference in the parameters in READ(FIRST) and WRITE(SUM, 0). Because READ inputs a value from outside the program and assigns it to the variable given as a parameter, FIRST in this case, it is not the value in FIRST which is passed to the routine but the object FIRST itself. Such a parameter is said to be passed by reference. Conversely, the parameters to WRITE (SUM and zero) represent the value to be output (SUM) and the minimum number of characters to be output (0), and hence it is the values which are important. These parameters are said to be passed by value. Whether parameters are passed by reference or by value depends on how the procedure was defined; this will be discussed later.

As the call to WRITE only requires the value of SUM as its first parameter, and that value just happens to be First+Second the program could be rewritten:

```
      %begin
         %integer First, Second
         read(first)
         read(second)
         write(first+second, 0)
         newline
      %end
      %endoffile
```

Now for a program which introduces conditional statements. It simply reads in two numbers and outputs the relation between them.

```
      %begin
         %integer X, Y
         Read(X);  Read(Y)

         %if X > Y %start
            Printstring("The first was larger")
         %finish %else %if Y > X %start
            Printstring("The second was larger")
         %finish %else %start
            Printstring("They were equal")
         %finish

         Newline
      %end
      %end %of %file
```

This subject causes the most difficulty for beginners, mainly because it is different from most other languages. The first point to note is that lines starting %if..... and %finish...... are all complete statements and must be terminated by a newline or a semicolon. Secondly the bits between %start and %finish may contain as many statements as you like, including more conditional statements. Thirdly, the %else %if clause may be repeated as often as you wish (including zero times).

Finally, if there is nothing to be done in the case when none of the previous conditions has been satisfied then the final %finish %else %start may be omitted altogether.

This is the most general form of conditional structure and will always work. The difficulty results from the fact that conditions occur so frequently and are usually so simple that use of the general form can be like cracking a nut with a sledgehammer. Consequently IMP provides convenience forms which, if used with care, can lead to more readable programs. It is the firmly-held opinion of the author that extra effort spent in organising and writing programs is well worthwhile; programs are usually only written once but read many times. Therefore all the emphasis should go in attempting to make the program readable and its logic clear. Demanding that the standard form be used everywhere does not help to make code more understandable.

The first simplification is for the trivial case where something simple is to be done if a condition is true. This could be written:

```
%if [condition] %start
   Do something simple
%finish
```

but a simpler and more readable variant is:

```
Do something simple %if [condition]
```

after all, that was almost the form of words used to describe the problem in the first place!

The second change again follows from everyday English usage. The effect of the condition may be inverted by changing the keyword %if into %unless. This should be used with care as while it can make code clearer, misuse can make them very unclear:

```
Average = Total/Number %unless Number = 0
```

is clear enough, but what about:

```
%unless %not 1 <= N <= 9 %or M # 23 %start    {??????}
```

The final form is just a way to remove some of the wood so that the trees can be seen. Any statement starting with the keyword %FINISH <u>and</u> ending with the keyword %START may be rewritten with <u>both</u> of those keywords omitted.

Hence the original program could be written in what we think is a more understandable form:

```
      %begin
         %integer X, Y
         Read(X); Read(Y)

         %if X > Y %start
            Printstring("The first is larger")
         %else %if Y > X
            Printstring("The second is larger")
         %else
            Printstring("They were equal")
         %finish

         Newline
      %end
      %endoffile
```

IMP does not see conditions (things like X > Y) as being 'expressions' which give a boolean value, consequently IMP does not have boolean variables. Instead, IMP sees conditions as questions, so instead of the statement:

```
      %if A = B ......
```

being thought of as 'does A=B have the value TRUE' it is thought of as 'is A equal to B'. This may seem a trivial point but it can have can have a major effect on the language.
For example, in IMP the statement:

```
   Do something special %if A = B %and C = D
```

means exactly what most people would understand by the English statement formed by removing the percent signs. Pascal, for example would insist on having brackets round the components (A=B), (C=D).

Trying to keep as close to common English usage does help to make a language more readable (within limits) but it can cause problems when English is ambiguous. For example, what is the exact meaning of 'Bring me an apple or a pear and an orange'. Many computer languages resolve the ambiguity of AND and OR by means of precedence rules which bear no relation to everyday usage: AND is done before OR, the analogy being that AND = MULTIPLY and OR = ADD. Rather than leave the possibility of getting this wrong (and complex conditions are difficult enough anyway) IMP resolves the ambiguity by refusing to accept both AND and OR in the same condition unless the meaning is made clear with brackets. Hence the fruity example would become:
'Bring me (an apple and a pear) or an orange' or
'Bring me an apple and (a pear or an orange)'.

The next program will take in a sequence of 'words' (character sequences delimited by spaces or newlines) and count them. The program stops when it finds the word "%file" and so it can use itself as input for test purposes.

```
%begin       {program to count words}
   %string(63) Word
   %integer Number of words

   Number of words = 0
   %cycle
      read(word)
      Number of words = Number of words+1
   %repeat %until Word = "%file"

   printstring("There were ")
   write(Number of words, 0)
   printstring(" words")
   newline
%end
%end %of %file
```

Before getting into the main ideas in this program it is worth discussing the routine READ in a little detail. You may have noticed that in this example READ is given a string variable as its parameter while the previous example gave it an integer variable. This looks suspiciously like the non-standard procedures which are commonly used to perform input/output operations. Non-standard means that (taking Pascal as an example) although READ and WRITE look like ordinary procedures because they can take parameters of almost any type they cannot be defined in Pascal, as all user-defined parameters must have a fixed type. In IMP this is not the case as there is a 'general type' reference parameter which will accept a variable of any type. The functions ADDR, SIZE OF and TYPE OF are available for making use of such parameters.

In the same way that %integer introduces variables which may hold integer values, %string introduces variables which may hold string values, where a string is just a sequence of up to 255 characters. Any string variable has three properties: the number of characters it currently contains (its LENGTH), the maximum number of characters is can contain (its MAXIMUM LENGTH), and the actual characters themselves. Whenever a string variable is declared the maximum length of the string must be specified (63 characters in this example), and must be an integer in the range 1 to 255 inclusive. The reason for this limitation is so that the length of a string can be held as a sort of invisible character at the start of the actual characters. While this is not guaranteed to be the way in which strings will be implemented, to the knowledge of the author no compiler handles them differently. The only place where the knowledge that a string of maximum length N will take up N+1 characters-worth of storage is when the function SIZE OF is used. SIZEOF(WORD) will return the value 64, which is one greater than the maximum length of the string.

A frequent gripe against IMP is 'strings limited to 255 characters are useless'. There is no question that strings of any length would be ideal, but in practice the limit is not often a problem, and IMP strings are a lot more powerful and convenient (and useful) than no strings at all or the highly restrictive strings offered by the more common languages.

The main point of the example is to introduce cycles (or loops). In general any sequence of statements may be repeated by enclosing them in the statements: %CYCLE and %REPEAT. Note that %CYCLE and %REPEAT are statements and must be terminated (by a newline or a semicolon). If no furthur action is taken the cycle will continue indefinitely, so some means must be provided for terminating the loop. IMP provides one general mechanism and three 'syntactic sugarings' of common cases. The general mechanism is to use the instruction %EXIT, execution of which causes the loop to be terminated and control to pass to the statements following the corresponding %REPEAT. %EXIT can only be used to terminate one loop at a time; it cannot take you out of nested loops in one go.
The three common cases are provided by adding conditional clauses to either the %CYCLE or the %REPEAT statement, and they are:

```
Simple form                      Expanded form

%while [condition] %cycle        %cycle
   .........                        %exit %unless [condition]
%repeat                             .........
                                 %repeat


%cycle                           %cycle
   .........                        .........
%repeat %until [condition]          %exit %if [condition]
                                 %repeat


%for V = F, B, L %cycle          Templ = B;  Temp2 = L
   .........                      V = F-Templ
%repeat                          %cycle
                                    %exit %if V = Temp2
                                    V = V+Templ
                                    .........
                                 %repeat
```

Apart from simplicity there are no reasons why you shouldn't just stick to the %exit form of loops in all cases if you find it easier. In fact if the loop stops in the middle there is little choice other than contorting the program to force it into a %while or %until form.

In a direct parallel to the %if statement, IMP provides convenience forms of the %while, %until, and %for loops when the 'body' of the loop is very simple. E.g.

```
Buy something %while Money left # 0
X = X*10 %until X > 100
Visit(Patient) %for Patient = First, 1, Last
```

Beware that %while always does the test before the action which may not be executed at all, whereas %until always does the test after performing the action at least once.

The next program counts the number of letters, digits, and other characters in a piece of text, but instead of detecting the end of the text by looking for a special data item, it just waits for the event 'Input Ended' to be signalled instead.

```
%begin      {program to count letters and digits}
   %constant %integer Input Ended = 9
   %integer Sym, Letters, Digits, Others

   %on %event Input Ended %start
      Printstring("There were ")
      Write(Letters, 0);  Printstring(" letters, ")
      Write(Digits, 0);   Printstring(" digits, and ")
      Write(Others, 0);   Printstring(" other characters")
      Newline
      %stop
   %finish

   Letters = 0;  Digits = 0;  Others = 0
   %cycle
      Readsymbol(Sym)
      %if  'A' <= Sym <= 'Z'  %or  'a' <= Sym <= 'z'  %start
         Letters = Letters+1
      %else %if '0' <= Sym <= '9'
         Digits = Digits+1
      %else
         Others = Others+1
      %finish
   %repeat
%end
%endoffile
```

The statement %on %event Input Ended %start can be thought of as a sort of condition which is never satisfied. This means that when the %on statement is reached control will always pass by the bit between %start and %finish (the 'Event block') and carry on from after the %finish (or the %else if one is specified). However, if during the execution of the rest of the block the event (or events) mentioned in the %on statement is signalled, control is immediately passed to the first statement of the event block and execution continues from there. Any block may contain an %on statement, but there may only be one in a block and it must come immediately after the declarations (if there are any). In general it is not possible to resume execution from the point at which the event was signalled.

The exact definition of this event mechanism is a little more complicated but this description should be enough for the time being.

This program demonstrates the use of the single quote to provide the internal value (an integer) corresponding to any character. For example in the ASCII character set the constant 'A' is indistinguishable from the constant 65. Again, there are no limitations on the characters which can be placed between single quotes, so that ' ' is the value of a space (32 in ASCII) and '
' is the value of a newline (10 in ASCII). Because the newline constant can make it awkward to read a program, the named constant NL is available as a substitute.

The three tests of SYM are of interest as they are examples of 'double-sided conditions'. Effectively 'A' <= Sym <= 'Z' is an abbreviation for the condition: 'A' <= Sym %and Sym <= 'Z'. This sort of condition is very useful for testing for ranges of values as in this example. Note that the example assumes that there are no 'holes' in the character set and that the letters and digits are in sequence (beware of EBCDIC!).

In an earlier example the standard routine READ was used to input 'words' which were then counted. This program made the assumption that no word contained more than 63 characters. However, the implementation-provided routine READ cannot know this and will attempt to input words of any length. If it encounters a word of 64 characters or more it will quite happily take it in and then fail with 'string overflow' when trying to assign the value to its parameter.

The next program overcomes this problem by redefining the routine READ so that it simply truncates any words with too many characters.

```
      %begin     {program to count words revisited}
         %string(63) Word
         %integer Number of words = 0

         %routine Read(%string(63)%name Text)
            %integer Sym
            Text = ""
            %cycle
               readsymbol(Sym)
               %return %if Sym = ' ' %or Sym = NL
               Text = Text . To String(Sym) %unless Length(Text) = 63
            %repeat
         %end

         %on %event 9 %start                 {end of input}
            Printstring("There were ")
            Write(Number of words, 0)
            Printstring(" words")
            Newline
         %else                               {first entry comes here}
            %cycle
               Read(word)
               Number of words = Number of words+1
            %repeat
         %finish
      %end
      %endoffile
```

The definition of the routine READ is a <u>block</u>, nested within the %begin-%end block of the main program. Whenever an identifier is declared in a block it remains available for use throughout the rest of that block and any blocks which are subsequently nested within it. The only exception to this is the case of labels which are visible in the block in which they were defined but are not visible in any nested blocks. Labels will be discussed in a later section. If an identifier is redeclared within a nested block, that definition effectively masks out access to the outer definition. The result of all this is that in the sample program references to READ will access the newly-defined procedure and not the standard procedure, which along with all standard procedures is defined in a sort of 'super block' which contains the whole program.

In brackets following the procedure identifier, READ, is a list of declarations which define the number and type of the parameters which must be given at each call. The %NAME suffix specifies that the parameter is to be passed by reference, that is, in this example any reference to TEXT within the procedure will be exactly equivalent to a reference to the string variable actually given in the call.

Execution of the routine is terminated whenever the instruction %RETURN is executed. For convenience the %END of a routine is considered to be an abbreviation for %RETURN; %END. Note that this is only true for <u>routines</u>, other types of procedure must have explicit terminating statements or the compiler will flag an error (RESULT missing).

Three new features of IMP are illustrated by the statement:

```
   Text = Text . To String(Sym) %unless Length(Text) = 63
```

The dot is the only operator available to form string expressions and indicates concatenation. Concatenation is simply the joining-together of the two string operands to give a new string. The number of characters in this new string will be the sum of the numbers of characters in the original strings, not the sum of their maximum lengths; magic padding characters are never inserted.
The identifier TO STRING refers to a standard function which takes as its parameter an integer expression which must give a value corresponding to a character value (commonly any value in the range 0 to 255 inclusive). The result is a string value of length one character, that character being the value of the parameter.
For example, the following two assignments both give the same value to the variable Text:

```
   Text = "H"
   Text = To String('H')
```

Once again, note the difference between "H" which is a <u>string</u> value and 'H' which is an <u>integer</u> value.

The identifier LENGTH refers to a standard function which takes as its parameter a reference to a string variable and returns as its result the number of characters currently contained in the string.

In passing it is worth commenting on the initial assignment to Text:

```
   Text = ""
```

The double-quotes contain no characters, that is, a string of length zero; this is termed a null string.

Before leaving the routine READ itself it should be clear that the definition as given will accept string variables as parameters only if they are defined to have a maximum length of exactly 63. In the program in question this is no limitation, but if the procedure were wanted in other contexts it would be useful if it could accept strings of any maximum length.

To specify this the maximum length of the parameter definition should be changed to a star:

```
%routine Read(%string(*)%name Text)
   %integer Sym
   Text = ""
   %cycle
      readsymbol(Sym)
      %return %if Sym = ' ' %or Sym = NL
      Text = Text . To String(Sym) -
              %unless Length(Text) = Size of(Text)-1
   %repeat
%end
```

Also the reference to 63 has been changed to Size of(Text)-1. 'Size of' returns the size of the storage allocated to the variable given as its parameter. Remember that the -1 is needed as there is a one character overhead in strings to hold the current length.

Finally, there are two minor points about the complete program. The first is the declaration:

```
%integer Number of Words = 0
```

which may loosely be thought of as a contraction of:

```
%integer Number of Words
Number of Words = 0
```

although strictly the initialisation is performed when the variable is created, but this is only significant when %OWN variable are concerned. %OWN variables will be discussed later.

The second is the use of an %ELSE clause with an %on %event statement. The %else clause is executed when control first reaches the %on statement, and is skipped if the %on clause is executed following the signalling of a suitable event.

The following program demonstrates the definition and use of a function. In addition it shows the method by which constants can be given identifiers by means of the %constant declaration. Such identifiers may be used wherever their value may be given as a literal constant. This provides a convenient way of parameterising a program so that it is easier to read and change.

The example also uses the CHAR NO map which returns a reference to the N'th character in the given string variable. An error will be signalled if Char No attempts to access characters which are not within the string, i.e. if N is outwith the range 1 to Length(First parameter) inclusive.

```
      %begin      {program to test capitalisation function}
         %routine Read Line(%string(*)%name Line)
            %integer Sym
            Line = ""
            %cycle
               Readsymbol(Sym)
               %exit %if Sym = NL
               Line = Line.Tostring(Sym)
            %repeat
         %end

         %string(127)%function Capital form of(%string(127) Who)
            %constant %integer  Shift = 'A'-'a'
            %integer Up, N, Sym

            %for N = 1, 1, Length(Who) %cycle
               Sym = Char No(Who, N)
               Sym = Sym-Shift %if 'A' <= Sym <= 'Z'  {to lower case}
               %if 'a' <= Sym <= 'z' %start
                  %if Up # 0 %start
                     Sym = Sym+Shift                  {to upper case}
                     Up = 0
                  %finish
               %else
                  Up = 1
               %finish
               Char No(Who, N) = Sym
            %repeat
            %result = Who
         %end
         %string(80) Line

         %cycle
            Read Line(Line);  %exit %if Line = ""
            Printstring(Capital form of(Line))
            Newline
         %repeat
      %end
      %endoffile
```

This time the declaration of the parameter for the procedure does not end with %NAME and so the parameter is passed by value, that is when the procedure is called an ordinary string variable (Who) is created and it is assigned the value of the string expression given as the parameter to the call. The use of a star for the maximum length of a string value parameter is not permitted.

In the same way that %RETURN terminates the execution of a %ROUTINE, the instruction %RESULT=..... terminates the execution of a %FUNCTION. The expression following the equals sign must produce a value of the same type as the function (a string value in this case).

A common feature of languages which have been influenced by FORTRAN or ALGOL 60 is that they return the result of functions by means of a rather unpleasant pun on the function name. The mechanism is roughly that the use of the function identifier on the left hand side of an assignment specifies the 'result so far' but does not terminate the function. Eventually when the END of the function is reached the 'result so far' is returned as the actual result. IMP does not (indeed cannot) permit this as apart from the generally unpleasant nature of the pun and its associated difficulties, there would be no way of returning the result of a %MAP, as use of the map's identifier on the left of an assignment would quite naturally be seen as a recursive call on the map!

Within a function there may be as many %RESULT= statements as you wish; the execution of any one of the will immediately terminate the function.
For example, the following function returns the ordinal number of an upper-case letter, when that letter is from the EBCDIC character set. The difficulty is caused by the fact that EBCDIC has 'holes' between the letters; their values are not consecutive as they are in ASCII.

```
%integer %function Ebcdic letter number(%integer Sym)
   %constant %integer Ea = 16_C1,  Ei = 16_C9,
                      Ej = 16_D1,  Er = 16_D9,
                      Es = 16_E2,  Ez = 16_E9

   %result = Sym-Ea+1  %if Ea <= Sym <= Ei
   %result = Sym-Ej+9  %if Ej <= Sym <= Er
   %result = Sym-Es+18 %if Es <= Sym <= Ez
   %result = 0
%end
```

The equivalent function for ASCII would be:

```
%integer %function Ascii letter number(%integer Sym)
   %result = Sym - 'A' +1
%end
```

Note the form of the constants defining the EBCDIC values the the letters A,I,J,R,S,Z. The 16_ specifies that the following constant is expressed in base 16 (hexadecimal). In such constants the letters (upper or lower case) represent the 'digits' 10 (A), 11 (B), 12 (C) etc. The notation is quite general and any base greater than one can be specified. For example: Octal is 8_77715, Binary is 2_010101110 and base seventeen is 17_ABCDEFG. The notation may also be used for real constants. This is especially useful when the limit of accuracy is required as putting the constant into the base used by the machine can give more accuracy than expressing it in decimal.

For example PI could be defined as:

```
%Constant %long %real PI = 16_3.243F 6A89
```

Now for a program using real variables.

```
%begin     {reals}
   %real %function Compound interest(%real Capital, Rate,
                                       %integer Years)
      %real Balance
      %integer Years Left

      Balance = Capital
      Years Left = Years
      %while Years Left > 0 %cycle
         Years Left = Years Left-1
         Balance = Balance+(Balance*Rate/100)
      %repeat
      %result = Balance-Capital
   %end

   %real Cap, Rate
   %integer Time
   Read(Cap); Read(Rate); Read(Time)
   Printstring("The interest on £"); Print(Cap, 0, 2)
   Printstring(" at "); Print(Rate, 0, 2)
   Printstring("% per annum for")
   Write(Time, 0); Printstring(" years is £")
   Print(Compound interest(Cap, Rate, Time), 0, 2)
   Newline
%end
%endoffile
```

The function Compound interest takes three parameters, two real and one integer, all of which are passed by value. Again, this means that the function creates three variables and copies into them the values given in the call. Apart from this initialisation there are absolutely no differences between parameters and other variables declared in a procedure.
The program uses the two standard output routines WRITE and PRINT to generate its output. WRITE outputs an integer value using its second parameter to control the minimum number of characters output. PRINT outputs a real value using the second parameter to control the minimum size of the part before the decimal point, and its third parameter to control the actual number of places printed after the decimal point. if the third parameter of print is zero the decimal point and the fractional part of the number are not output.

So far, all the variables declared inside blocks have been destroyed when the execution of the blocks terminated. In several cases it is convenient for procedures to be able to exist in different 'states', that is to remember what they did last. This could be achieved by using variables declared outside the procedures (global variables) but then there would be no protection against other procedures altering those varaiables (perhaps as the result of a typing error). This is where %OWN variables are useful. An %OWN variable is identical to equivalent non-%OWN variables in every respect except that they effectively always exist, at least as far as the program which declares them is concerned. This means that they are not created and destroyed like other variables but exist throughout the execution of the whole program. As they are not destroyed when control passes from a procedure their values will be retained and will be available for use on subsequent calls of the procedure. However, access to the identifier of an %OWN variable is still limited to the block in which it was declared and to blocks subsequently defined within that block.

```
%constant %integer FF = 12
%routine Print and Suppress(%integer Sym)
   %own %integer Previous = FF
   %return %if Sym = FF %and Previous = FF
   Previous = Sym
   Printsymbol(Sym)
%end
```

This routine is intended for use in place of PRINTSYMBOL when the output is to be sent to a printer and consecutive form feeds (FF) are to be suppressed. The %OWN variable PREVIOUS is used to remember the character last output and to stop printing consecutive ones.

Unlike the initialisation of non-%OWN variables the statement:

```
%own %integer Previous = FF
```

does not mean:

```
%own %integer Previous
Previous = FF
```

as this would set PREVIOUS to FF every time the routine were called. Rather, the initialisation is performed when the variable PREVIOUS is created, which is effectively when the program containing the routine starts execution.

The basic data types provided by IMP are <u>integer</u>, <u>real</u>, and <u>string</u>. There are several ways in which these types may be used to create more complicated objects. The first of these is by means of <u>record</u> variables.

A <u>record</u> is a variable which is made up from a collection of other variables. The collection of other variables is described using a %RECORD
%FORMAT declaration. This defines the type, order, and identifiers of the components and gives an identifier, the format identifier, to the complete collection.

The format identifier can then be used to create objects with the internal structure described by the %FORMAT declaration:

```
      %record %format Person(%string(63) Surname, Prename,
                             %integer Age,
                             %real Weight)

      %record (Person) Fred
```

This defines the variable FRED to be a record containing two string variables, one integer and one real variable. To extract a particular variable from a record just follow the reference to the record by an underline followed by the identifier of the required component variable. For example:

```
         Fred_Prename = "Frederic"
         Fred_Surname = "Chopin"
         Printstring(Fred_Prename." ".Fred_Surname)
```

Records may be used in the same ways as the other sorts of variable discussed previously, in particular they may be passed as parameters, by value or be reference, and may be the results of record functions and record maps. The only operations available on records as complete entities are to copy one into another of the same format (Rec1=Rec2) or to set the complete record to zero (Rec=0).

```
%recordformat Coordinate(%real X, Y)
%real %function Distance between(%record(Coordinate) Point1, Point2)
   %external %real %function %spec Sqrt(%real Arg)
   %real Dx, Dy
   Dx = Point1_X - Point2_X
   Dy = Point1_Y - Point2_Y
   %result = Sqrt(Dx^2 + Dy^2)
%end
```

This program fragment defines a function which operates on points in a two-dimensional plane, and returns the distance between two of them. The parameters to DISTANCE BETWEEN are records passed by value. The user should take care with record value parameters as they require that the whole record be copied, not very expensive in this particular case, but with large records the overhead can be considerable. For this reason records are more commonly passed by reference, even though they will not be altered by the procedure. The function makes use of another function, SQRT, which returns the square root of its parameter. This function is not defined in this program but will exist at run-time in some other module. However, as all identifiers must be declared before they can be used, this information must be presented to the compiler. This is the function of the %external statement. The %external keyword tells the compiler that the identifier about to be declared is to be made available to the environment outside the program (commonly a linker of some sort). The %spec keyword tells the compiler that this declaration is a specification of an identifier which is not actually being defined by this

statement. In other words the complete declaration tells the compiler what sort of thing SQRT is (a longreal function with one longreal value parameter), and that it is defined somewhere in the external environment of the program. For the program containing the %spec to be able to run, there must be a module somewhere which defines SQRT.
This module could be written in IMP and the complete module would look something like:

```
      %external %longreal %function Sqrt(%longreal Arg)
         ..........
         ..........
         %result = ......
      %end
      %endoffile
```

This definition of Sqrt must match the specification statement used to reference it. In fact they are identical with the exception that the definition does not contain the keyword %spec.

The external mechanism is not just limited to procedures but may be used with variables. For example a module could define an external record into which various modules can place data:

```
      %record %format Things(%integer Number of washers,
                             %string(255) the saying of the day,
                             %real rate of inflation)

      %external %record(Things) Useful rubbish

      %endoffile
```

%external declarations like this also give variables the properties of %OWN variables. If a program or another module wishes to access this record it just declares it with an external specification:

```
   %begin
      %recordformat Things(%integer Number of washers,
                           %string(255) the saying of the day,
                           %real Rate of Inflation)

      %external %record(Things) %spec Useful rubbish

      Printstring(Useful rubbish_the saying of the day)
      newline

   %end
   %endoffile
```

Now you should be able to see that there is nothing magic about the routines NEWLINE, PRINTSTRING, WRITE etc. They are just external routines which the compiler automatically %SPECs for you. In effect the compiler starts each compilation by compiling a special file which contains statements like:

```
%external %routine      %spec Newline
%external %routine      %spec Write(%integer Value, Places)
%external %routine      %spec Print(%real V, %integer B, A)
%external %integer %fn %spec Length(%string(*)%name S)
%external %byte %map   %spec Char No(%string(*)%name S, %integer N)
.........
.........
%endoffile
```

Another way of creating more complex objects is to gather together a number of objects of the same type as an <u>array</u>.

```
      %begin            {counting letters}
         %integer %array Times('A':'Z')
         %integer Sym, J, N

         %on %event 9 %start
            %for J = 'A', 1, 'Z' %cycle
               N = Times(J)
               Printstring("There ")
               %if N = 1 %start
                  Printstring("was ")
               %else
                  Printstring("were ")
               %finish
               write(N, 0)
               Printsymbol(J);  Printstring("'s")
               newline
            %repeat
            %stop
         %finish

         Times(J) = 0 %for J = 'A', 1, 'Z'

         %cycle
            Readsymbol(Sym)
            %if 'A' <= Sym <= 'Z' %start
               Times(Sym) = Times(Sym)+1
            %else %if 'a' <= Sym <= 'z'
               Times(Sym-'a'+'A') = Times(Sym-'a'+'A') + 1
            %finish
         %repeat
      %end
      %endoffile
```

This program creates an array TIMES with 26 elements: TIMES('A'), TIMES('B') ..... TIMES('Z') and uses it to accumulate the number of times each letter (upper or lower case) appears in the input.

As a general point about efficiency, the expression -'a'+'A' has the value -32 but it is much clearer to write it in the form given rather than as the magic value -32. Perhaps even better would be to define a %constant %integer with the value -'a'+'A'. In all of these cases though, the compiler will generate the same machine code so there is nothing at all to be gained by calculating such constant expressions and obscuring the program with wierd and wonderful values. In the same way absolutely nothing is gained by using 65 instead of 'A' where that is meant, in fact legibility and perhaps character-set independence is lost by doing it.

Unlike Pascal and FORTRAN the bounds of arrays need not be constants; it is quite common for a program to calculate the size of arrays needed and then create them dynamically. The only restriction is that all arrays must have a non-negative number of elements. In other words the upper bound (the second one) minus the lower bound (the first one) plus one must not be negative.

```
      %integer %array A(1:0)    {is valid}
      %integer %array B(2:0)    {is not}
```

Consistently in the definition of IMP repetitions of zero or more times are always valid whereas negative repetitions are not. Does anyone know what it means to execute a loop -1 times? Do you do it backwards once?

The program also illustrates the use of a %for clause to initialise all of the elements of an array. This is a place where IMP is a little weak; it would not be difficult to permit simple operations on complete arrays as a direct parallel to the operations on complete records viz. copying and initialisation. This could be included as part of the continuing evolution of the language.

Records and array may be combined, that is you can have arrays of records and arrays within records, although any arrays inside record formats must have constant bounds and be one-dimensional.

The following program uses an array of records. It also uses a specification of a routine which is defined in the same block. This is to enable the routine to be used before is is actually defined. Whether you put routines first and then use them, or put specs first with the routines at the end of the program is purely a matter of taste; in general it has no effect on the efficiency of the program.

```
%begin
   %constant %integer Max Items = 100
   %recordformat Inf(%string(63) Word, %integer Occurred)
   %record(Inf)%array Item(1:Max Items+1)
   %string(63) Word
   %integer Items in = 0
   %constant %string(3) End Mark = "*E*"

   %routine %spec Add word to table

   %cycle
      Read(word)
      %exit %if Word = End Mark
      add word to table
   %repeat

   %for J = 1, 1, Items in %cycle
      printstring(Item(J)_Word)
      printstring(" occurred ")
      write(Item(J)_Occurred, 0)
      printstring(" time")
      printstring("s") %if Item(J)_Occurred # 1
      newline
   %repeat

   %routine Add word to table
      %integer P
      {insert the word provisionally}
      Items in = Items in+1
      Item(Items in)_Word     = Word
      Item(Items in)_Occurred = 0

      {now look for it}
      %for P = 1, 1, Items in %cycle
         %exit %if Item(P)_Word = Word
      %repeat
      {remove it if duplicated}
      Items in = Items in-1 %if P # Items in
      Item(P)_Occurred = Item(P)_Occurred+1
   %end
%end
%endoffile
```

It has been mentioned previously that parameters can be passed by reference, that is a reference to a variable can be assigned to the parameter rather than the actual value of that variable. Variable which can hold references to other variables are called 'pointer variables' and they may be declared and used like other variables.

For example, the procedure 'Add word to table' described above could be rewritten:

```
%routine Add word to table
   %integer P
   %record(Inf)%name New, Old
   {insert the word provisionally}
   Items in = Items in+1
   New == Item(Items in)
   New_Word     = Word
   New_Occurred = 0

   {now look for it}
   %for P = 1, 1, Items in %cycle
      Old == Item(P)
   %repeat %until Old_Word = Word
   {remove it if duplicated}
   Items in = Items in-1 %unless New == Old
   Old_Occurred = Old_Occurred+1
%end
```

The assignment Old == Item(P) assigns to Old a reference to Item(P). This can be thought of as making Old 'point at' Item(P). It is important to realise that it is the current value of P which is used in the sense that it after the == assignment P is altered Old will still point at the same element of the array Item.

Except when being used as the left hand side of an == assignment, use of a pointer variable is exactly equivalent to the use of the variable to which it is pointing. For example given:

```
%integer X = 0, Y = 1
%integer %name N
N == X
```

The assignment Y = N is equivalent to Y = X, N = 3 is equivalent to X = 3, M == N is equivalent to M == X, and Read(N) is equivalent to Read(X).

Now the mechanism of passing parameters can be understood a little more clearly. parameters passed by value are assigned using = and those passed by reference are assigned using ==.

Pointer variables may be used as components of records as the following program fragment shows. It is part of a program which manipulates lists of cells, with each cell using a pointer variable to point at the next one. The record NULL is a dummy record used to mark the end of the lists.

```
%begin
   %recordformat Cellfm(%integer Data, %record(Cellfm)%name Link)
   %record(Cellfm)%array Cells(1:Max Cells)
   %record(Cellfm) Null

   %record(Cellfm)%map New Cell
      %owninteger Last = 0
      %signal 14,1,Last %if Last = Max Cells  {none left}
      Last = Last+1
      %result == Cells(Last)
   %end

   %record(Cellfm)%map Copy of(%record(Cellfm)%name List)
      %record(Cellfm) Head
      %record(Cellfm)%name End, Cell
      End == Head
      %while List ## Null %cycle
         Cell == New Cell
         Cell_Data = List_Data
         End_Link == Cell
         End      == Cell
         List == List_Link
      %repeat
      End_Link == Null
      %result == Head_Link
   %end

   %record(Cellfm)%map Reversed copy of(%record(Cellfm)%name List)
      %record(Cellfm)%name New, Cell
      New == Null
      %while List ## Null %cycle
         Cell == New Cell
         Cell_Data = List_Data
         Cell_Link == New
         New       == Cell
         List == List_Link
      %repeat
      %result == New
   %end
   .......
```

The example also shows the use of %result == to give the result of a %map. This is an exact parallel to functions where %result = is used to return a value, while in a map %result == is used to return a reference. The compiler will report a fault if an attempt is made to use %result= in a map or %result== in a function.

The next to functions demonstrate the use of the standard function SUB STRING to split a string into fragments:

```
   %integer %function Index(%string(255) Data, Pattern)
      {returns the index of the first occurrence of Pattern
      {in the string Data. Zero is returned if the pattern
      {cannot be found}

      %integer Chars Left, Here, Limit, Len
      Len = Length(Pattern)
      Limit = Length(Data)-Len+1  {limit of search}
      Here = 0
      %while Here < Limit %cycle
         Here = Here+1
         %result = Here %if -
             Sub String(Data, Here, Here+Len-1) = Pattern
      %repeat
      %result = 0
   %end

   %routine Insert Today(%string(*)%name Line)
      %integer Pos
      %string(255) Before, After
      Pos = Index(Line, "*DATE*")
      %if Pos # 0 %start           {found}
         Before = Sub String(Line, 1, Pos-1)
         After  = Sub String(Line, Pos+6-1, Length(Line))
         Line = Before . Date . After
      %finish
   %end
```

SubString simply returns as its result the string made up of the sequence of characters between and including the characters at the positions specified by the second and third parameters. E.g. Substring("123456", 2, 4) = "234". If the third parameter is equal to the second a string of length 1 is returned, while is the third parameter is one less than the second a null string is returned.

Date is a standard string function which returns the date in the system-standard format. Similarly there is a function Time which does the same thing for the time of day.

As the operation programmed in the previous example is quite common IMP provides a unique instruction for doing it. This is termed 'string resolution' and looks like a backwards sort of assignment using ->.

```
      %routine Insert Today(%string(*)%name Line)
         %string(255) Before, After
         %if Line -> Before .("*DATE*"). After %start
            Line = Before . Date . After
         %finish
      %end
```

The string expression in brackets is evaluated and the string variable on the left (Line) is searched for that value. If the value is found the characters to the left are assigned to the variable to the left of the bracket (Before), and those on the right of the pattern are assigned to the variable on the right (After). A string resolution instruction has the strange but useful property that it can either be used on its own as an instruction, or, as in the example, it can be used as a condition. When so used the success of the resolution satisfies the condition and the implied assignments are carried out (so beware, this condition has a side-effect!). If the resolution fails, that is the pattern cannot be found, the condition is not satisfied and no assignments are performed. When a resolution is used as an instruction failure causes an event to be signalled (Resolution fails).

Either or both of the variables outside the brackets may be left out in which case the corresponding fragments of the original string are simply discarded. Hence the condition:

```
      %if S -> (Rude Word) %start
```

asks the question: 'does the string S contain within it the string contained in Rude Word?'.

Single-dimensional arrays of constants may be declared:

```
      %external %string (3) %function Month(%integer N)
         %constant %string (3) %array M(1:12) =
            "Jan", "Feb", "Mar", "Apr", "May", "Jun",
            "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"

         %result = M(N) %if 1 <= N <= 12
         %result = "???"
      %end
      %endoffile
```

Procedures may be textually nested inside other procedures:

```
 {1}  %routine Print Hex(%integer N, Width)
         %integer Places

         %routine Hex Digit(%integer D)
            %if D <= 9 %then Printsymbol(D+'0')     -
                       %else Printsymbol(D-10+'A')
         %end

         %return %if Width <= 0
         %for Places = (Width-1)*4, -4, 0 %cycle
            Hex Digit( (N>>Places)&15 )
         %repeat
      %end
 (2)  %routine Print to Base(%integer Base, N)
         %routine Print Digit(%integer D)
            %if D <= 9 %then Printsymbol(D+'0')    -
                       %else Printsymbol(D-10+'A')
         %end

         Print to Base(Base, N//Base) %unless N < Base
         Print Digit(Rem(N, Base))
      %end
```

The second example (Print to Base) demonstrates that procedures in IMP may be recursive, that is they may be defined in terms of themselves. This routine will output any positive integer to any base greater than 1, although the output will be a bit odd for bases greater than 36.