'THE ANALYSIS STAGE OF AN IMP-to-PL/1 TRANSLATOR'

by

JOHN P. WEBBY

A thesis submitted for the degree of Master of Science to the University of Glasgow.

September, 1973

ACKNOWLEDGEMENTS

The author is indebted to Professor T.R.F. Nonweiler of the Glasgow University Aeronautics Department for considerable help and guidance during the course of this work; and to Mr. Patterson of the Glasgow University Computing Department for criticisms made during the preparation of this thesis.

LIST OF CONTENTS

page

CHAPTER 1.

INTRODUCTION

1.1 BACKGROUND

In computing it sometimes happens that a person will want to use a program that is written in a language that is not implemented by the installation he uses. In such circumstances an additional program, a translator, which could translate the desired program into a language acceptable to the user's installation would be useful. However, because of essential differences between the two languages, it is possible that the translator will not be able to translate all of the original program.

This report describes the author's contribution to a project aimed towards the development of a translator that could systematically translate a program defined in the IMP language into one defined in PL/1; the implied intention being that the IMP source program and the translated PL/1 program will have in general the same effect upon their computational environment, when run under the control of an IBM system/360 operating system.

IMP is a language developed from ATLAS AUTOCODE. The particular implementation of IMP considered here is that supported by the Edinburgh Regional Computing Centre (hereafter referred to as the E.R.C.C.). To the author's knowledge, IMP is only supported at two other installations within Great Britain. In the United States Irons (Irons 70) has described his experiences with a language called IMP. Although this language has many of the general properties of E.R.C.C. IMP, it would appear to be defined in a substantially different manner.

The E.R.C.C. have written much of their software in IMP and it is hoped that, with the proving of this translator, this software will be of interest to other computing centres or programmers who have access to an IBM, PL/1 compiler.

The feasibility of the IMP-to-PL/1 translator was originally investigated by T. Nonweiler (Non 72) who concluded that certain features of IMP would not be translated because they were difficult to simulate in PL/1. The E.R.C.C. have written compilers for two forms of IMP, known as IMP(AA) and IMP(SYS); IMP(AA) being a subset of IMP(SYS). The translator accepts all programs written in IMP(AA) but may reject programs using the language extensions of IMP(SYS).

In addition Nonweiler drew up a set of rules by which an internal representation of a restricted form of IMP(SYS) called IMP(INT) could be translated directly into PL/1.

It remained the author's responsibility to write a program to translate IMP text into text defined in IMP(INT) form; the IMP text having already been passed through a lexical scanner (see Appendix 1). This involved a basically context-free syntax analysis followed by a context-sensitive syntax analysis and the simultaneous construction of symbol tables.

1.2 RELEVANT LITERATURE

For reasons that are given in the next section, it was decided that the translation from IMP-to-PL/1 should be controlled by an automatic syntax analyser.

Papers such as those by Mulholland (Mul 69), McEwan (McE 67) and the systems guide to the ALGOL-to-PL/1 translator (IBM 68a) describe language-to-language translators which are designed in a much less formal way; the translation being performed in a mostly ad-hoc manner. Therefore, these papers were of only superficial interest in the design of the translator considered here, though it was useful to compare the performance figures of the ALGOL-to-PL/1 translator with those of the IMP-to-PL/1 one.

It can be noted too that the IMP-to-PL/1 translator required more analysis of the source language (IMP) than any of those mentioned above.

A compiler is a particular type of translator; it translates a high-level language into the assembly language or machine language of a digital computer. Compilers are thus integral parts of most modern computing systems and there has been a considerable amount of literature not only describing particular compilers but also methods by which the writing of compilers and translators in general may be automated. It was this literature that was consulted mostly in the design of the IMP-to-PL/1 translator.

Gries and Feldman (Gries 68) have written a good review of efforts to automate the writing of a translator. Two books which have been useful in the writing of the IMP-to-PL/1 translator have been those by Gries (Gries 70) and Hopgood (Hop 69). Donovan and Ledgard (Don 67) have written a paper that descibes a method of tackling the translation problem that is fundamentally different to the one adopted here.

1.3 STRATEGY

The full translation from IMP-to-PL/1 has already been defined as a multi-pass process; both the lexical scan and the translation from IMP(INT) to PL/1 represent a full pass.

The IMP language being considered is still at a developement stage and the language specifications given in the current language manual (ERCC 70) are liable to change. It was thus felt that a formal approach to the IMP-to-PL/1 translation should be adopted so that any subsequent changes in the IMP language specifications could be easily incorporated in the translator. It was decided that, apart from the initial pass which made a lexical scan, the translation should be controlled by an automatic syntax analyser (IMP being a Chomsky type 2 language). This decision almost certainly made the translator easier to write and debug too, though it also probably slowed it down.

Of the two parsing strategies, or methods of syntax analysis, top-down and bottom-up, the latter is acknowledged to be the quicker. However, Capon and Argent (Cap 73) have shown that the bottom-up method called operator precedence is only a fraction faster than a top-down method. Also, the IMP-to-PL/1 translator will not be an integral part of a computing system as most compilers are, and so it's efficiency (that is it's ability to conserve both time and space during translation) was not such an important design criterion; more importance was placed on developement time.

So a top-down method of syntax analysis was chosen, using many of the techniques given in a report by Millard (Mill 66), because it was found easy to implement and quite powerful enough for the problem at hand. A full description of the method of syntax analysis is given in Chapter 2.

Other methods of analysis were not investigated with any thoroughness. It can be noted though that the ambiguity of the symbol '!' in IMP and the possible occurrence of implied multiplication, makes the meaning of expressions such as

!A!!B!

ambiguous. Thus, as it is defined in the language manual (ERCC 70), the grammar would have to undergo a possibly tedious transformation to bring it to a form suitable for either of the bottom-up parsing techniques, operator precedence or precedence.

For a translation to be affected, a requirement of the IMP source program is that it be error-free. That is, it must conform to the rules of syntax and semantics described in the IMP language manual (ERCC 70). In addition there are certain constructions not accepted by the translator. These unacceptable syntax constructions were defined by Nonweiler; they are listed in appendix (7). On discovering an error in the IMP source, processing is terminated immediately, as in the case of a context-sensitive syntax error, or at the end of the pass, as in the case of a context-free syntax error.

IMP does not allow the use of a variable before it's declaration. Thus a separate pass to build up the symbol tables is not necessary. The symbol table construction can be performed during the same pass as the generation of the IMP(INT) text. However, in order to check that the syntax of the IMP source is acceptable (in a context-free sense), a complete pass was introduced. This pass has the additional advantage of defining the syntax, or type, of each statement. This in turn reduces the amount the analyser needs to back-up in subsequent passes. In general back-up should be avoided, but particularly so if any context-sensitive processing is being done during the analysis. Back-up is described and discussed more fully in Chapter 2.

So the translation from IMP-to-PL/1 involves four passes in all. Figure 1 illustrates the translation of a particular IMP program 'f', using 'tombstones'. (This method of illustrating a translation is described by Earley and Sturgis (Ear 70).) Each pass is programmed in PL/1 (like the ALGOL-to-PL/1 translator). However, the translator will be made available to users in load-module form. Thus each pass is shown in Figure 1 as being written in ML, machine language.

The first pass is described in detail in Appendix 1. It reads in the IMP source program in card-image form and converts it into a form known here as IMP(SCAN) which is more easily processed by subsequent passes. In addition a dictionary of names, comment and constants is created which is referenced by the three other passes.

The second pass is described in Chapter 3. It performs a context-free syntax analysis of the scanned IMP text and in addition a few textual transformations. The processed text is called IMP(CHKD).

The third pass performs the translation to IMP(INT) form. This involves a context-sensitive analysis of the IMP program and the construction of symbol tables. This is often generally referred to as semantic analysis in this report, though it is acknowledged that it is possible to take exception to such a title. The third pass is described in Chapter 4.

The fourth pass performs the translation to PL/1 text and is described in Appendix 9. The PL/1 text may be put out in card-image form or alternatively compiled by the PL/1 preprocessor and compiler.

At the time of writing each pass represents a job-step on an IBM 370/155 machine and so at the end of each pass the processed text must be written to a disk file (organised sequentially and named IMPSCAN , IMPCHKD or IMPINT). It is hoped though, that this structure will be replaced by an overlay structure in the near future.

Each pass has associated with it a return code. In general, subsequent passes will not be attempted if the return code of a previous pass indicates that an error has been found in the IMP program or the PL/1 translator.

The design and construction of IMP(INT) is not directly related to the generation of a PL/1 translated text. It is believed that the analysis up to, and including the third pass may conceivably be of use in translations to other high-level languages. If any minor changes are needed in the second or third passes, they should usually be easily achieved because of the formal structure of these passes.

Finally it should be stated that an important consideration in the detailed design of this translator was that it should not be limited in any way by the size of the IMP source program .

CHAPTER 2.

A SYNTAX - DIRECTED METHOD OF TRANSLATION

2.1 INTRODUCTION

This chapter describes a method of systematically translating a given source string into a target string. The translation is performed in two stages

(i) a top-down syntax analysis of the source string, followed by

(ii) a generation of the target string based on information gathered in (i).

The method described is used in both the second, third and fourth passes of an IMP-to-PL/1 translation.

2.2 CONCEPTS OF TOP - DOWN SYNTAX ANALYSIS

In order to analyse a source string, a grammar must be defined. If this grammar is of a type called Chomsky Type 2 then it can be described by a set of syntax definitions written in Backus-Naur Form (BNF). This is the method adopted throughout this report (all grammars being of Chomsky Type 2), with the following exception; the symbol '|' in BNF is replaced by a new-line and '::='. Definitions numbered (2.1) to (2.8) below illustrate such a form.

```
<EXPRN>  ::= <TERM> * <EXPRN>          (2.1)   How
                                               extraordinary
         ::= <TERM>                    (2.2)
<TERM>   ::= <VAR> + <TERM>            (2.3)
         ::= <VAR>                     (2.4)
<VAR>    ::=  A                        (2.5)
```

::= B (2.6)

::= C (2.7)

::= D (2.8)

A,B,C,D,+,* are elements of the source language. They are called terminal characters. <EXPRN>, <VAR> and <TERM> are termed meta-variables. As can be seen above, to the left of each syntax definition is a meta-variable. This meta-variable is called the subject meta-variable. To the right is a string of terminal characters and/or meta-variables which may replace the subject meta-variable during a process called parsing or syntax analysis. This string may be called a derivation of the subject meta-variable. By applying another definition to a meta-variable within the string, a further derivation can be found. In such a way the original subject meta-variable should be ultimately 'reducible' to a string of terminal characters only.

Definitions (2.1) and (2.3) above are termed right-recursive because the subject meta-variable also appears as the right-most component of it's definition. Left-recursive definitions induce infinite loops in top-down syntax analysers and thus should be avoided. That is, the components of each definition should be arranged such that the subject does not also occur as the first component of it's definition. In addition, the definitions should not be reducible to a form where the subject occurs as the first component of it's definition.

The task of a syntax analyser is to find out whether the given source string is a derivation of the (predetermined) syntax definitions. All pure top-down analysers share a common strategy; they are goal-oriented.

The analyser takes as it's initial goal the starting point in a set of syntax definitions, also known as the root node or the unique non-terminal. In the syntax definitions above, <EXPRN> might be the root node. The analyser then examines successive components of the definition of this meta-variable. If a component is a meta-variable, then this meta-variable is set as a new sub-goal, the state of the analysis being recorded in a push-down stack. If the component is a terminal character, then a check is made to see if it matches the next character in the source string. If it does then the next component of the current definition is considered. If it does not then the analyser must look for an alternative definition to the current goal. For example, in the above definitions <VAR> ::= C is the next alternative definition to <VAR> ::= B .

The analyser continues in such a fashion until the definition of the initial goal has been satisfied (success) or until all alternatives to the initial goal have been tried without finding a match with the source string (failure).

When a syntax analyser looks for an alternative definition to a meta-variable, it is said to be backing-up or backtracking. Aho and Ullman (Aho 72) have described two types of backtracking; limited backtracking and full backtracking. The difference is that in a limited backtracking syntax analyser, once a meta-variable has been found which derives a prefix of the remaining source string (i.e. once a sub-goal has been found), no other alternative meta-variables (sub-goals) to the identified one will be tried, even if it is later found that the prefix was identified incorrectly. With full backtracking all possible alternative derivations of the initial goal are tried by the analyser.

For most computing language translators it should be possible to make a satisfactory syntax analysis of any source string without needing full backtracking capability. Indeed, it is often possible to arrange the syntax definitions so that even limited backtracking is not required. Such a situation is very much to be desired in translators for two reasons:

(a) The time taken by a syntax analyser with no backtracking capability to recognise a source string is directly proportional to the number of symbols in the source string. However, the the time taken by a syntax analyser to recognise a source string using backtracking is proportional to the number of symbols raised to the power n, where $n \geqslant 2$. This is proved by Aho and Ullman (Aho 72), for example. Thus a parse which involves no backtracking is inherently faster than one that does.

(b) In identifying a meta-variable, some semantic processing may have been done. If this meta-variable's definition is later rejected then the semantic processing must be undone.

It is often convenient during the syntax analysis to be able to perform some special, less formal, processing. This can be achieved by embedding in the syntax definitions, a signal to suspend the formal syntax analysis and call a special routine defined by the signal. For example, during the analysis of a declaration statement it might be convenient to insert an identifier into a symbol table. If these special routines can force the analyser into a specific course of action then the analysis may be made 'selective'. This is discussed further in the section (2.3.2).

2.3 TRANSLATION TECHNIQUES

2.3.1 Data Structures

The steps involved in analysing a source string may be represented by a syntax tree. For example, considering the syntax definitions numbered (2.1) to (2.8) in the previous section, the canonical derivation of the string A * B + C * D may be represented by the tree shown in diag. (2.1) on page 14.

A more concise representation of this syntax tree is obtained by replacing each node which is a meta-variable by the number of the definition which was satisfied during the analysis. If in this instance the prefix '2.' to each definition number is removed then the tree shown in diag. (2.2) on page 14 is obtained.

To define a translation, it is neccessary to associate with each syntax definition a string of characters representing the corresponding form of the target language.

Suppose that the string A * B + C * D is to be translated into a language in which W,X,Y,Z, represent A,B,C,D, respectively and the precedence of the arithmetic operations is defined by parentheses. Then this could be achieved by augmenting the definitions numbered (2.1) to (2.8) above with the target language constructs shown below

```
<EXPRN>  ::= <TERM> * <EXPRN>   {(@2*@1)}          (2.1)
         ::= <TERM>    {@1}                        (2.2)
<TERM>   ::= <VAR> + <TERM>  {(@2+@1)})}           (2.3)
```

(contd. on page 15)

Diagram (2.1)

A canonical derivation of the string A * B + C * D .

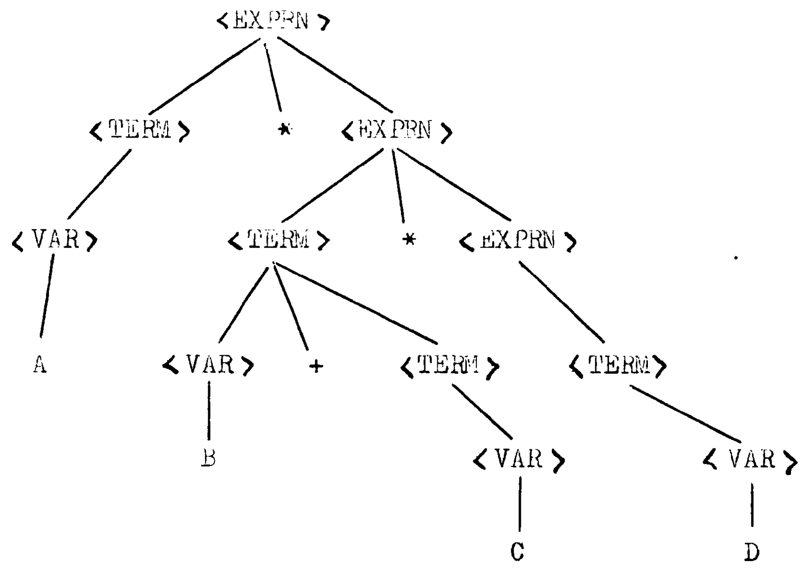
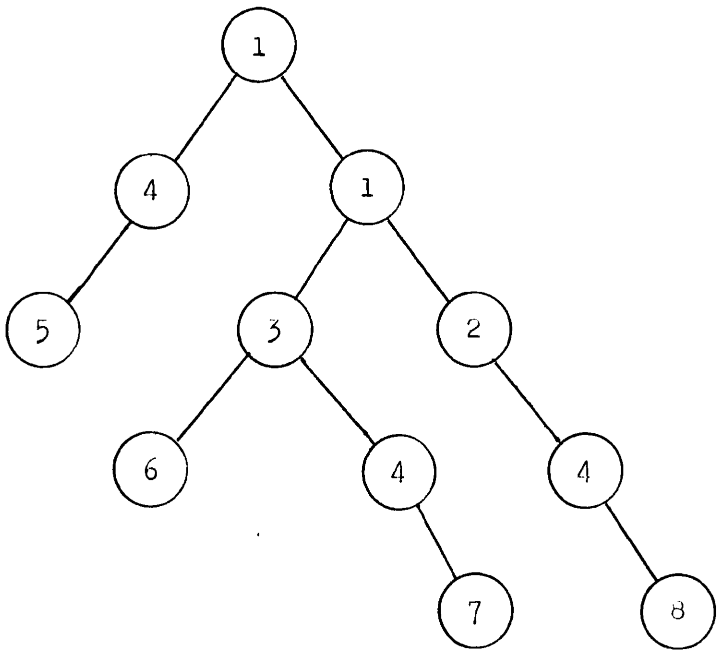


Diagram (2.2)

A more concise representation of the above syntax tree.



```
        ::= <VAR>  {@1}          (2.4)
<VAR>   ::=  A     {W}           (2.5)
        ::=  B     {X}           (2.6)
        ::=  C     {Y}           (2.7)
        ::=  D     {Z}           (2.8)
```

The target language language constructs are enclosed in the brackets { }. Characters in the constructs are one of two types; characters which have a special significance and terminal characters belonging to the target language.

In the above example there is one special character, '@' which is always followed by a digit. This instructs the generation routine on the way in which the syntax tree is to be read. The sequence of characters @n means process the node on the nth branch of the node being considered, numbered from the right. Considering the syntax definitions, this corresponds to an instruction to process the meta-variable numbered n from the right of the one being considered.

Whenever the generation routine encounters a terminal character, such as '(' above, the character is immediately put out as a member of the translation.

Thus the generation routine will interpret the syntax tree of diag. (2.2) in the following way. The root node is considered; this points to the target construct associated with definition (2.1), that is (@2*@1). The generation routine will process this in the following way.

(1) Put out the terminal character (.

(2) Process the node pointed to by the instruction @2, which is the construct of definition (2.4).

(3) Put out *.

(4) Process the node pointed to by the instruction @1, which is the construct of definition (2.1). This involves a recursive use of the construct.

(5) Put out).

The translated string is then ((W*X)+(Y*Z)).

2.3.2 Storage Structures

In this section one way of storing and accessing a syntax tree in a computing environment is described.

A syntax tree like that described by diag. (2.2) can be mapped onto a single dimension array, called LIST say, by the following set of rules.

(1) Consider the root node, i.e. the head of the syntax tree.

(2) If any branch of the node being considered does not have a terminal node then proceed to rule (3); otherwise add the contents of the node to the next location in LIST, followed by the contents of it's branching nodes copied from right to left. Replace the node by a terminal node containing a link to it's position in LIST. This link is an index to LINK prefixed by a minus sign. If the node was the root node then stop; else proceed to rule (1).

(3) Consider the first branching node (left to right) that is not a terminal node. Proceed to rule (2).

These rules are easily implemented during a top-down syntax analysis, being determined by the nature of the algorithm. The resulting 'threaded' list is easily stored in the core of a computer, and can be quickly searched by a generation routine.

Applying the rules to the tree of diag. (2.2) gives the array shown in diag. (2.3) on page 18.

The link to the whole list is -15. Thus the generation routine must inspect element LIST(15) first. This points to the construct associated with the first syntax definition (2.1) which is (@2*@1).

To process the node referred to by @2 is simple. The list is generated such that the contents of the node on the nth branch of a node N (numbered from the right) is indicated by the nth entry in LIST following that corresponding to the node N. The node referred to by @2 can thus be found by examining the entry 15+2 in LIST. This is a link to entry LIST(1). This entry points to the construct of definition (2.4).

A generation routine that will successfully operate on the list above to produce the target string ((W*X)+(Y*Z)) is described in section (2.4.3).

2.3.3 Analytic-Routines

As mentioned in section (2.2) it is often convenient during syntax analysis to be able to execute special routines. This is achieved by the translator considered here by embedding within the syntax definitions a signal which tells the analyser to suspend syntax analysis and call an analytic-routine, defined by the signal.

For example, during the analysis of a declaration statement an identifier may have to be inserted in a symbol table. This can be achieved by writing the syntax definition

```
<DECL> ::= <TYPE> <VAR> $12 ;    {@2@1;}
```

(contd. on page 19)

Diagram (2.3)

The syntax tree shown in diag. (2.2) in the form of a threaded list.

I	LIST (I)
1	4
2	5
3	4
4	7
5	3
6	-3
7	6
8	4
9	8
10	2
11	-8
12	1
13	-10
14	-5
15	1
16	-12
17	-1

When the analyser encounters the character $, it would call a special routine (analytic routine 12) which inserts the identifier derived from <VAR> in the symbol table.

After invocation of an analytic-routine, control can be returned to one of three points in the analyser. Let those points be labelled A,B and C.

If control is returned to label A, then the analysis continues in a normal fashion, i.e. the next component of the current definition is considered.

By returning to label B in the analyser, a redundant node is created in the syntax tree. The analytic-routine can be used to assign to that redundant node a parameter, available during the analysis, which is needed during the generation stage. (This parameter might also be communicated via a global storage location but this might create difficulties if the same routine was called several times during the analysis of a particular source string, since a second call on the routine might overwrite a parameter set in the previous call.) An example of the implementation of this technique is given in section (2.4.2).

If an analytic-routine returns control to label C in the analyser then the analyser is forced to reject the immediately preceeding meta-variable (the definition of which has been satisfied) and look for an alternative to it. This introduces a degree of selectivity into the analyser which may speed up the analysis. For example, suppose definitions (2.1) and (2.2) of section (2.2) were changed to

```
    <EXPRN> ::= <DUMMY> $10 <TERM> * <EXPRN>       {(@2*@1)}
```

```
        ::= <DUMMY> $10 <TERM>        {@1}
```

and the definition introduced

```
  <DUMMY>  ::=  <NULL>
```

where <NULL> is a special meta-variable representing the null string. Then analytic-routine 10 might inspect the next character in the source string to see if it is either A, B, C or D. If it is not then by returning to label C in the analyser both definitions are rejected and the analyser is saved the time spent in setting first <TERM> and then <VAR> as sub-goals. This technique is used extensively during the third pass to guide the analyser such that certain syntax definitions and thus certain target language constructs are processed.

The use of analytic routines is based on an idea by Millard (Mill 66); where they are called pi-routines.

2.3.4 Constructor-Routines

During the generation stage, as with the analysis stage, it is often convenient to be able to execute a special routine. This is achieved by embedding within a target language construct a signal to the generation routine to call a constructor-routine.

During the processing of the construct (@3@1?12) the string ?12 instructs the generation routine to call constructor-routine 12. It can be noted that the character ? which is the signal to the generation routine to execute a constructor-routine is always followed by a two-digit number defining the particular constructor-routine to be invoked.

Constructor-routines are used when it is more convenient to edit a target string by calling a special routine than by trying to write a suitable arrangement of syntax definitions and target language constructs. They are analogous to the phi-routines described by Millard (Mill 66).

2.4 THREE RELEVANT ALGORITHMS

In his report, Millard (Mill 66) described three algorithms.

(1) A Syntax Rules Processor to convert the syntax definitions from a readable form as shown in appendix (3) for example, to a form more suited to the subsequent analysis.

(2) An Analyser to perform a top-down syntax analysis of a source string and produce as output a 'threaded' list.

(3) A Generation Routine to operate on a threaded list and thus produce a translated string.

For the IMP-to-PL/1 translation both the syntax rules processor and the generation routine were considerably modified to fully utilise PL/1's string handling abilities. However, the same basic structure of these two algorithms was retained and since they are both quite simple algorithms they are only summarily described below.

Although not explicitly stated in his report, Millard's analyser does not incorporate any backtracking capability. Rather than trying to contrive a set of syntax definitions such that backtracking is never required, it was decided to write a new analyser for the IMP-to-PL/1 translator which had a limited backtracking capability and which retained the concept of analytic routines. This new analyser is described in section (2.4.2).

2.4.1 The Syntax Rules Processor

The purpose of this algorithm was to convert the syntax definitions from a BNF-like notation to tabular form. The definitions and their associated target constructions are mapped onto four arrays called SUB, COMP, ALT and OS. The process is described simply by diag. (2.4) which shows the mapping of the definitions

```
<EXPRN> ::= <TERM> * <EXPRN>   {(@2*@1)}
        ::= <TERM>   {@1}
```

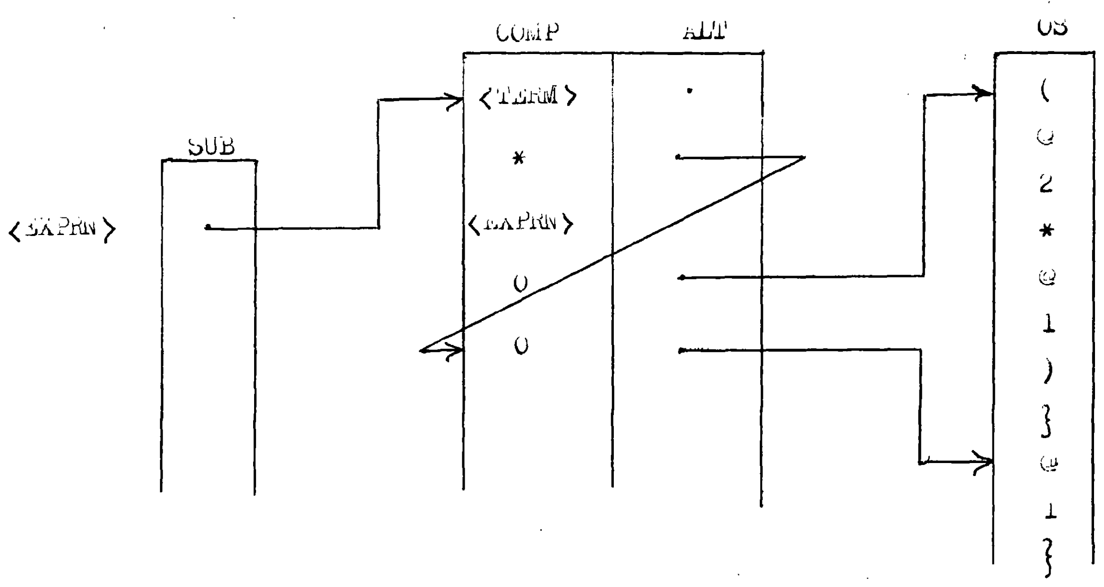
Before describing this mapping in detail, it can be noted that each component of a definition must be reduced to an integer so that it can index the arrays SUB, COMP and ALT. This integer must be unique for each different component. The process by which this is achieved is as follows.

IMP delimiters, identifiable by their initial character % are first reduced to a single (non-printing) character as defined by the lexical scan (see appendix (1)). The location in which this character is stored, which is of length 8 bits with OS/360, is then read as an unsigned integer and the resulting number assigned to that delimiter. In a similar way, a number is associated with a terminal character. Meta-variables are sent to a hash function which sets a switch if the meta-variable has been used before and returns an integer code to represent the meta-variable. This code is always greater than 255 to avoid confusion with terminal characters and IMP delimiters.

The mapping is achieved as follows. When the first component of a definition is encountered, which would be a subject meta-variable coded M say, a pointer to the first available location in COMP is put in SUB(M). The components are then entered in order in the array

Diagram (2.4)

Diagram to illustrate the syntax definitions in tabular form:



COMP followed by a zero to terminate the definition. The pointer to the next available location in OS is placed in the array ALT in the same position as the terminator in the array COMP. The target language constructs are mapped linearly onto the array OS. It can be noted that the terminating bracket } is included to signal to the generation routine that a construct has been fully processed.

If a further definition with the same subject is encountered then the components are compared with those of the previous definition until one is found which does not correspond to one previously stored, at location N say,in the array COMP. A pointer is inserted in ALT(N) to the next available location in COMP, from which point the remaining components of the new definition are stored.

If definition includes an analytic-routine call then the warning character $ is stored in COMP in integer form and the corresponding analytic-routine number in the parallel location in ALT.

For the IMP-to-PL/1 translator, the target language brackets {} are relaced by the character !.

The following restrictions are placed on the syntax definitions by the nature of the syntax rules processor and the analyser.

(1) The syntax definitions must be arranged so that there can never be an occurrence of left-recursion. It is possible to contrive a syntax rules processor which checks for such an occurrence but this was not attempted here.

(2) The meta-variable <NULL>,denoting the empty string, is represented in the array COMP by integer zero. Because of the structure of the analyser, if <NULL> is used it must be the only component of a definition. To introduce empty strings into compound

definitions, dummy meta-variables can be used. See for example the use of the meta-variable <DUM1> in definitions (75) and (295) in appendix (4).

(3) If the components of one definition correspond to the initial components of a longer definition with the same subject then the longer definition must preceed the shorter one.

(4) It is impossible to include an alternative to an analytic-routine in the array ALT, as ALT holds the analytic-routine number. This could be overcome by storing analytic-routines as a negative integer in COMP, but then the analyser would have to decide whether to invoke analytic-routines during backtracking. In the circumstances it is quite easy to engineer an alternative to an analytic-routine by using a dummy meta-variable. Definitions (181) and (182) in appendix (4) illustrate an example.

In general, once the syntax definitions have been correctly written out, the syntax rules processor is not a part of a translation process; the definitions being kept in tabular form on a disk or magnetic tape.

2.4.2 The Analyser

The new analyser is described by the flow diagrams of figures (2A) and (2B). Figure (2A) illustrates the analysis and figure (2B) the way the threaded list is built up.

Like all top-down analysers, this one uses a push-down stack. Each element of this stack has storage for five variables- ISTACK, JSTACK, SSTACK, TSTACK and PARAM (though PARAM is sometimes left undefined). For the analysis only ISTACK and SSTACK are essential, the others being used to build up the list.

Variable I points to the component in COMP currently being examined; J is used to order the list correctly; P and N are used as work variables recording values stored in the arrays COMP and ALT respectively; S points to the next character in the source string to be identified and T points to the next available location in LIST. The function INPUT returns that character in the source string indexed by it's arguement.

In diag. (2.2) on page 14 the contents of a node of the syntax tree was the number of a syntax definition. In this implementation the contents of each node is a pointer to the array OS, indicating to the generation routine the target language construct to be processed.

The calling routine initialises S, T and I. Suppose H is the integer code of the meta-variable which is to represent the root node or head of the resulting syntax tree. Then I is set to SUB(H).

Successive components of the definition of H are then examined by referencing array COMP (label A in figure 2A). If a component is a meta-variable (label D) then the state of the parse is recorded in the stack and the new meta-variable set as a sub-goal. When a terminal character is identified (label E), the state of the parse is again recorded so that backtracking is possible.

When a goal has been successfully identified (label F), the pointer to the corresponding target language construct is copied from the array ALT for subsequent inclusion in the array LIST. After processing associated with the build-up of the list is completed, the stack is popped-up (label H) and examination of the previous meta-variable is continued.

Processing is continued until the source string is recognised or until all definitions of H have been tried without a match occurring, when the analyser reports failure.

After invocation of an analytic-routine, control may be returned to either label A, B or C. These labels correspond to the labels mentioned in section (2.3.3).

The significance of label A has already been noted; it is the point where the next component of the current definition is about to be examined. Label B is the position the analyser reaches after a meta-variable's definition has been successfully identified. If an analytic-routine returns control to label B and in addition assigns a value to the variable N, then this value is stored in a unique position in the array LIST. As mentioned in section (2.3.2), this is a way in which a parameter may be passed to the generation stage. This technique is used extensively in the third pass to pass the code representing an IMP delimiter to the generation routine. Consider the following definitions

```
  < S2 >     ::= <TYPE> %NAME <PARAM1> <NAMELIST> ;    {@3?92@1;}
 <PARAM1>    ::= $1  {}
```

Internally %NAME is represented as a single non-printing character and so cannot be written out explicitly as a target language construct. After the analyser has recognised the terminal character %NAME, it examines the meta-variable <PARAM1>. This results in a call to analytic-routine 1 which assigns to the variable N in the analyser the code of the previous delimiter and returns to label B in the analyser. This has the effect of creating a node of the syntax tree corresponding to the meta-variable <PARAM1> but storing, not a pointer to the construct associated with <PARAM1> but the internal code of the delimiter %NAME. At a later stage this code is retrieved by the generation routine invoking constructor-routine 92 which examines the node corresponding to <PARAM1>.

Label C in the analyser is the point reached when a meta-variable's definition has been unsuccessfully applied and the analyser must backtrack to look for an alternative definition. The effect of returning to label C from an analytic-routine was explained in section (2.3.2).

Within the IMP-to-PL/1 translator, the analyser is declared recursive and is invoked recursively during the third pass.

2.4.3 The Generation Routine

The generation routine used by the IMP-to-PL/1 translator is identical to one described by Millard (Mill 66) with the following two exceptions

(a) For the PL/1 implementation the two variables REC and PARAM are made elements of a stack by using a PL/1 structure variable with the CONTROLLED attribute.

(b) The existence of a NEWLINE instruction in the target language constructs is not considered.

The routine operates on a syntax tree in the form of a threaded list of similar construction to the array LIST of section (2.3.2). By referencing the array OS which contains the strings of target language constructs, a translated string is generated.

The routine first accepts a link to the whole list. This indicates where in the list is the pointer to the construct associated with the root node. The routine then processes this construct in the following way.

If a member of the construct is a terminal character then it is put out immediately as a member of the translated string.

If a member of the construct is the character ? then the next two members are interpreted as an integer defining a constructor-routine that is to be executed.

If a member of the construct is the character @ then a new node is to be processed. Information about the old node is recorded in a push-down stack and the pointer to the construct for this new node found by the process described in section (2.3.2). When this construct has been fully processed, processing of the old node continues from the point where it was interrupted.

If a member of the construct is the character } then this signifies that the construct has been fully processed. The stack is examined to see if there are other nodes which have not been fully processed. If there are then the stack is popped-up and processing continues; if there are not then the translation has been completed.

CHAPTER 3.

THE SECOND PASS

3.1 PURPOSE OF THE SECOND PASS

The second pass was originally introduced into the translator design to check that the syntax of each IMP statement is acceptable to the third pass. IMP statements are considered syntactically in error if

(a) They do not conform to the syntax rules defined by the current IMP language manual (ERCC 70) or

(b) They contain IMP phrase structures that cannot be translated to PL/1 by this edition of the translator. These phrase structures are listed in appendix (7).

For reasons given in section (3.2.2), it was also decided to check during the second pass that the block structure, nesting and sequence of %CYCLE, %REPEAT statements (hereafter simply referred to as the block structure) is consistent.

The method employed to perform this checking is described in section (3.2). If any error is found during the second pass, subsequent passes are not usually attempted.

In addition a number of transformations of the IMP text are performed to facilitate further translation. These transformations are described in section (3.4).

3.2 STRATEGY OF THE SECOND PASS

To recognise unacceptable syntax constructions, the scanned IMP text is analysed a statement at a time by the syntax analyser described in chapter (2). The generation routine described in chapter (2) is then used to produce a target string.

Since the second pass performs few transformations of the IMP text, it was decided to use the analyser and generation routine to produce a statement-description code rather than translated IMP text. This code is either

(a) A two digit integer describing the type of statement analysed (e.g. assignment statement) or

(b) An error code in the form of a character string of length four (or possibly a multiple of four). This string is composed of the letter 'F' followed by a three digit integer. This integer is subsequently used to index a file of error messages.

Such an error code is generated if the corresponding IMP statement contains a syntax error or a phrase structure that cannot be translated to PL/1 or if the IMP statement does not produce a consistent block structure. It can be noted that instead of asking the generation routine to produce an error code and then indexing a file of error messages, the generation routine could generate the error message directly. In this instance though, the first method described above requires less storage and is quicker.

The method used to identify the above errors is described in sections (3.2.1) and (3.2.2) below. The syntax definitions used for the analysis of each IMP statement and the target language constructs used to generate the statement-description codes are shown in appendix (2).

The problem of performing the text transformations was resolved in the following way. Any text editing that needs to be done to an IMP statement is recorded in a push-down store during the analysis stage. Then when the analysis is completed an editing routine is invoked which operates on the push-down store and the IMP statement to produce the required translated statement. The editing routine that was devised to achieve this is described in section (3.4) together with a list of the transformations performed during the second pass.

3.2.1 Checking the Syntax

The syntax definitions which control the second pass are so prescribed that all IMP statements are recognised, no matter now wrongly structured they are. However, the target language constructs of those definitions describing an unacceptable phrase structure contain a four character error code. This code is subsequently generated indicating that the corresponding IMP statement is unacceptable.

3.2.2 Checking the Block Structure

IMP is a block-structured language. Like ALGOL or PL/1, entry to a new block, signified by the processing of a %BEGIN statement or a routine heading corresponds to an increase in the level of nomenclature.

It is important that the block structure of the input IMP program is correct and that at the end of the program the level of blocking is zero; otherwise the third pass cannot generate the required symbol tables correctly. Normally a user will have tested his program with the IMP compiler before submitting it for translation, and there will be no error in the block structure. However, to account for the possibility of say a lost card the block structure of the program is checked by the second pass.

As it is then relatively simple, the second pass also checks that the nesting (signified in IMP by the brackets %START ... %FINISH) and sequence of delimiters %CYCLE ... %REPEAT is consistent. That is, the field defined by one set of brackets does not intersect the field defined by another. For example, the program below is in error

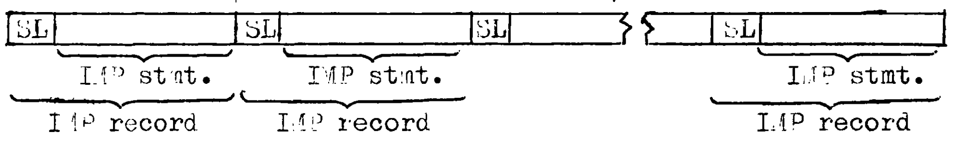
```
%BEGIN
  %CYCLE .....
    ...
%END
  %REPEAT
```

This check is performed in the following way. During the processing of an IMP program, each time an IMP statement is analysed which corresponds to the beginning of a new block an analytic-routine ($50 or $51) is called which makes a suitable entry in a push-down store called STAC say. Similarly, when a statement beginning with the delimiter %CYCLE or one containing %START is analysed, other analytic-routines ($52, $53 or $54) are invoked to make different entries in STAC. Whenever %END, %FINISH or %REPEAT is processed, analytic-routine ($57, $55 or $56) checks that the top element of STAC corresponds to %BEGIN (or a routine heading), %START or %CYCLE respectively. The stack STAC is then popped-up. If the correspondence is not made then the analytic-routine switches on a variable in a location global to the constructor-routines. During the subsequent generation stage, constructor-routine (?04, ?02 or ?03) examines this variable and if it is switched on then the constructor-routine writes a four character error code to the target string. No error-recovery technique is attempted other than leaving the stack STAC in it's previous state. Thus, it is possible for such an error to propagate through the rest of the program.

3.3 ORGANISATION OF THE SECOND PASS

Figure (3) illustrates the organisation of the second pass. Initially the main procedure reads the syntax definitions and target language constructs in tabular form from file SYNTAB2.

Next function READSS is called which reads file IMPSCAN and returns to the main procedure a copy of successive scanned IMP statements. File IMPSCAN, like files IMPCHKD and IMPINT which record the output from the second and third passes respectively, has the following structure



SL stands for a string-length indication. The IMP statement is in internal form.

Function READSS also breaks down compound conditional statements (see section (3.4) and appendix (2))which sometimes requires the construction of new labels. Thus the dictionary of names, comment and constants created during the first pass must be referenced. This dictionary is kept in a sequentially organised file named IMPDICT and is read into core by the first call on READSS.

Before calling the syntax analyser, the main procedure examines the first and last characters and if necessary the length of the scanned IMP statement in order to estimate the type of statement about to be analysed. This estimation is not as precise as the statement-description code generated when an acceptable IMP statement has been processed. However by making it, the subsequent analysis is speeded up. For example, if the first character in the statement is that corresponding to the delimiter %END, then the analyser is invoked with the meta-variable <S09> as a root node. If the meta-variable <SS> had been the root node , as it would have to have been had the estimation not been made, then the analyser would first have examined <S01>, <S02> etc. (see appendix (3)).

The analysis of an IMP statement is controlled by the syntax definitions shown in appendix (3). During the analysis, analytic-routines may be invoked to check the block structure of the program or to record information for any editing to be done. As already noted, an IMP statement is always recognised. The resulting syntax tree is written to a storage structure maintained by routine OUTPUT in the form of a threaded list.

Following the analysis, an editing routine is invoked which operates on information recorded in a stack by the analytic-routines, to perform any required textual transformation of the IMP statement. The processed statement is returned to the main procedure.

Next the generation routine is called with the set of target language constructs read from file SYNTAB2 as a parameter. This routine operates on the threaded list maintained by routine OUTPUT, possibly invoking constructor-routines in the process. The length of the generated target string is examined. This string is a statement-description code.

If the length of the target string is two, it indicates the statement was processed successfully. The string, which is a two digit integer defining the type of statement analysed is prefixed to the processed IMP statement. This record is then written to the file IMPCHKD together with a string-length indication.

If the string length is greater than two then an error has been found. The target string, which is then an error code is reduced to an index to the file DGNSTIC, the entry of which is a corresponding error message. This message is written out. The file DGNSTIC is declared in PL/1, INDEXED.

Processing is continued by calling READSS for the next statement. When the statement

```
                    %END %OFPROGRAM ;
                or  %END %OFFILE ;
```

has been fully processed the second pass is terminated; these statements signifying that the IMP program is finishing.

3.4 TEXT EDITING

In section (3.4.1) below is described the method of performing all the text transformations made by the second pass with the following exception. Compound conditional statements and statements beginning with the delimiter %UNLESS are reduced to the simpler form

```
          %IF <UBSC> %THEN %START ;
```

by an interface routine called READSS. This process is described further in appendix (2). Section (3.4.2) lists the particular transformations made by the second pass.

3.4.1 Method of Performing Text Transformations

Text transformations of the scanned IMP statements are performed, a statement at a time, by invoking analytic-routines during the analysis of a statement. These analytic-routines record on a stack information defining the editing to be done. Each element of the stack has storage for three variables; PTR and L (both integers) and INSERT (a variable length character string). PTR points to the position in the scanned IMP statement where the editing is to be done, L records the type (i.e. whether deletion, insertion or replacement) and extent of the editing and INSERT stores (if necessary) a character string to be inserted in the statement.

When the analysis of a particular statement is completed, an editing routine is invoked which performs the required transformations. This editing routine is described by the block diagram shown in figure (4). In this figure the string referred to is the scanned IMP statement and LENGTH is a function returning the character string length of it's arguement. The reference table mentioned is an array defining the correspondence between the position of a character in the scanned IMP statement and it's position in the processed statement. If an attempt is made to edit text that has been previously deleted, this correspondence is undefined (PTR=0). Such a situation should not arise during the simple editing required by the second pass.

3.4.2 Text Transformations made by the Second Pass

Below are listed the text transformations performed using the editing routine described above. As already noted, the reduction of compound conditional statements is made separately by an interface routine (appendix (2)). The transformations are best discussed by reference to the syntax definitions controlling the second pass which are shown in appendix (3). It can be noted by examining definitions (71), (265) and (269) that the translator accepts, and later appropriately translates to PL/1, resolutions of the form

%IF T -> S ('*') ;

though the current IMP compiler does not accept such a construction.

(1) <u>74 and 76</u> Isolated %START statements and their associated %FINISH statements are deleted.

(2) <u>159, 160 and 161</u> The prefix operator to arithmetic expressions is eliminated (if it exists) by

(a) Replacing the phrase \<OPERAND> by (0cccc - <OPERAND>) where the character string 0cccc represents the internal code of integer minus one.

(b) Replacing the phrase -<OPERAND> by (0dddd - <OPERAND>) where the character string 0dddd represents the internal code of integer zero.

(c) Deleting the prefix operator + .

The elimination of the prefix operator means that in the third pass the analysis of arithmetic expressions may be achieved using right-recursive syntax definitions which turns out to be very useful (see section (4.5)).

(3) <u>162 to 173</u> These definitions perform three tasks

(a) The null operator representing implied multiplication in IMP is specified explicitly by inserting the delimiter * .

(b) The logical operator 'exclusive or', represented in IMP by the symbols !! is replaced by the special delimiter %NEV.

(c) Modulus signs, represented in IMP by the symbol ! are replaced by the special delimiter %M.

Besides it's meaning described in (b) and (c) above, the symbol ! is also used in IMP to denote the logical 'inclusive or' operator.

The IMP language manual (ERCC 70) states that implied multiplication can only be used in expressions where there is no ambiguity. However, the phrase ! A ! ! B ! ! C ! for example is accepted by the current IMP compiler (release 7, version 11) but is ambiguous. It could be written out by the second pass as

```
            %M A %NEV B %NEV C %M
    or      %M A ! %M B %M ! C %M
    or      %M A %M * %M B %M * %M C %M
```

A short series of tests on the current IMP compiler has suggested a precedence concerning the symbol ! and the null operator

```
    !!                                  - most binding
 ! (inclusive or) and null operator
    ! (modulus)                         - least binding
```

and this is followed in the second pass. Thus the phrase above is translated as

```
            %M A %NEV B %NEV C %M
```

It is understood that in the near future a new delimiter is to be introduced by the E.R.C.C. to represent the modulus sign which will resolve the above ambiguity.

To achieve the necessay translation of the symbol ! , the second pass must perform a small amount of context-sensitive syntax analysis; the only time context-sensitive analysis is needed during the second pass.

(4) 110 The phrase %PRINTTEXT ⟨STRING⟩ is reinterpreted as PRINTSTRING (⟨STRING⟩). PRINTSTRING is an IMP library function.

(5) 120 and 121 The statement

%MONITOR %STOP ;

is reinterpreted as two separate statements.

(6) 206 The IMP text is transformed to remove namelists from formal parameter lists; the formal parameter delimiter being repeated as necessary.

(7) 209 The optional comma is removed.

(8) 229, 230 and 231 An operator prefixed to an integer signed constant is applied to the internal form of the constant and so eliminated.

(9) 265 Resolutions are converted into self-embedding productions by the introduction of a special delimiter, %RSLV.

(10) 264 The special delimiter %FRM is substituted for the symbols <-.

CHAPTER 4

THE THIRD PASS

4.1 INTRODUCTION

The task of the 3rd pass is to perform a context-sensitive syntax analysis of the IMP source program. The text is translated to a form called IMP(INT). IMP(INT) is an internal form of a subset of the IMP(SYS) language, augmented by markers (such as the statement type markers and the delimiter %RSLV introduced during the second pass). The syntax of IMP(INT) is defined by the rules shown in appendix (5).

To perform the context-sensitive analysis, loosely referred to as semantic analysis here, a knowledge of the attributes of the identifiers is required. These attributes are found in declarations and in the way the identifiers are used in the program, and they are collected and stored in a symbol table. The construction of the symbol table used by this translator and the technique used to create and access it is described in section (4.3).

The pass is successfully achieved by first analysing individual statements in the IMP source text and then generating IMP(INT) text using the techniques described in Chapter (2). Analytic-routines and constructor-routines are used to perform the necessary semantic analysis. The organisation of the third pass is described in greater detail in section (4.2).

One of the most difficult problems associated with the third pass was concerned with the 'typing' of expressions. The syntax of IMP(INT) requires that every general expression is prefixed by a 'type marker'. This problem is expanded on in section (4.4) together with the methods used to solve it.

A full description of all the text transformations made by the third pass is given in section (4.5). Included is a brief description of the individual methods used to affect these transformations.

4.2 ORGANISATION OF THE THIRD PASS

In general, the third pass is only attempted so long as the second pass has not discovered any syntax errors or any 'untranslatable' IMP phrases.

Figure (5) illustrates the organisation of the third pass. The translation of the text from the syntactically verified form to IMP(INT) is achieved using the syntax analyser and generation routine described in Chapter (2). The generation routine is used to directly generate the IMP(INT) text.

Initially the main procedure reads from file SYNTAB3 the syntax definitions used to control the analysis, and their associated target language constructions in tabular form. Next the main set of analytic-routines (denoted analytic-routines 3 in figure (5)) for the third pass are initialised. This involves reading from file LIBRTS the names of all intrinsic and implicitly specified routines of the IMP language library together with their attributes and loading this information into the symbol table. These library routines are treated as if they had been declared in the outermost block. In addition, the syntax definitions needed to type a conditional statement (see section (4.4)) are read from file SYNTABU in tabular form.

Next, a record of an IMP statement is read from file IMPCHKD. This record consists of a string length indication, the statement type code defined by the second pass, and a variable length character string representing the IMP statement. The appropriate root node (⟨S1⟩ , ⟨S2⟩... or ⟨S38⟩ in appendix (4)) is calculated from the statement type code and control is passed to the syntax analyser.

Each input IMP statement is syntactically analysed using the method described in Chapter 2. Analytic-routines may be invoked to perform any necessary semantic analysis. This might involve a reference to the symbol table or the construction of a new identifier (in which case file IMPDICT must be referenced). If a conditional statement is encountered then the analyser is invoked recursively and a second set of analytic-routines referenced (denoted analytic-routines B in figure (5)). This process is described in section (4.4).

The analyser builds up a syntax tree and this syntax tree is written to a storage structure maintained by routine OUTPUT.

When the input statement has been successfully passed, the generation routine is invoked, with the set of target language construction read from file SYNTAB3 as a parameter. This routine operates on the syntax tree stored in routine OUTPUT and generates a statement in IMP(INT) form. The generated statement will have prefixed to it, a statement type code, slightly different to that read from file SYNTAB3.

The IMP(INT) statement is written to file IMPINT together with its type code and a string length indication. Then the next IMP statement record is read from file IMPCHKD. The process continues until the statement

%ENDOFPROGRAM ;

or %ENDOFFILE ;

is processed, which denotes that the IMP source program is finishing. Then the symbol tables and their associated lists are written to the sequentially organised files IMPIDS and IMPIDSA.

If so instructed, the IMP(INT) text can be converted back to a subset of IMP(SYS) on completion of the third pass. This request is affected by the insertion of a parameter in the relevant job-control card. This involves the elimination of all the introduced markers and special delimiters and the reconversion of the internal coded form of the program to a card-image format. This output program should be capable of being compiled by the IMP compiler.

If a semantic error is discovered in the input text then no further processing is attempted. A message is printed out indicating the statement that the error occurred in and the cause of error. In general, no further translation of the IMP program will then be attempted.

The third pass uses at most 118 analytic-routines (plus a further 20 analytic-routines for the typing of conditional statements).

4.3 SYMBOL TABLES

As also described by Gries (Gries 70) a symbol table associates with each identifier used in a program a 'descriptor'. This descriptor stores, in coded form, all the information about the identifier necessary to perform a semantic analysis of the program.

All symbol tables have the general form shown in diagram (4.1) on page 46.

When dealing with block structured languages such as IMP, it is necessary to be able to differentiate within the symbol tables between identically spelt identifiers which are declared and used in different blocks. Such identifiers are given the same four-character dictionary key by the first pass.

This differentiation is made here by making the argument of the symbol table the identifier concatenated with an integer (in the form of a two-character string) defining the block in which the identifier was declared. This technique is also described by Gries (Gries 70) for example.

The convention is taken that the outermost block of the IMP program is numbered zero. Each block is then numbered in the order in which it is opened during a top-down parse of the program. In this way a unique integer, called the block sequence number say, is associated with every block. To convert the signed integer to a string suitable for concatenation with an identifier, the location the integer is stored in is overlaid on a two-character string. (This string is not usually 'printable'). The maximum block sequence number using this technique is 32,767 which is considered quite large enough.

The rule for finding the correct declaration corresponding to the use of an identifier is to look first in the current block (the one where the identifier is used), then the surrounding block, and so on until a declaration of that identifier is found. Each surrounding block is said to be at a lower level of nomenclature or blocking to the reference one. During the processing of an IMP program an array is kept recording the block sequence number at each active level of nomenclature at any particular time. This array is referenced whenever a declaration of an identifier is not found in the current block.

There are two distinct problems associated with symbol tables.

(1) the format of the descriptor and

(2) the organisation of the symbol tables.

These problems are discussed under separate headings below.

In addition to the main symbol tables, the third pass keeps a second type of table for semantic analysis. This is discussed in section (4.3.3) under the heading 'RFTABLE'.

Finally, the main symbol tables constructed during the third pass are also referenced by the fourth pass, which at the time of writing is a separate job step. Thus the symbol tables must be capable of being written to, and read from, the disc.

4.3.1 The Descriptor

4.3.1.1 Data Structures

The amount of information which a descriptor needs to store depends on what the associated identifier is; that is whether the identifier has the attribute simple variable or function or label etc. This can be accounted for by having varying length descriptors or allocating two or three say, successive symbol table elements for identifiers requiring extra space. A more popular technique, used by the ALGOL-to-PL/1 translator for example, is to have separate symbol tables for different types of identifiers. Yet another technique is to use part of the descriptor field as a pointer to a secondary table or list when extra space is required.

This last method was that adopted by the third pass. The descriptor field was made large enough to hold all the necessary information for all but two types of identifiers. Then a portion of the descriptor field was used as a pointer to a linked list describing

(contd. on page 47.)

Diagram (4.1)

The General form of a Symbol Table:

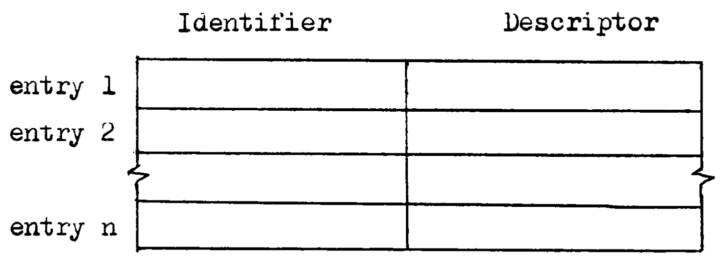


Diagram (4.2)

Diagram to show the Formation of the Lists pointed to by the Descriptors of IMP record variables and <RT> type variables:

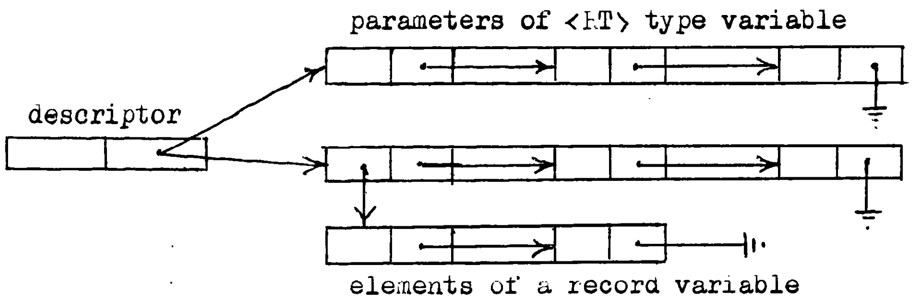
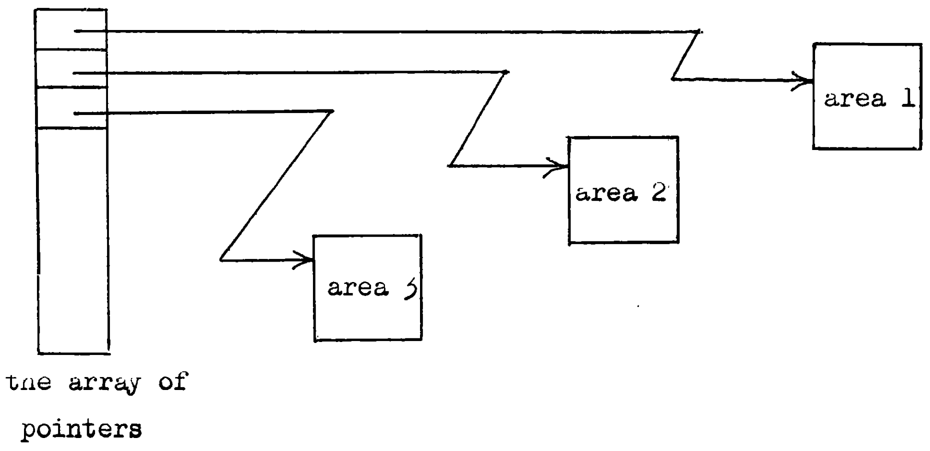

parameters of <RT> type variable
descriptor
elements of a record variable

Diagram (4.3)

Diagram to illustrate the method of indexing the areas containing the Lists:



the additional attributes. The two types of identifier using this structure were

(a) IMP record variables (these are data aggregates whose elements may be of different types). The linked list then describes the attributes and names of each element of the record.

(b) <RT> type variables i.e. routines, functions and maps. The linked list then described the parameters of such variables. The descriptor fields for such identifiers can thus be described by diag (4.2) on page 46.

From this diagram it can be noted that

(a) an element of an IMP record may point to another record

(b) the last element of a list is enclosed by a special entry in its pointer field. (it is said to be 'ancnored'.)

(c) identifiers other than records or routines have the pointer fields of their descriptors undefined.

<u>4.3.1.2 Storage Structure</u>

The descriptor which occupies 56 bits of storage has the following storage structure

1-3	4-11	12-14	15	16-17	18	19	20-24	25-56
TYPE	QUAL	DIM	AP	VP	USED	INTNC	ANO	AOFF

ANO and AOFF represent that part of the descriptor which points to a linked list. A problem in the format of this field was created by the fact that the symbol tables and the associated lists must be written to disk. This is discussed further below.

The meaning of the other sub-fields of the descriptor is given in appendix (8). Note that more information is stored than is strictly needed for the translation to PL/1. For instance, it is not necessary to store labels in the symbol tables. This additional information is recorded for a sense of 'completeness'.

The easiest way to build up a variable length linked list like those mentioned in the previous section is to use an absolute machine address as the pointer to each list element. This is easily implemented in PL/1 by the use of BASED variables.

However the value of such pointers becomes meaningless across input/output when different areas of storage for the list elements might be used. And the lists built up by the third pass must be written to, and read from, a disk file if they are to be referenced in the fourth pass (a different job step).

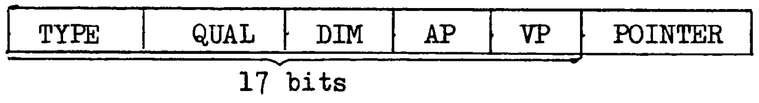
To overcome this problem, PL/1 allows lists to be built up in a specific area of storage, termed an AREA variable. The pointers used to qualify the elements of such lists (termed OFFSETS) are not absolute machine addresses but are addresses defined relative to the start of the area. During output the area of storage containing the lists is written to a disk file. When later input, this area containing the lists is read into another AREA variable, whose absolute address is known. The offset variables qualifying the elements of the lists can then be easily converted to absolute machine addresses by referencing the absolute address of the new AREA variable. Most of this 'housekeeping' is done by the PL/1 compiler.

During the creation of the lists, whenever an area becomes full, a new area is allocated and the construction of the lists continues in this new area. In this way, the translator is not limited in any way by the size of the IMP program. Thus each pointer contains two fields, an OFFSET variable and an index to the particular area referenced. This is in fact an index to an array of pointers. This is illustrated by diag. (4.3) on page 46.

Within the descriptor, the OFFSET variable and the index to the array of pointers are kept in their internal (bit) form.

N.B. It is an implementation restriction of the current PL/1 compiler that the declaration of AREA(*) based variables is not allowed. Thus when an area becomes full, the area cannot be reallocated with a larger field. Thus the above technique, using an array of pointers is necessary.

Each element of a list which describes a parameter has the following structure.



The sub-fields, TYPE, QUAL, DIM, AP, VP have the same significance as the fields described in appendix (8). POINTER points to the next element of the list (or is 'anchored'). It is composed of an index to the array of pointers and an OFFSET variable as before but is not stored in a bit representation.

Each element of a list which describes an element of a record has the following structure

$$\boxed{\text{NAME}}\underbrace{\boxed{\text{DESCRIPTOR}}}_{\text{56 bits}}\boxed{\text{POINTER}}$$

NAME is the four-character key to the element name supplied by the first pass. The format of the descriptor field is identical to that of appendix (8). POINTER has the same construction as for elements of parameter lists.

The POINTER fields of the terminal entries in a list are 'anchored' using the PL/1 function NULLO.

4.3.2 ORGANISATION OF THE SYMBOL TABLE

Like most translators (including compilers), the third pass may reference the symbol table many times during the analysis of a single expression. Thus the organisation of the symbol table can significantly affect the speed with which a translation is affected.

Probably the most frequently used method of organising a symbol table is that based on hash addressing. Hash addressing associates a number with a character-string argument. For a symbol table, this argument is usually the identifier name.

In general, it will often occur that the same number is associated with different arguments. When such a situation arises, it is known as a collision. Hopgood (Hop 69) gives a good account of the various methods of resolving this problem, such as rehashing and chaining.

The argument of the symbol table constructed by the third pass is a six-character string composed of an identifier in the form of a four-character key to the dictionary constructed by the first pass, concatenated to a two character number defining the block in which it was declared.

The hash function adopted is similar to that used by the PL/1 compiler. The six character argument is overlaid into three double-byte integer stores which are then added together (with a no-overflow condition attached). The result is divided by 211; the remainder being the number associated with that string.

This number is the index to a 'hash table'. The elements of the hash table are not the descriptors associated with the hashed identifiers but pointers to the symbol table proper as illustrated by diag (4.4) on page 51.

An element of the symbol table was composed of three fields.

IDENTIFIER	DESCRIPTOR	CHAIN

CHAIN is a pointer field. It is the implementation of the chaining method of solving the 'collisions' problem. If an identifier is hashed to a location in the symbol table which has already been filled then it is put in the next free location. The address (index) of that location is then placed in CHAIN. If CHAIN was full too, then the location it pointed to is examined and the process repeated.

The data structure for this symbol table can be represented by diag. 4.5 on page 51. It can be noted than when one symbol table becomes full, another is allocated.

The data structure described above is implemented in the PL/1 program by using an array of structures to represent a symbol table and declaring it BASED. When this array becomes full, another array is allocated. The hash table is an array of 211 elements.

4.3.3 <u>RFTABLE</u>

In IMP a data structure is defined by a record format statement. Record type identifiers are then declared by reference to a previously specified record format. This reference consists simply of the record format name, thus

```
%RECORD %FORMAT STRUCT(%INTEGER I, %REAL A,B )
%RECORD R(STRUCT)
```

(contd. on page 52.)

Diagram (4.4)

Diagram showing method of hash addressing:

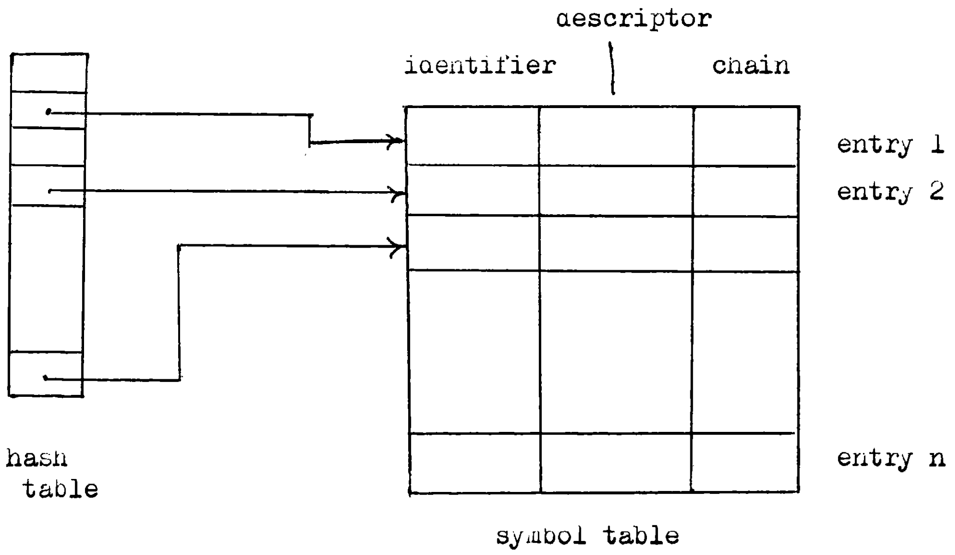
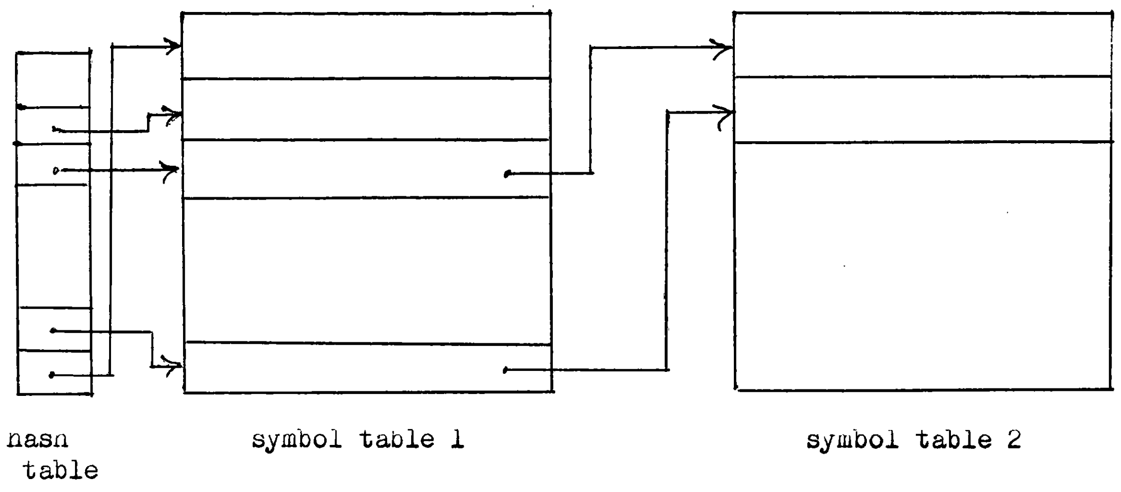


Diagram (4.5)

Data structure of symbol table showing method of Chaining:



The third pass keeps a special table which associates with each record format name, a pointer to the list describing the data structure. Concatenated with each record format name is a block sequence number, as described at the beginning of section (4.3). The organisation of this table is exactly the same as that of the symbol table. The descriptor consists of a pointer, ANO and AOFF.

$$\underbrace{\boxed{\text{ANO}}\boxed{\text{AOFF}}}_{\text{37 bits}}\underbrace{\boxed{\text{MARK}}}_{\text{1 bit}}$$

as before concatenated with a marker, MARK, that is set to '1'B only if the record format contains a record pointer. This information is needed by the fourth pass.

The third pass replaces all integer labels by translator-created names. RFTABLE is used to associate each integer label with its new name. The integer label is the argument and the descriptor is the associated generated name (four characters or thirty-two bits).

The integer label is overlaid on a four-character location to produce the desired character-string argument of RFTABLE. Since the maximum value of an integer label in IMP can only be 16383, the left-most two characters in this argument are always hexa-decimal 0000. The four character keys to the identifier dictionary constructed during the first pass are such that their left-most two characters are never hexa-decimal 0000. Thus there can never be any confusion between integer labels and record format names.

4.4 EXPRESSION TYPING

The specification of IMP(INT) requires all expressions, but those such as subscript lists which are contextually defined as integer, to be prefixed by a type marker (appendix (5): 336). In the case of a string expression this type marker also includes a length designation.

In IMP it is often impossible to define exactly the type of expression being examined until it has been completely parsed. Using techniques discussed in Chapter (2) it would not be difficult to insert a type marker at this stage, but there is a further complication.

The reactions of the IMP and PL/1 compilers to finding a constant in an arithmetic expression differ considerably. According to the current manual (IBM 70), the PL/1 compiler types a constant by appearance only. The action of the IMP compiler is not explicitly defined in the current manual (ERCC 70) but the author understands that all constants are initially placed in double length floating point (long real) locations and reduced to single length or integer form if the context so allows it. This difference could produce an error between an IMP source program and the PL/1 translated program unless special action is taken.

Consider the following IMP assignment statement

```
SRV  =  3.61728@1   * 4 .236@3  * LRV
```

where SRV represents a short real variable and LRV represents a long real variable.

The IMP compiler (because of the presence of LRV) performs the whole evaluation of the right-hand side double length, the two constants remaining in their double length floating point locations.

When processing a similar statement in PL/1, the PL/1 compiler assigns both constants to short real locations (FLOAT DEC (6) and FLOAT DEC (4) respectively) and performs the multiplication 3.67128@1 * 4.236 @3 in single length. This evaluation will not always be so accurate as that of the IMP compiler.

For this reason it was decided to replace all constants in expressions, except integer constants in integer expressions by generated names. A suitable declaration of the generated name, appropriately initialised is included at the head of the outer block.

These declarations are not inserted until the expression (or sub-expression) has been completely parsed. For in the example above, the constants would first have been recognised as short real constants since the left-hand-side is short real. The relevant syntax definitions (app.(4): 74, 192 and 193) are

```
LHS   : : = < REXPRN >
      : : = < LREXPRN >
```

Because of the presence of the variable LRV, the analyser rejects REXPRN and accepts LREXPRN as the goal.

When dealing with conditional statements, there is no 'left-hand-side' to the expression. In addition, the condition may have three expressions.

Consider the IMP statement

%IF 'A' < 'B' < 6 %THEN %START ;

Until the third expression has been parsed, the first two expressions could have been identified as either integer or string. Thus the type of expressions within conditional statements cannot be confirmed until the condition has been fully parsed; neither can any constants be replaced by generated names.

This problem was resolved by making two parses of conditional statements. The first parse types the individual expressions within a conditional statement; the second parse makes the appropriate text transformations. This is achieved by invoking the syntax analyser recursively. The process involves the following steps:

(1) On recognising the delimiter %IF, the analyser invokes ∅83 (app. (4):59).

(2) ∅83 invokes the analyser recursively. The analyser is now controlled by the syntax definitions described in appendix (6). The analytic-routine calls embedded in these definitions refer to a different set of analytic-routines to those embedded in the main definitions of appendix (4).

(3) Each expression is initially assumed to be of type integer. As each element of the expression is recognised, a transition matrix is referenced. The rows of this matrix correspond to the type of expression (state) as surmised to the present time; the columns correspond to the type of the new element (Input). Each element of the matrix has one of three different values.

Value	Meaning
1	Update the expression type to that indicated by the expression element just recognised (occurs if, for example, a real variable is encountered in an integer expression).
0	No further information; leave the expression type.
-1	Conflicting types; call error (occurs if, for example, a real constant is encountered in a string expression).

(4) When the condition has been fully parsed, a decision as to the type of each expression is reached and this decision conveyed to the analytic-routines of the main definitions of appendix (4) via global storage locations.

(5) Control is returned to $85.

(6) Control is returned to the analyser and the parse continues with the examination of definition 59 in appendix (4).

The analyser mentioned above is that described in Chapter 2. The root node of the parse can be defined at the time of calling the analyser. Thus the analyser could be used to type an individual expression by controlling it with the definitions of appendix (6) but invoking it with ⟨EXPRN⟩ as the root node instead of ⟨UBSC⟩.

Since IMP initially puts all constants in arithmetic expressions into double length floating point locations regardless of context or appearance, it follows that real constants, capable of being stored in integer locations with no overflow are allowed in integer expressions. A test on the IMP compiler revealed that such is indeed the case, though it is not mentioned in the current language manual (ERCC 70).

This allowed for by the third pass, the real constants being replaced by their integer equivalents (app. (4):127). String constants are replaced by generated names, declared and suitably initialised at the head of the outer block.

4.5 DETAILS OF THE TRANSLATION TO IMP(INT)

Appendix (4) shows the syntax definitions and target language constructions used to control and affect the translation.

Each statement has been categorised into one of several types (⟨S1⟩ , ⟨S2⟩ .... or ⟨S38⟩).

The definitions include a number of dummy meta-variables. These always begin with the string 'PARAM' or 'DUM' and are reducible to the empty string. Those beginning 'PARAM' are used to store parameters in the syntax tree which are needed during the generation stage. Such a technique has been described in section (2.4.2). The meta-variables, beginning 'DUM' are used to enable the analytic-routines to control the analysis and make it selective as explained in section (2.3.2).

Appendix (5) shows the syntax definitions of IMP(INT) in a form suitable for comparison with appendix (4). Some definitions in appendix (5) have a direct correspondence to definitions in appendix (4) and they are given identical definition numbers. Those that have a less precise correspondence (such as ⟨OQSVNAME⟩), together with the definitions of newly introduced meta-variables are numbered from 303.

The various text transformations performed by the third pass are described below. Each transformation is described under a heading which is a number(s) referring to the relevant definition(s) of appendix (4).

39

In the specification of explicitly specified routines of the IMP library, the delimiter %EXTERNAL is replaced by the new delimiter %LIBRARY (app. (5):39).

42

Internal routines need not be explicitly specified in IMP. If a routine heading is processed and there has been no previous specification for that routine in the current block, then the IMP compiler first treats the routine heading as an appropriate specification and makes the corresponding entries in the symbol table. IMP(INT), however, allows no such implicit specification. Thus in definition 42 of appendix (4), ∅76 checks to see if the corresponding routine name has been declared in the current block, If it hasn't then a corresponding specification is inserted immediately prior to the routine heading. Control is returned to point C in the analyser, forcing it to process definition 43 and the appropriate entries in the symbol table are made.

53

The reduced form of routine specification, as described by definition 53 of appendix (4) is expanded to the full form as described by definition 41. This is achieved by referencing the information in the appropriate parameter definition at the current level.

55, 56 and 57

A type marker (app. (5): 355 to 357) is inserted following the %END of a function or map body.

59 and 60

Expressions occurring within conditional statements are given a prefixed type marker (app (5): 345 to 354). This involved two parses of the condition as described in section (4.4). The string length of conditional string expressions is given a nominal (255) value.

67 and 81

Integer labels are replaced by constructed names. The meta-variable ⟨PARAM110⟩ invokes ∅110 which checks routine RFTABLE (see section (4.3.3)) to see if a constructed name has already been associated with the label. It uses a dummy node in the syntax tree to pass the four character key representing the constructed name to constructor-routine 11 (?11) during generation. (∅110 returns to point B in the analyser as described in section (2.4.2)). ∅87 inserts the new label in the symbol table, if necessary.

70 and 168

A type marker (app (5): 343 and 344) is post-fixed to the name of a pointer variable. If this variable is a record pointer, the type marker is a delimiter (%REC) followed by the name of the record format with which that pointer is associated. The entry for the record format name in RFTABLE is given a distinctive marking (see section(4.3.3)). The type marker refers to the qualified name. This may be the element of a record pointer which in turn is an element of a record pointer etc. This is one of the reasons for the complex looking syntax construction and many analytic-routines used in the analysis of < OQSVNAME > .

In addition to the markers noted above, a map function name (occurring in the general context of < OQSVNAME >) is postfixed by the delimiter %MP, followed by the marker appropriate to the map (app (5): 32∅). Thus when processing definition 172 of appendix (4), ∌26 exits to point C in the analyser (and thus forces definition 173 to be considered) unless the name associated with <NAME1> is a map function name. Constructor-routine 17 (?17) in the associated target language construct inserts %MP.

Another complication in the analysis of <OQSVNAME> arises due to the possible occurrence of transparent library functions, such as LOG or MOD. The type of these functions depends on the type of their arguments; they may be either real or long real. The symbol table entry for them has a distinctive marking (see appendix (8)). In definition 173 of appendix (4), ∌29 examines the descriptor associated with the identifier name and exits to point C in the analyser if it is not a transparent function, forcing definition 175 to be processed. Otherwise, the argument of the function is first analysed using the meta-variable < REXPRN > . If that fails, <LREXPRN> should work and the function is pronounced long real.

74 to 78

The expression on the right hand side of an assignment statement is prefixed by an appropriate type marker (app (5): 74 to 78).

84 and 85

%RESULT expressions are prefixed by a type marker. Those relating to map functions include the delimiter %MP (app. (5): 84 and 85).

105 to 114

The hierarchy of logical operators is made redundant by the insertion of special bracket delimiters %L and %R (app (5):306).

This is achieved by syntax analysis alone, the technique being similar to that quoted in the revised report on Algol 60 (Back 60). Note that the technique could not have been used if the prefix operator to arithmetic expressions had not been eliminated during the second pass, because the syntax definitions would then have become left-recursive.

126, 127, 153 and 159

As stated in section (4.4), the only type of constant allowed in expressions in IMP(INT), are integer constants in integer expressions. All others are replaced by generated names and a %OWN variable of suitable type (%REAL, %LONGREAL or %STRING (q)), appropriately initialised, is included at the head of the outer block.

106, 130 and 156

The delimiter %EXP is replaced by %IXP in integer contexts, %RXP in real contexts and %DXP in long real contexts (app (5): 309, 130, 156).

119

Arithmetic division of integers is distinguished by replacing the division operator by the special delimiter %BY. Constructor-routine 30 writes %BY to the output.

184 and 185

IMP(INT) requires actual parameters which are positionally defined as type general to be prefixed by the special delimiter %NM (app (5):332). This applies only to %LIBRARY routine parameter lists.

198

Subscript lists have their brackets replaced by the special delimiters %LB and %RB (for example app (5):322).

GENERAL DISCUSSION

The original purpose of the project was to write a suite of programs to translate a program written in IMP(SYS) into one defined in IMP(INT); the source program having already been passed through a lexical analyser. This has been achieved using the methods described in the previous four chapters. At the time of writing about 25 test programs have been translated into IMP(INT) form. These programs have incorporated all the foreseeable features of the translator.

The translation from IMP(INT) form into PL/1 was the responsibility of T. Nonweiler. At the time of writing, the program to affect this translation had not been completed. Details of this program are summarised in appendix 9.

In its design, the efficiency of the translator (that is its ability to conserve both time and space during translation) has been generally considered secondary to the speed with which it could be programmed. As described later on, this has resulted in a program that is relatively slow.

5.1 PROGRAMMING

The writing of the translator, once the overall methods had been defined was done in three separate steps.

First, algorithms central to both the second and third passes were written. This included the development and testing of the analyser. The importance of being able to have an absolute knowledge of the analyser's capabilities cannot be overestimated. When this was realised, the debugging of the second and third passes became a great deal easier. It can be noted that the analyser itself was probably the most tedious routine of all to debug.

Secondly, programs to achieve the second pass were written. This included the formation of the syntax definitions and the writing of appropriate analytic-routines and constructor-routines. In general, the second pass demanded a lot less expertise from a programmer than the third. Thus, because it was attempted first, valuable experience of using the various translation techniques was gained before they were applied to the more difficult third pass.

Thirdly, programs to achieve the third pass were written. This demanded by far the most programming effort and time. Over 1000 Pl/1 statements are used to define the analytic-routines alone. The programs for the third pass were written so that the full translation to IMP(INT) of every feature of a possible source program was attempted straight away. It might possibly have been easier to write a program to translate only a few features to IMP(INT) at first and gradually develop the full program.

It was always possible to quickly identify any errors encountered during the second or third passes. This was because it was very easy to get a good trace of the path taken by the translator by examining the syntax definitions and noting the particular analytic-routines or constructor-routines called. Note that such a process would have been much more hazardous if the analyser could not be absolutely relied upon.

Thus although the author has had no previous experience on which to base a judgement, it is felt that the translation was quite easy to program and debug considering the problems to be solved. In addition, it is thought that any future superficial modifications to the specifications of either IMP(SYS) or IMP(INT) will be easily incorporated within the translator.

5.2 PERFORMANCE

A breakdown of the translation of a typical 330 statement IMP source program is

PASS	TIME (secs)
1st	16
2nd	43.7
3rd	104.2

From consideration of these figures, the fourth pass (which requires less analysis than the second pass) would probably take about 40 secs.

This translation was made on an IBM 370/155 machine using a maximum of 190K bytes of storage. Neglecting the time taken in initialising routines, setting up each job step and reading in library files (such as the syntax definitions) which is of the order of 3 secs, the average translation time per IMP statement is about 0.6 secs.

This does not compare very favourably with the times quoted by the relevant users guide (IBM 68) for a translation from ALGOL to PL/1. On an IBM 360/40 machine, using 128K bytes of main storage, the average translation time is quoted as about (65 + 0.6N) seconds where N is the number of ALGOL statements.

The times taken by each pass during the translation to IMP(INT) show that the third pass takes far longer than the other two.

By comparing the times taken by a variety of IMP programs, it was soon realised that the analysis of arithmetic expressions represents the most time-consuming process. As explained below, this becomes obvious when one considers the large number of analytic-routines called during the processing of arithmetic expressions. Ironically, the first person to inquire about the translator submitted an IMP program of 4500 statements, mostly composed of long arithmetic expressions. The third pass took over 33 minutes and the complete translation time to PL/1 is expected to be over 0.8 secs per statement.

At the time of writing, there is no function supplied by IBM that can give a user access to a CPU clock. Certainly, there is a function, TIME, that gives elapsed time but in an MVT environment (in which the translator was programmed) this is of limited value.

However, a routine, written in assembly language became available to the author which did access the CPU clock. On testing various features of the third pass, it was confirmed that analysis of arithmetic expressions is the most time-consuming. The table below gives the times taken (on an IBM 370/55) processing various statements in the third pass.

Statement	Time in milliseconds
%REAL X,Y,Z:	123
%GOTO 25:	53
X = Y+Z:	505
%IF X=1 %THEN START:	485
X = (Y+Z) *X **2 + Y-Z/4 2	1184

Originally it was thought that the long time spent during the third pass processing the 4500 statement program may have also been due to the fact that the symbol tables were very large and thus the search time for an identifier was increased. On investigation with the CPU timer routine this was found to be untrue. The search time for an identifier in the symbol table being 3 milliseconds whether the table had 100 or 500 elements in it.

Various methods of improving the efficiency of the translator in general and the speed of the second and third passes in particular come to mind. They are numbered below.

1. The number of passes made by the translator could be reduced. Most easily eliminated would be the second pass. Untranslateable IMP phrase structures could be easily identified during the third pass and it could be clearly stated that no particular reaction by the translator to an IMP program with a syntax error in it is guaranteed (as is stated for the ALGOL-to-PL/1 translator). Or alternatively, the IMP compiler could be invoked from an assembly language routine within the translator to check both the syntax and semantics.

However the second pass also defines precisely the meaning of a particular use of the symbol ! (section (3.4.2)). This results in a much simpler structure for the third pass. It is understood though, that when a new specification for E.R.C.C. IMP is introduced towards the end of 1973, a new delimiter will be defined to represent the modulus sign which would largely solve the above problem.

Other passes made by the translator could not be removed wihtout a complete redesign of the translator system.

2. The top-down parsing algorithm chosen would not be as fast as one using a bottom-up method of analysis, though, as mentioned in Chapter 1, Capon and Argent (Cap 73) have shown that a bottom-up method is only fractionally faster. Probably more important is the time taken to invoke analytic routines. The reason why arithmetic expressions take so long to analyse becomes obvious when one considers that in the analysis of the assignment statement

$$L(K) = H * B + C$$

sets of analytic routines are invoked 5 times by the 2nd pass and of the order of 40 times (depending on the arithmetic type) by the 3rd pass. In the case of the third pass this can be attributed to a particularly ugly definition of the meta-variable <OQSVNAME> - which is used to represent an identifier and stands for 'optionally qualified and subscripted variable name'. More effort by the author should result in a simpler and/or more quickly processed definition; but such a thing is difficult because of the possible presence of record pointers or transparent functions, etc.

If the parsing algorithm were changed, the second and third passes would of course have to be substantially rewritten as it is unlikely that the new parsing algorithm would incorporate concepts identical to that of the analytic and constructor-routines used here.

3. It is rare for a program to reach the full sophistication of its compiler and thus most IMP programs will not be affected by the optimisation of boolean expressions done by the IMP compiler. So an easy modification that would speed up the second pass would be the elimination of the call on functions READS. Instead of breaking down compound conditional statements they could be translated directly into PL/1 with a warning given of the possibility of error between the IMP source and the PL/1 translation.

4. Another modification which would speed up the translation is to replace the four separate job steps by a single job step using an overlay structure. This would save the time spent in initialising a job step and in writing out and reading in the intermediate temporary disk files such as IMPCHKD. It would also mean that absolute machine addresses could be used to point to the various lists built up during the third pass. The use of offsets and the process of identifying which particular area a list was constructed in would then be unnecessary, which should represent the saving of an appreciable amount of time.

5.3 CONCLUSIONS

A feature of the IMP to IMP(INT) translation is that it does not relate to the PL/1 translated text directly. Thus, it is conceivable that this translation may be of use in the translation to a high level language other than PL/1; Algol 68 perhaps, through no research in this direction has been done.

It is hoped that the translator will finally be available to a user in the form of a catalogued procedure with a variety of options as to the form of the PL/1 translation.

The Edinburgh implementation of the IMP language is still at a development stage. Towards the end of this year, 1973, a new specification for it is to be introduced. This will include various new delimiters such as the one already mentioned to denote a modulus and the delimiter %WHILE to be used in do-statements (%CYCLE statements).

Whether or not all these changes will be incorporated in the translator or whether much effort is made to increase its efficiency depends on the demand there is for its use.

Much of the software in Edinburgh is written in IMP. In addition, Edinburgh is expected to buy an ICL 'New Range' machine shortly which will be supplied with a PL/1 compiler. It is thus hoped that this will produce a demand for the translator from users who want a direct transliteration of their programs from IMP to PL/1 instead of completely rewriting them. It is expected though that the main demand for the translator will come from users who want to export their IMP programs to computing environments with a PL/1 compiler but no IMP one.

THE FIRST PASS

The first pass performs a lexical analysis of the IMP source program, converting the text into a form more easily processed during the subsequent analysis.

As noted by Nonweiler (Non 72), the IMP source is supposed to conform to the syntax rules defined in the current IMP manual (ERCC 70) with the following amendments and additions:

(1) %VECTOR is acceptable in place of the declarator %ARRAY.

(2) The following statements are permitted

%SHORTROUTINE <SEPARATOR>

%DEFINECOMPILER <SEPARATOR>

%SPECIALNAME <NAMELIST> <SEPARATOR>

%REGISTER <NAMELIST> <SEPARATOR>

<TYPE> <ARRAY'> %NAME <NAMELIST> <SEPARATOR>

(3) %MONITOR <CONST> is an alternative unconditional instruction to %MONITOR.

(4) %NAMEARRAYNAME is not an accepted form of <QNAME'>.

(5) <TYPE> %NAMEARRAY is not an accepted form of declarator.

(6) The symbol $ is a valid constant.

(7) The ordering of array declarators after scalar declarations within record formats is optional.

The first pass reduces delimiters, identifiers, constants and comment to fixed length, easily recognisable units as follows:

(1) Cased letter strings (that is, IMP delimiters recognisable by an initial character %) are passed to a routine that defines a single EBCDIC character to represent that delimiter. This

will not be printable.

(2) Certain two-character delimiters are converted to translator-created cased letter strings as follows

**	becomes	%EXP
//	"	%DIV
==	"	%PTS
<<	"	%LSH
>>	"	%RSH
->	"	%GOTO

which are then processed as (1) above. Other two-character delimiters and all single character delimiters are passed unchanged, with the exception of the delimiter \= which is reinterpreted as £.

(3) Integer constants, including decimal integers (unsigned), hexadecimal and binary constants (signed) are converted to a five-character code; composed of character O followed by a four-character string obtained by overlaying the internal (32 bits) representation of the integer code onto the string.

(4) Real (unsigned) constants are converted to five-character strings. These strings are composed of character R followed by a four-character string which indexes a dictionary of constants, identifiers and comment. This dictionary is written to the file IMPDICT at the end of the pass.

(5) Symbol constants, including strings and single or multiple character constants are converted to five-character strings composed of character ' followed by a four-character string as in (4) above.

(6) Identifiers are converted to five-character strings composed of character A followed by a four-character string as in (4) above.

(7) Comment text is converted to five-character strings composed of character C followed by a four-character string as in (4) above.

In addition to the above lexical analysis, the first pass makes the following text transformations:

(1) The query marker ? is ignored.

(2) Newlines occuring as statement separators are replaced by the semi-colon.

(3) The delimiter %COMMENT (or ! when used to replace it) is eliminated.

(4) The delimiter %VECTOR is replaced by %ARRAY, and %REGISTER by %INTEGER.

(5) The symbol $ is reinterpreted as a real constant 3.1415...

(6) The intended effect (upon the subsequent interpretation of %REAL) is given in response to reading statements of the form

%REALS ⟨LN⟩

from the source text, and such statements are consequently eliminated.

(7) Null statements are ignored.

(8) The following statements are ignored:

```
⟨COMSTOP⟩%QUERIES ;
%LIST ;
%CONTROL ⟨CONST⟩ ;
%SHORTROUTINE ;
```

```
%QUERIES <ONOFF> ;
%ENDOFLIST ;
%DEFINECOMPILER ;
```

(9) The following statements are passed as null statements with an indication of error:

```
%SPECIALNAME <NAME> ;
* <INSTRN> ;
```

and the sequence %MCODE to %ENDOFMCODE inclusive.

(10) Conditional statements are reordered so that they begin with %IF or %UNLESS, and so that %THEN and %ELSE are always followed by %START ; (the appropriately positioned %FINISH statement being also inserted if necessary).

This first pass was programmed by T. Nonweiler and it's effect is described in slightly greater detail in his report (Non 72).

APPENDIX 2.

THE REDUCTION OF COMPOUND CONDITIONAL STATEMENTS

Consider the boolean expression

```
A %AND ( B %OR \C )
```

If A is 'false' then there is no need to evaluate further; the result is 'false'. Similarly, if A is 'true' and B is 'true', the result is 'true' and there is no need to evaluate \C. A compiler which produces code such that at run-time redundant boolean expressions (such as (B %OR \C) in the first example above and \C in the second) are not evaluated is said to be optimising boolean expressions.

The IMP compiler optimises boolean expressions whereas the PL/1 compiler does not. Thus, if a compound conditional statement is translated directly from IMP into PL/1, the codes generated by the respective IMP and PL/1 compilers would differ operationally.

There are at least two examples of cases where the evaluation of redundant boolean expressions might affect program execution:

(a) If evaluation of the redundant boolean expression created an error condition. For example consider the compound conditional statement

```
%IF A=0 %OR B/A <3.0 %THEN <UCI> ;
```

If A was equal to zero then the IMP compiler produces code which results in the second boolean expression not being evaluated. The PL/1 compiler however, asks for the evaluation to be made, which results in an overflow condition.

(b) If the evaluation of the redundant expression involved a call on a routine which in turn influenced global variables.

In order to overcome this possible source of error in the translation, all compound conditional statements and all statements beginning with the delimiter %UNLESS are first reduced to the form

```
            %IF <UBSC> %THEN %START ;
```

before a translation to PL/1 is made. This often involves the generation of new statements (and labels and thus reference to the dictionary kept on file IMPDICT).

For example the statement

```
            %IF A=0 %OR B/A < 3.0 %THEN C=D ;
```

might be transformed to

```
            %IF A=0 %THEN %START ;
                  %GOTO LAB1 ;
                  %FINISH ;
            %IF B/A < 3.0 %THEN %START ;
                  %GOTO LAB1 ;
                  %FINISH ;
            %GOTO LAB2 ;
      LAB1: C=D ;
      LAB2: ....
            ....
```

This transformation is performed by the function READSS and was programmed by T. Nonweiler. READSS is an interface routine to the second pass (see chapter (3)).

Syntax Definitions Controlling the Second Pass

```
<SS>          ::= <S01> !@1!                                              ...( 1)
              ::= <S02> !@1!                                              ...( 2)
              ::= <S03> !@1!                                              ...( 3)
              ::= <S04> !@1!                                              ...( 4)
              ::= <S05> !@1!                                              ...( 5)
              ::= <S06> !@1!                                              ...( 6)
              ::= <S07> !@1!                                              ...( 7)
              ::= <S08> !@1!                                              ...( 8)
              ::= <S09> !@1!                                              ...( 9)
              ::= <S10> !@1!                                              ...( 10)
              ::= <S11> !@1!                                              ...( 11)
              ::= <S12> !@1!                                              ...( 12)
              ::= <S13> !@1!                                              ...( 13)
              ::= <S14> !@1!                                              ...( 14)
              ::= <S15> !@1!                                              ...( 15)
              ::= <S16> !@1!                                              ...( 16)
              ::= <S17> !@1!                                              ...( 17)
              ::= <S18> !@1!                                              ...( 18)
              ::= <S19> !@1!                                              ...( 19)
              ::= <S23> !@1!                                              ...( 20)
              ::= <S24> !@1!                                              ...( 21)
              ::= <S25> !@1!                                              ...( 22)
              ::= <S26> !@1!                                              ...( 23)
              ::= <S27> !@1!                                              ...( 24)
              ::= <S28> !@1!                                              ...( 25)
<S01>         ::= <TYPEA> <DECLN> ; !@2@101!                              ...( 26)
              ::= <TYPEA> %NAME <NAMELIST> ; !@2@102!                     ...( 27)
              ::= <TYPEA> %NAME %ARRAY !@1F900!                           ...( 28)
              ::= <TYPEB> %ARRAY %NAME !F903!                             ...( 29)
<S02>         ::= <RT> %NAME !@1F901!                                     ...( 30)
<S03>         ::= %EXTERNAL $37 <RT> %SPEC <NAME> <EFPDEFN0> $38 ; !@104! ...( 31)
              ::= <SYS0> <RT> %SPEC <NAME> <EFPDEFN0> ; !@105!            ...( 32)
              ::= <EXTRN0> <RT> %SPEC !F100!                              ...( 33)
              ::= <EXTRN0> <RT> <NAME> <EFPDEFN0> ; $50 !@106!            ...( 34)
              ::= <EXTRN0> <RT> $50 !F101!                                ...( 35)
              ::= <TYPEB> !F102!                                          ...( 36)
              ::= <NULL> !F103!                                           ...( 37)
```

```
<S04>       ::= %OWN <TYPEA> <NAME> <INITIAL0> ; !@2@107!                          ...( 38)
            ::= %OWN <TYPEA> %ARRAY <NAME> <CBPAIR> <INITLIST0> ; !@3@2@108!        ...( 39)
            ::= %OWN <TYPEA> %ARRAY <NAME> !@1F104!                                ...( 40)
            ::= %OWN <TYPEA> !@1F105!                                              ...( 41)
            ::= %OWN !F106!                                                        ...(-42)
            ::= <NULL> !F107!                                                      ...( 43)
<S05>       ::= %SWITCH <SWITCHLIST> ; !@109!                                      ...( 44)
            ::= %SWITCH !F108!                                                     ...( 45)
            ::= <NULL> !F109!                                                      ...( 46)
<S06>       ::= %RECORD %FORMAT <NAME> ( <RFDEFN> ) ; !@110!                       ...( 47)
            ::= %RECORD %FORMAT <NAME> !F110!                                      ...( 48)
            ::= %RECORD <DECLN> ( <NAME> ) ; !@111!                                ...( 49)
            ::= %RECORD <DECLN> !@1F134!                                           ...( 50)
            ::= %RECORD %NAME <NAMELIST> ( <NAME> ) ; !@112!                       ...( 51)
            ::= %RECORD %NAME <NAMELIST> !F111!                                    ...( 52)
            ::= %RECORD %SPEC <NAME> <ENAME00> ( <NAME> ) ; !13!                   ...( 53)
            ::= %RECORD %SPEC !F112!                                               ...( 54)
            ::= %RECORD %ARRAY %NAME !F902!                                        ...( 55)
            ::= %RECORD !F113!                                                     ...( 56)
            ::= <NULL> !F114!                                                      ...( 57)
75. <S07>   ::= %SPEC <NAME> ⊥ !F904!                                              ...( 58)
            ::= %SPEC <NAME> <EFPDEFN0> ; !@114!                                   ...( 59)
            ::= %SPEC !F116!                                                       ...( 60)
            ::= <NULL> !F117!                                                      ...( 61)
<S08>       ::= %BEGIN ; $51 !15!                                                  ...( 62)
            ::= <NULL> !F118!                                                      ...( 63)
<S09>       ::= %END ; $57 !?0416!                                                 ...( 64)
            ::= %END %OFPROGRAM ; $57 $58 !?04?0517!                               ...( 65)
            ::= %END %OFFILE ; $57 $58 !?04?0517!                                  ...( 66)
            ::= <NULL> !F119!                                                      ...( 67)
<S10>       ::= C $12 !18!                                                         ...( 68)
            ::= <NULL> !F120!                                                      ...( 69)
<S11>       ::= %IF <UBSC> %THEN %START ; $54 !@119!                               ...( 70)
            ::= %IF <RESOLUTION> %THEN %START ; $54 !@120!                         ...( 71)
            ::= %IF !F121!                                                         ...( 72)
            ::= <NULL> !F122!                                                      ...( 73)
<S12>       ::= %START ; $53 !!                                                    ...( 74)
            ::= <NULL> !F194!                                                      ...( 75)
```

```
<S13>            ::= %FINISH ; $55 !?0222!                                          ...( 76)
                 ::= %FINISH %ELSE !?0223!                                          ...( 77)
                 ::= <NULL> !F195!                                                  ...( 78)
<S14>            ::= %CYCLE <OQSVNAME> = <ARITHEX> , <ARITHEX> , <ARITHEX> ; $52
                     !@4@3@2@123!                                                   ...( 79)
                 ::= %CYCLE <OQSVNAME> = <ARITHEX> , <ARITHEX> $52 !@3@2@1F124!     ...( 80)
                 ::= %CYCLE <OQSVNAME> = <ARITHEX> $52 !@2@1F125!                   ...( 81)
                 ::= %CYCLE <OQSVNAME> $52 !@1F126!                                 ...( 82)
                 ::= <NULL> !F127!                                                  ...( 83)
<S15>            ::= %REPEAT ; $56 !?0324!                                          ...( 84)
                 ::= <NULL> !F128!                                                  ...( 85)
<S16>            ::= %FAULT <FAULTLIST> ; !@125!                                    ...( 86)
                 ::= %FAULT !F129!                                                  ...( 87)
                 ::= <NULL> !F130!                                                  ...( 88)
<S17>            ::= <LABEL> !26!                                                   ...( 89)
                 ::= <NULL> !F131!                                                  ...( 90)
<S18>            ::= A $13 ( \ 0 $42 !27!                                           ...( 91)
                 ::= A $13 ( + 0 $43 !27!                                           ...( 92)
                 ::= A $13 ( - 0 $44 !27!                                           ...( 93)
                 ::= A $13 ( 0 !27!                                                 ...( 94)
                 ::= A $13 ( <PLUS0> <INTONLY> !@1!                                 ...( 95)
                 ::= A $13 ( !F132!                                                 ...( 96)
                 ::= <NULL> !F133!                                                  ...( 97)
<S19>            ::= <OQSVNAME> <ASSOP> <EXPRN> ; !@3@130!                          ...( 98)
                 ::= <OQSVNAME> <ASSOP> !@2F203!                                    ...( 99)
                 ::= <OQSVNAME> %PTS <OQSVNAME> ; !@2@129!                          ...(100)
                 ::= <OQSVNAME> %PTS !@1F202!                                       ...(101)
                 ::= $10 <RESOLUTION> !@128!                                        ...(102)
                 ::= $10 $10 <NAME> <ACT_PARMS0> ; !@131!                           ...(103)
                 ::= $10 $10 <NAME> <ACT_PARMS0> _ !@1F914!                         ...(104)
                 ::= $10 $10 <NULL> !!                                              ...(105)
<S23>            ::= %GOTO <LABEL> ; !32!                                           ...(106)
                 ::= %GOTO <NAME> ( <ARITHEX> ) ; !@133!                            ...(107)
                 ::= %GOTO A !F142!                                                 ...(108)
                 ::= <NULL> !F143!                                                  ...(109)
<S24>            ::= %PRINTTEXT <STRING> $11 ; !31!                                 ...(110)
                 ::= <NULL> !F145!                                                  ...(111)
<S25>            ::= %RETURN ; !35!                                                 ...(112)
```

```
                  ::= <NULL> !F146!                                              ...(113)
<S26>             ::= %RESULT = <EXPRN> ; !@136!                                 ...(114)
                  ::= <NULL> !F147!                                              ...(115)
<S27>             ::= %STOP ; !37!                                               ...(116)
                  ::= <NULL> !F148!                                              ...(117)
<S28>             ::= %MONITOR Ø $13 ; !38!                                      ...(118)
                  ::= %MONITOR ; !38!                                            ...(119)
                  ::= %MONITOR Ø $13 %STOP ; !3837!                              ...(120)
                  ::= %MONITOR %STOP ; !3837!                                    ...(121)
                  ::= <NULL> !F149!                                              ...(122)
<TYPEA>           ::= <ARITHTYPE> !!                                             ...(123)
                  ::= <STRINGTYPEA> !@1!                                         ...(124)
<TYPEB>           ::= <ARITHTYPE> !!                                             ...(125)
                  ::= <STRINGTYPEB> !!                                           ...(126)
<ARITHTYPE>       ::= %INTEGER !!                                                ...(127)
                  ::= %REAL !!                                                   ...(128)
                  ::= %BYTE %INTEGER !!                                          ...(129)
                  ::= %SHORT %INTEGER !!                                         ...(130)
                  ::= %LONG %REAL !!                                             ...(131)
<STRINGTYPEA>     ::= %STRING ( Ø $13 ) !!                                       ...(132)
                  ::= %STRING !F907!                                             ...(133)
<STRINGTYPEB>     ::= %STRING ( Ø $13 ) !!                                       ...(134)
                  ::= %STRING !!                                                 ...(135)
<DECLN>           ::= %ARRAY <ARRAYLIST> !@1!                                    ...(136)
                  ::= <NAMELIST> !@1!                                            ...(137)
<ARRAYLIST>       ::= <NAMELIST> ( <BPLIST> ) , <ARRAYLIST> !@2!                 ...(138)
                  ::= <NAMELIST> ( <BPLIST> ) !@1!                               ...(139)
                  ::= <NAMELIST> !F150!                                          ...(140)
<NAMELIST>        ::= <NAME> , <NAMELIST> !!                                     ...(141)
                  ::= <NAME> !!                                                  ...(142)
<BPLIST>          ::= <ARITHEX> : <ARITHEX> <RO_BPLIST> !@3@2@1!                 ...(143)
                  ::= <ARITHEX> : !@1F153!                                       ...(144)
<RO_BPLIST>       ::= , <BPLIST> !@1!                                            ...(145)
                  ::= , !F196!                                                   ...(146)
                  ::= <NULL> !!                                                  ...(147)
<EXPRN>           ::= <ARITHEX> !@1!                                             ...(148)
                  ::= <CONCATION> !@1!                                           ...(149)
<CONCATION>       ::= <STR_OPD> . <STR_OPD> <RO_CONCATION> !@3@2@1!              ...(150)
```

75.

```
                ::= <NULL> !!                                                    ...(189)
<ACT⊥PARMS0>    ::= ( <EXPRN> <RO⊥ACT⊥PARMS> ) !@2@1!                            ...(190)
                ::= ( !F161!                                                     ...(191)
                ::= <NULL> !!                                                    ...(192)
<RO⊥ACT⊥PARMS>  ::= , <EXPRN> <RO⊥ACT⊥PARMS> !@2@1!                              ...(193)
                ::= , !F162!                                                     ...(194)
                ::= <NULL> !!                                                    ...(195)
<RT>            ::= %ROUTINE !!                                                  ...(196)
                ::= <TYPEB> <FM> !!                                              ...(197)
<FM>            ::= %FN !!                                                       ...(198)
                ::= %MAP !!                                                      ...(199)
<SYS0>          ::= %SYSTEM !!                                                   ...(200)
                ::= <NULL> !!                                                    ...(201)
<EXTRN0>        ::= %EXTERNAL !!                                                 ...(202)
                ::= <SYS0> !!                                                    ...(203)
<EFPDEFN0>      ::= ( <EFPLIST> ) !@1!                                           ...(204)
                ::= <NULL> !!                                                    ...(205)
<EFPLIST>       ::= $32 <EFPDELIMITER> $35 <FPNAMELIST> <RO⊥EFPLIST> !@3@1!      ...(206)
<EFPDELIMITER>  ::= <FPDELIMITER> !@1!                                           ...(207)
                ::= <ARRAY0> %NAME !?01!                                         ...(208)
<RO⊥EFPLIST>    ::= <COMMA0> <EFPLIST> !@1!                                      ...(209)
                ::= <NULL> !!                                                    ...(210)
<COMMA0>        ::= , $20 !!                                                     ...(211)
                ::= <NULL> !!                                                    ...(212)
<FPNAMELIST>    ::= <NAME> <RO⊥FPNLIST> !!                                       ...(213)
<RO⊥FPNLIST>    ::= , <NAME> $33 $36 <RO⊥FPNLIST> !!                             ...(214)
                ::= <NULL> !!                                                    ...(215)
<FPDELIMITER>   ::= <RT> !!                                                      ...(216)
                ::= <TYPEB> %NAME %ARRAY %NAME !F908!                            ...(217)
                ::= <TYPEB> <ARRAY0> %NAME !!                                    ...(218)
                ::= <TYPEA> !@1!                                                 ...(219)
                ::= %RECORD <ARRAY0> %NAME !@1!                                  ...(220)
<INITIAL0>      ::= = <SGND⊥CONST> !@1!                                          ...(221)
                ::= = !F168!                                                     ...(222)
                ::= <NULL> !!                                                    ...(223)
<SGND⊥CONST>    ::= <SGND⊥INT> !!                                                ...(224)
                ::= <PLUS0> <STRING> !!                                          ...(225)
                ::= \ R !F169!                                                   ...(226)
                ::= <MINUSØ> <REAL⊥CONST> !!                                      ...(227)
<SGND⊥INT>      ::= Ø $13 !!                                                      ...(228)
                ::= + Ø $43 !!                                                    ...(229)
                ::= - Ø $44 !!                                                    ...(230)
                ::= \ Ø $42 !!                                                    ...(231)
<MINUSØ>        ::= + $20 !!                                                      ...(232)
                ::= - !!                                                          ...(233)
                ::= <NULL> !!                                                     ...(234)
<PLUSØ>         ::= \ !!                                                          ...(235)
                ::= <MINUSØ> !!                                                   ...(236)
<INTONLY>       ::= R $13 !F905!                                                  ...(237)
                ::= ' $13 !F906!                                                  ...(238)
                ::= Ø $13 !!                                                      ...(239)
<CBPAIR>        ::= ( <SGND⊥INT> : <SGND⊥INT> ) !!                                ...(240)
                ::= ( <PLUSØ> <INTONLY> : <PLUSØ> <INTONLY> ) !@3@1!              ...(241)
                ::= ( !F172!                                                      ...(242)
                ::= <NULL> !F173!                                                 ...(243)
<INITLISTØ>     ::= = <ILIST> !@1!                                                ...(244)
                ::= = !F174!                                                      ...(245)
                ::= <NULL> !!                                                     ...(246)
<ILIST>         ::= <SGND⊥CONST> <QUALIFIERØ> , <ILIST> !!                        ...(247)
                ::= <SGND⊥CONST> <QUALIFIERØ> !!                                  ...(248)
                ::= <NULL> !F175!                                                 ...(249)
<QUALIFIERØ>    ::= ( Ø $13 ) !!                                                  ...(250)
                ::= ( !F176!                                                      ...(251)
                ::= <NULL> !!                                                     ...(252)
<SWITCHLIST>    ::= <NAMELIST> <CBPAIR> ,<SWITCHLIST> !@3@2@1!                    ...(253)
                ::= <NAMELIST> <CBPAIR> !@2@1!                                    ...(254)
                ::= <NULL> !F177!                                                 ...(255)
<PCNAMEØ>       ::= %NAME !!                                                      ...(256)
<PCNAME>        ::= <NULL> !!                                                     ...(257)
<ENAMEØØ>       ::= ⊥ <NAME> !!                                                   ...(258)
                ::= <NULL> !!                                                     ...(259)
<ENAMEØ>        ::= ⊥ <OQSVNAME> !@1!                                             ...(260)
                ::= ⊥ !F183!                                                      ...(261)
                ::= <NULL> !!                                                     ...(262)
<ASSOP>         ::= = !!                                                          ...(263)
                ::= <- $41 !!                                                     ...(264)
```

```
<RESOLUTION>    ::= $14 <OQSVNAME> %GOTO <RHEADØ> <RESLN> <RTAILØ> $16 !@4@3@2@1!        ...(265)
                ::= $14 <OQSVNAME> %GOTO !@1F135!                                         ...(266)
<RHEADØ>        ::= <OQSVNAME> . !@1!                                                     ...(267)
                ::= <NULL> !!                                                             ...(268)
<RESLN>         ::= ( $15 <STR⊥EXPRN> ) . <OQSVNAME> , <RESLN> !@3@2@1!                  ...(269)
                ::= ( $15 <STR⊥EXPRN> ) !@1!                                              ...(270)
<RTAILØ>        ::= . <OQSVNAME> !@1!                                                     ...(271)
                ::= <NULL> !!                                                             ...(272)
<STR⊥EXPRN>     ::= <STR⊥OPD> !@1!                                                        ...(273)
                ::= <CONCATION> !@1!                                                      ...(274)
<FAULTLIST>     ::= <NLIST> %GOTO <LABEL> , <FAULTLIST> !@3@2@1!                          ...(275)
                ::= <NLIST> %GOTO <LABEL> !@2@1!                                          ...(276)
<NLIST>         ::= <INT> , <NLIST> !@1!                                                  ...(277)
                ::= <INT> !!                                                              ...(278)
                ::= <NULL> !F200!                                                         ...(279)
<ARRAYØ>        ::= %ARRAY !!                                                             ...(280)
                ::= <NULL> !!                                                             ...(281)
<UBSC>          ::= <EXPRN> <COMP> <EXPRN> <RO⊥COND> !@4@3@2@1!                           ...(282)
<RO⊥COND>       ::= <COMP> <EXPRN> !@2@1!                                                 ...(283)
                ::= <NULL> !!                                                             ...(284)
<COMP>          ::= £ !!                                                                  ...(285)
                ::= = !!                                                                  ...(286)
                ::= > = !!                                                                ...(287)
                ::= > !!                                                                  ...(288)
                ::= < = !!                                                                ...(289)
                ::= < !!                                                                  ...(290)
<RFDEFN>        ::= <RFDEC> <RO⊥RFDEFN> !@2@1!                                            ...(291)
<RFDEC>         ::= <TYPEA> <DECLN> !@2@1!                                                ...(292)
                ::= <TYPEA> %NAME <NAMELIST> !@2!                                         ...(293)
                ::= %RECORD %NAME <NAMELIST> !!                                           ...(294)
                ::= %RECORD %ARRAY %NAME <NAMELIST> !F910!                                ...(295)
                ::= <RT> <NAMELIST> !@2F911!                                              ...(296)
                ::= <TYPEB> %ARRAY %NAME <NAMELIST> !F912!                                ...(297)
<RO⊥RFDEFN>     ::= <COMMAØ> <RFDEFN> !@1!                                                ...(298)
                ::= <NULL> !!                                                             ...(299)
<LABEL>         ::= A $13 !!                                                              ...(300)
                ::= Ø $13 !!                                                              ...(301)
<CONST>         ::= <INT> !!                                                              ...(302)
```

```
                  ::= <REAL⊥CONST> ⊥⊥                                    ...(303)
                  ::= <STRING> ⊥⊥                                        ...(304)
 <NAME>           ::= A $13 ⊥⊥                                           ...(305)
 <REAL⊥CONST>     ::= R $13 ⊥⊥                                           ...(306)
 <STRING>         ::= ' $13 ⊥⊥                                           ...(307)
 <INT>            ::= 0 $13 ⊥⊥                                           ...(308)
```

APPENDIX 4.

Syntax Definitions Controlling the Third Pass

```
<SS>      ::= <S1> !@1!                                                   ...(  1)
          ::= <S2> !@1!                                                   ...(  2)
          ::= <S4> !@1!                                                   ...(  3)
          ::= <S5> !@1!                                                   ...(  4)
          ::= <S6> !@1!                                                   ...(  5)
          ::= <S7> !@1!                                                   ...(  6)
          ::= <S8> !@1!                                                   ...(  7)
          ::= <S9> !@1!                                                   ...(  8)
          ::= <S1Ø> !@1!                                                  ...(  9)
          ::= <S11> !@1!                                                  ...( 1Ø)
          ::= <S12> !@1!                                                  ...( 11)
          ::= <S13> !@1!                                                  ...( 12)
          ::= <S14> !@1!                                                  ...( 13)
          ::= <S15> !@1!                                                  ...( 14)
          ::= <S16> !@1!                                                  ...( 15)
          ::= <S17> !@1!                                                  ...( 16)
          ::= <S18> !@1!                                                  ...( 17)
          ::= <S19> !@1!                                                  ...( 18)
          ::= <S2Ø> !@1!                                                  ...( 19)
          ::= <S22> !@1!                                                  ...( 2Ø)
          ::= <S23> !@1!                                                  ...( 21)
          ::= <S24> !@1!                                                  ...( 22)
          ::= <S25> !@1!                                                  ...( 23)
          ::= <S26> !@1!                                                  ...( 24)
          ::= <S27> !@1!                                                  ...( 25)
          ::= <S28> !@1!                                                  ...( 26)
          ::= <S29> !@1!                                                  ...( 27)
          ::= <S3Ø> !@1!                                                  ...( 28)
          ::= <S31> !@1!                                                  ...( 29)
          ::= <S32> !@1!                                                  ...( 3Ø)
          ::= <S33> !@1!                                                  ...( 31)
          ::= <S35> !@1!                                                  ...( 32)
          ::= <S36> !@1!                                                  ...( 33)
          ::= <S37> !@1!                                                  ...( 34)
          ::= <S38> !@1!                                                  ...( 35)
<S1>      ::= <TYPE> <NAMELIST> $17 ; !Ø1@2@1;!                           ...( 36)
          ::= <TYPE> %ARRAY <PARAM1> <ARRAYLIST> $18 ; !Ø1@3?92@1;!       ...( 37)
```

```
<S2>        ::= <TYPE> %NAME <PARAM1> $44 <NAMELIST> $17 ; !04@3?92@1;!                   ...( 38)
<S4>        ::= %EXTERNAL <PARAM1> <RT> %SPEC <PARAM1> <NAME> $45 <FPSPFN0> ;
                 !11?16@4?93@2@1;!                                                     ...( 39)
            ::= %EXTERNAL <PARAM1> <RT> %SPEC <PARAM1> <NAME2> <FPSPFN0> ;
                 !12?95@4?93@2@1;!                                                     ...( 40)
<S5>        ::= <SYS0> <RT> %SPEC <PARAM1> <NAME> <FPSPFN0> ; !12@5@4?93@2@1;!        ...( 41)
<S6>        ::= <EXTRN0> <RT> <NAME> <DUM1> $76 <FPDEFN0> ; !14@5@4@3@1;!             ...( 42)
            ::= <EXTRN0> <RT> <NAME> $66 <FPSPFN0> $88 ; !14@4@3@2@1;!                ...( 43)
<S7>        ::= %OWN <PARAM1> <TYPE> <NAME> <INITIAL0> $47 ; !02?94@3@2@1;!           ...( 44)
<S8>        ::= %OWN <PARAM1> <TYPE> %ARRAY <PARAM1> <NAME> ( <INTEGER> $49 :
                 <INTEGER> $50 ) <INITLIST0> $47 ; !03?97@6?95@4(@3:@2)@1;!            ...( 45)
<S9>        ::= %SWITCH <PARAM1> $53 <SWITCHLIST> ; !05?92@1;!                         ...( 46)
<S10>       ::= %RECORD <PARAM1> %FORMAT <PARAM1> <NAME> ( <RFDEFN> $57 ) ;
                 !06?94?93@2(@1);!                                                     ...( 47)
<S11>       ::= %RECORD <PARAM1> $55 <NAMELIST> ( <NAME> ) $60 ; !07?93@2(@1);!       ...( 48)
            ::= %RECORD <PARAM1> $55 %ARRAY <PARAM1> <ARRAYLIST> ( <NAME> ) $61 ;
                 !07?94?93@2(@1);!                                                     ...( 49)
<S12>       ::= %RECORD <PARAM1> $55 %NAME <PARAM1> $44 <NAMELIST> ( <NAME> ) $60
                 ; !08?94?93@2(@1);!                                                   ...( 50)
<S13>       ::= %RECORD <PARAM1> %SPEC <PARAM1> <NAME> ⊥ $62 <NAME> $63 ( <NAME> )
                 $64 ; !09?95?94@3⊥@2(@1);!                                            ...( 51)
            ::= %RECORD <PARAM1> %SPEC <PARAM1> <NAME> $62 ( <NAME> ) $65 ;
                 !09?94?93@2(@1);!                                                     ...( 52)
<S14>       ::= %SPEC <PARAM1> <NAME> <PARAM6> <FPSPFN20> ; !12?42?94@3@1;!           ...( 53)
<S15>       ::= %BEGIN <PARAM1> $66 ; !15?91;!                                         ...( 54)
<S16>       ::= %END <PARAM1> $67 ; !16?91;!                                           ...( 55)
<S17>       ::= %END <PARAM1> %OFPROGRAM <PARAM1> $67 ; !99?92?91;!                    ...( 56)
            ::= %END <PARAM1> %OFFILE <PARAM1> $67 ; !99?92?91;!                       ...( 57)
<S18>       ::= C <PARAM8> ; !17C?11;!                                                 ...( 58)
<S19>       ::= %IF <PARAM1> <DUM1> $83 <TEXPRN> <COMP> $85 <TEXPRN> <COMP> $86
                 <TEXPRN> %THEN <PARAM1> %START <PARAM1> ; !18?99@7@6@5@4@3?92?91;!
                                                                                       ...( 59)
            ::= %IF <PARAM1> $84 <TEXPRN> <COMP> $85 <TEXPRN> %THEN <PARAM1>
                 %START <PARAM1> ; !18?96@5@4@3?92?91;!                                ...( 60)
<S20>       ::= %IF <PARAM1> <RESLN⊥HEAD> <RTAIL0> %THEN <PARAM1> %START <PARAM1>
                 ; !18?95@4@3?92?91;!                                                  ...( 61)
<S22>       ::= %FINISH <PARAM1> ; !20?91;!                                            ...( 62)
<S23>       ::= %CYCLE <PARAM1> <OGSVNAME> $40 = <IEXPRN> , <IEXPRN> , <IEXPRN> ;
```

```
                 !21?95@4=@3,@2,@1;!                                       ...( 63)
<S24>        ::= %REPEAT <PARAM1> ; !22?91;!                                ...( 64)
<S25>        ::= %FAULT <PARAM1> <FAULTLIST> ; !23?92@1;!                   ...( 65)
<S26>        ::= <NAME> : $87 !24@1:!                                       ...( 66)
             ::= 0 <PARAM110> : $87 !24A?11:!                               ...( 67)
<S27>        ::= <NAME> ( <INTEGER> ) : !25@2(@1):!                         ...( 68)
<S28>        ::= <RESLN_HEAD> <RTAIL0> ; !26@2@1;!                          ...( 69)
<S29>        ::= <QUAL2> <NAME> $90 <PARAM7> $93 %PTS <PARAM1> <OQSVNAME> $40 ;
                 !27@5@4?25A?10?92@1;!                                      ...( 70)
             ::= <QUAL2> <NAME1> $94 <PARAM9> %PTS <PARAM1> <OQSVNAME> $40 ;
                 !27@5@4?83?92@1;!                                          ...( 71)
             ::= <NAME> $95 <PARAM96> %PTS <PARAM1> <OQSVNAME> $40
                 !27@4?25A?10?92@1;!                                        ...( 72)
             ::= <NAME1> $97 <PARAM9> %PTS <PARAM1> <OQSVNAME> $40 ; !27@4?83?92@1;!
                                                                            ...( 73)
<S30>        ::= <OQSVNAME> = $100 <TEXPRN> ; !28@2=@1;!                    ...( 74)
             ::= <OQSVNAME> %FRM <DUM1> $101 <PARAM102> <STR_EXPRN> ; !28@4?45@1;!  ...( 75)
             ::= <OQSVNAME> %FRM <DUM2> $103 <IEXPRN> ; !28@3?14@1;!        ...( 76)
             ::= <OQSVNAME> %FRM <DUM3> $104 <IEXPRN> ; !28@3?13@1;!        ...( 77)
             ::= <OQSVNAME> %FRM <PARAM1> <TEXPRN> !28@3?92@1;!             ...( 78)
<S31>        ::= <NAME> $105 <ACT_PARMS0> ; !29@2@1;!                       ...( 79)
<S32>        ::= %GOTO <PARAM1> <NAME> ; !30?92@1;!                         ...( 80)
             ::= %GOTO <PARAM1> 0 <PARAM110> ; !30?92A?11;!                 ...( 81)
<S33>        ::= %GOTO <PARAM1> <NAME> ( <IEXPRN> ) ; !32?93@2?33@1?34;!    ...( 82)
<S35>        ::= %RETURN <PARAM1> ; !33?91;!                                ...( 83)
<S36>        ::= %RESULT <PARAM1> = $106 <TEXPRN> ; !34?92=@1;!             ...( 84)
             ::= %RESULT <PARAM1> = <TEXPRN> ; !34?92=?17@1;!               ...( 85)
<S37>        ::= %STOP <PARAM1> ; !35?91;!                                  ...( 86)
<S38>        ::= %MONITOR <PARAM1> <INTEGER> ; !36?92@1;!                   ...( 87)
             ::= %MONITOR <PARAM1> ; !36?91;!                               ...( 88)
<TYPE>       ::= %BYTE <PARAM1> %INTEGER <PARAM1> $10 !?92?91!              ...( 89)
             ::= %SHORT <PARAM1> %INTEGER <PARAM1> $11 !?92?91!             ...( 90)
             ::= %INTEGER <PARAM1> $12 !?91!                                ...( 91)
             ::= %LONG <PARAM1> %REAL <PARAM1> $13 !?92?91!                 ...( 92)
             ::= %REAL <PARAM1> $14 !?91!                                   ...( 93)
             ::= %STRING <PARAM1> ( <INTEGER> ) $15 !?92(@1)!               ...( 94)
             ::= %STRING <PARAM1> $16 !?91!                                 ...( 95)
<NAMELIST>   ::= <NAMES> , <NAMELIST> !@2,@1!                               ...( 96)
```

```
                  ::= <NAMES> !@1!                                                    ...( 97)
<NAMES>           ::= A <PARAM3> !A?11!                                               ...( 98)
<INTEGER>         ::= 0 <PARAM4> !0?11!                                               ...( 99)
<NAME>            ::= A <PARAM8> !A?11!                                               ...(100)
<ARRAYLIST>       ::= <NAMELIST> $115 ( <BPLIST> ) $58 , <ARRAYLIST> !@3(@2),@1!      ...(101)
                  ::= <NAMELIST> $115 ( <BPLIST> ) $58 !@2(@1)!                       ...(102)
<BPLIST>          ::= <IEXPRN> : <IEXPRN> $19 , <BPLIST> !@3:@2,@1!                   ...(103)
                  ::= <IEXPRN> : <IEXPRN> $19 !@2:@1!                                 ...(104)
<IEXPRN>          ::= <TERM> <LOP1> <IEXPRN> !?31@3@2@1?32!                           ...(105)
                  ::= <TERM> %EXP <IEXPRN> !@2?28@1!                                  ...(106)
                  ::= <TERM> !@1!                                                     ...(107)
<TERM>            ::= <FACTOR> <LOP2> <TERM> !?31@3@2@1?32!                           ...(108)
                  ::= <FACTOR> * <TERM> !@2*@1!                                       ...(109)
                  ::= <FACTOR> !@1!                                                   ...(110)
<FACTOR>          ::= <IOPERAND> <LOP3> <FACTOR> !?31@3@2@1?32!                       ...(111)
                  ::= <IOPERAND> + <FACTOR> !@2+@1!                                   ...(112)
                  ::= <IOPERAND> - <FACTOR> !@2-@1!                                   ...(113)
                  ::= <IOPERAND> !@1!                                                 ...(114)
<LOP1>            ::= ! !!!                                                           ...(115)
                  ::= %NEV <PARAM1> !?91!                                             ...(116)
<LOP2>            ::= & !&!                                                           ...(117)
                  ::= %DIV <PARAM1> !?91!                                             ...(118)
                  ::= / !?30!                                                         ...(119)
<LOP3>            ::= %LSH <PARAM1> !?91!                                             ...(120)
                  ::= %RSH <PARAM1> !?91!                                             ...(121)
<IOPERAND>        ::= <INTEGER> !@1!                                                  ...(122)
                  ::= <OQSVNAME> $20 !@1!                                             ...(123)
                  ::= ( <IEXPRN> ) !(@1)!                                             ...(124)
                  ::= %M <PARAM1> <IEXPRN> %M <PARAM1> !?93@2?91!                     ...(125)
                  ::= ' <PARAM21> !A?11!                                              ...(126)
                  ::= R $140 <PARAM2> !0?11!                                          ...(127)
<REXPRN>          ::= <ROPERAND> <RO_REXPRN> $144 !@2@1!                              ...(128)
<RO_REXPRN>       ::= <AOP1> <ROPERAND> <RO_REXPRN> !@3@2@1!                          ...(129)
                  ::= %EXP <ISOPERAND> <RO_REXPRN> !?27@2@1!                          ...(130)
                  ::= <NULL> !!                                                       ...(131)
<ROPERAND>        ::= 0 $142 <PARAM2> !A?11!                                          ...(132)
                  ::= R $141 <PARAM2> !A?11!                                          ...(133)
                  ::= <OQSVNAME> $143 !@1!                                            ...(134)
                  ::= ! <PARAM21> !A?11!                                        ...(135)
                  ::= ( <ISUB⊥EXPRN> ) !(@1)!                                   ...(136)
                  ::= ( <REXPRN> ) !(@1)!                                       ...(137)
                  ::= %M <PARAM1> <ISUB⊥EXPRN> %M <PARAM1> !?93@2?91!           ...(138)
                  ::= %M <PARAM1> <REXPRN> %M <PARAM1> !?93@2?91!               ...(139)
<AOP1>            ::= <AOP2> !@1!                                               ...(140)
                  ::= / !/!                                                     ...(141)
<AOP2>            ::= + !+!                                                     ...(142)
                  ::= - !-!                                                     ...(143)
                  ::= * !*!                                                     ...(144)
<ISUB⊥EXPRN>      ::= <ISOPERAND> <RO⊥ISEXPRN> !@2@1!                           ...(145)
<RO⊥ISEXPRN>      ::= <AOP2> <ISOPERAND> <RO⊥ISEXPRN> !@3@2@1!                  ...(146)
                  ::= <NULL> !!                                                 ...(147)
<ISOPERAND>       ::= <INTEGER> !@1!                                            ...(148)
                  ::= <OQSVNAME> $20 !@1!                                       ...(149)
                  ::= ! <PARAM21> !A?11!                                        ...(150)
                  ::= ( <ISUB⊥EXPRN> ) !(@1)!                                   ...(151)
                  ::= %M <PARAM1> <ISUB⊥EXPRN> %M <PARAM1> !?93@2?91!           ...(152)
                  ::= R $140 <PARAM2> !0?11!                                    ...(153)
<LREXPRN>         ::= <LROPERAND> <RO⊥LREXPRN> $147 !@2@1!                      ...(154)
<RO⊥LREXPRN>      ::= <AOP1> <LROPERAND> <RO⊥LREXPRN> !@3@2@1!                  ...(155)
                  ::= %EXP <ISOPERAND> <RO⊥LREXPRN> !?26@2@1!                   ...(156)
                  ::= <NULL> !!                                                 ...(157)
<LROPERAND>       ::= Ø $145 <PARAM2> !A?11!                                    ...(158)
                  ::= R $146 <PARAM2> !A?11!                                    ...(159)
                  ::= <OQSVNAME> $4Ø !@1!                                       ...(160)
                  ::= ! <PARAM21> !A?11!                                        ...(161)
                  ::= ( <ISUB⊥EXPRN> ) !(@1)!                                   ...(162)
                  ::= ( <REXPRN> ) !(@1)!                                       ...(163)
                  ::= ( <LREXPRN> ) !(@1)!                                      ...(164)
                  ::= %M <PARAM1> <ISUB⊥EXPRN> %M <PARAM1> !?93@2?91!           ...(165)
                  ::= %M <PARAM1> <REXPRN> %M <PARAM1> !?93@2?91!               ...(166)
                  ::= %M <PARAM1> <LREXPRN> %M <PARAM1> !?93@2?91!              ...(167)
<OQSVNAME>        ::= <QUAL> <NAME> $22 <PARAM5> !@3@2?81!                      ...(168)
                  ::= <QUAL> <NAME1> $23 <ACT⊥PARMSØ> !@3@2@1!                  ...(169)
                  ::= <QUAL> <NAME2> $24 <SUBSLIST⊥BØ> !@3@2@1!                 ...(170)
                  ::= <NAME> $25 <PARAM5> !@2?81!                               ...(171)
                  ::= <NAME1> $26 <PARAM5> <ACT⊥PARMSØ> !@3?17?82@1!            ...(172)
```

```
                    ::= <TFN> $29 ( <REXPRN> ) $27 !@2@1!                          ...(173)
                    ::= <TFN> $29 ( <LREXPRN> ) $28 !@2@1!                         ...(174)
                    ::= <NAME2> $3Ø <ACT⊥PARMSØ> !@2@1!                            ...(175)
                    ::= <ANAME> ( $31 <SUBSLIST> ) $39 !@2?33@1?34!                ...(176)
<QUAL>              ::= <NAME> $32 <SUBSLIST⊥BØ> ⊥ <REC⊥SEQØ> !@3@2⊥@1!            ...(177)
                    ::= <NAME1> $33 <PARAM7> ⊥ <REC⊥SEQØ> !@3?25A?12⊥@1!           ...(178)
<REC⊥SEQØ>          ::= <NAME> $34 <PARAM7> ⊥ <REC⊥SEQØ> !@3?25A?12⊥@1!            ...(179)
                    ::= <NULL> !!                                                  ...(18Ø)
<ACT⊥PARMSØ>        ::= ( <DUM1> $35 <PARM> <RO⊥PARMS> $43 ) !(@2@1)!              ...(181)
                    ::= ( $41 <TEXPRN> <RO⊥PARMS> $43 ) !(@2@1)!                   ...(182)
                    ::= <NULL> !!                                                  ...(183)
<PARM>              ::= <DUM1> $37 <DUM2> $42 <QUAL> <NAME> <SUBSLIST⊥BØ> !?24@3@2@1!  ...(184)
                    ::= <DUM1> $37 <NAME> <SUBSLIST⊥BØ> !?24@2@1!                  ...(185)
                    ::= <DUM2> $42 <QUAL> <NAME> <SUBSLIST⊥BØ> !@3@2@1!            ...(186)
                    ::= <NAME> <SUBSLIST⊥BØ> !@2@1!                                ...(187)
<RO⊥PARMS>          ::= , <DUM1> $36 <PARM> <RO⊥PARMS> !,@2@1!                     ...(188)
                    ::= , $41 <TEXPRN> <RO⊥PARMS> !,@2@1!                          ...(189)
                    ::= <NULL> !!                                                  ...(19Ø)
<TEXPRN>            ::= <DUM1> $15Ø <PARAM153> <IEXPRN> !?82@1!                    ...(191)
                    ::= <DUM2> $151 <REXPRN> !?22@1!                               ...(192)
                    ::= <DUM2> $151 <LREXPRN> !?23@1!                              ...(193)
                    ::= <DUM3> $152 <LREXPRN> !?23@1!                              ...(194)
                    ::= <PARAM154> <STR⊥EXPRN> !?82@1!                             ...(195)
<SUBSLIST>          ::= <IEXPRN> $38 , <SUBSLIST> !@2,@1!                          ...(196)
                    ::= <IEXPRN> $38 !@1!                                          ...(197)
<SUBSLIST⊥BØ>       ::= ( <SUBSLIST⊥B> ) !?33@1?34!                                ...(198)
                    ::= <NULL> !!                                                  ...(199)
<SUBSLIST⊥B>        ::= <IEXPRN> , <SUBSLIST⊥B> !@2,@1!                            ...(2ØØ)
                    ::= <IEXPRN> !@1!                                              ...(2Ø1)
<INITIALØ>          ::= = <SGND⊥CONST> !=@1!                                       ...(2Ø2)
                    ::= <DUM1> $52 !=Ø?39!                                         ...(2Ø3)
                    ::= <DUM2> !=!?4Ø!                                             ...(2Ø4)
<INITLISTØ>         ::= = <ILIST> !=@1!                                            ...(2Ø5)
                    ::= <DUM1> $52 <PARAM51> !=Ø?39(Ø?11)!                         ...(2Ø6)
                    ::= <PARAM51> !=!?4Ø(Ø?11)!                                    ...(2Ø7)
<ILIST>             ::= <SGND⊥CONST> <QUALIFIERØ> , <ILIST> !@3@2,@1!              ...(2Ø8)
                    ::= <SGND⊥CONST> <QUALIFIERØ> !@2@1!                           ...(2Ø9)
<SGND⊥CONST>        ::= <INTEGER> !@1!                                             ...(21Ø)
```

```
                 ::= <MINUS0> R <PARAM4> !@2R?11!                                      ...(211)
                 ::= <NOT0> <DUM1> $52 ! <PARAM4> !@2M?11!                             ...(212)
                 ::= <NOT0> <DUM2> ' <PARAM4> !@2'?11                                  ...(213)
<MINUS0>         ::= - !-!                                                             ...(214)
                 ::= <NULL> !!                                                         ...(215)
<NOT0>           ::= \ !\!                                                             ...(216)
                 ::= <MINUS0> !@1!                                                     ...(217)
<QUALIFIER0>     ::= ( <INTEGER> ) !(@1)!                                              ...(218)
                 ::= <NULL> !!                                                         ...(219)
<SWITCHLIST>     ::= <NAMELIST> ( <INTEGER> $49 : <INTEGER> $50 ) $17 , <SWITCHLIST>
                     !@4(@3:@2),@1!                                                    ...(220)
                 ::= <NAMELIST> ( <INTEGER> $49 : <INTEGER> $50 ) $17 !@3(@2:@1)!      ...(221)
<RFDEFN>         ::= <RFDEC> <RFDEFN> !@2@1!                                           ...(222)
                 ::= <RFDEC> !@1!                                                      ...(223)
<RFDEC>          ::= <TYPE> <NAMELIST> $54 !@2@1!                                      ...(224)
                 ::= <TYPE> %ARRAY <PARAM1> <ARRAYLIST> $59 !@3?92@1!                  ...(225)
                 ::= <TYPE> %NAME <PARAM1> $44 <NAMELIST> $54 !@3?92@1!                ...(226)
                 ::= %RECORD <PARAM1> $55 %NAME <PARAM1> $56 <NAMELIST> $54 !?93?92@1! ...(227)
<RESLN_HEAD>     ::= %RSLV <PARAM1> <RESLN_HEAD> , <OQSVNAME> $40 . ( <STR_EXPRN> )
                     !?94@3.@2.(@1)!                                                   ...(228)
                 ::= <OQSVNAME> $40 %GOTO <PARAM1> <RHEAD0> ( <STR_EXPRN> )
                     !@4?93@2(@1)!                                                     ...(229)
<RTAIL0>         ::= . <OQSVNAME> $40 !.@1!                                            ...(230)
                 ::= <NULL> !!                                                         ...(231)
<RHEAD0>         ::= <OQSVNAME> . $40 !@1.!                                            ...(232)
                 ::= <NULL> !!                                                         ...(233)
<FPSPFN0>        ::= ( $46 <FPLIST> $68 ) !(@1)!                                       ...(234)
                 ::= <DUM1> $69 !!                                                     ...(235)
<FPLIST>         ::= <FPDELIM> <NAME> $70 <FPLIST> !@3@2@1!                            ...(236)
                 ::= <FPDELIM> <NAME> $70 !@2@1!                                       ...(237)
<FPDELIM>        ::= <RT> !@1!                                                         ...(238)
                 ::= <RTYPE> <ARRAY0> %NAME <PARAM1> $44 !@3@2?91!                     ...(239)
                 ::= <TYPE> !@1!                                                       ...(240)
                 ::= <ARRAY0> %NAME <PARAM1> $71 !@2?91!                               ...(241)
<RTYPE>          ::= %RECORD <PARAM1> $55 !?91!                                        ...(242)
                 ::= <TYPE> !@1!                                                       ...(243)
<EXTRN0>         ::= %EXTERNAL <PARAM1> !?91!                                          ...(244)
                 ::= <SYS0> !@1!                                                       ...(245)
```

```
<SYS0>          ::= %SYSTEM <PARAM1> ¡?91!                                        ...(246)
                ::= <NULL> ¡!                                                     ...(247)
<RT>            ::= %ROUTINE <PARAM1> $72 ¡?91!                                   ...(248)
                ::= <TYPE> %FN <PARAM1> $73 ¡@2?91!                               ...(249)
                ::= <TYPE> %MAP <PARAM1> $74 ¡@2?91!                              ...(250)
<ARRAY0>        ::= %ARRAY <PARAM1> $75 ¡?91!                                     ...(251)
                ::= <NULL> ¡!                                                     ...(252)
<FPDEFN0>       ::= ( <FPCOMPS> ) ¡(@1)!                                          ...(253)
                ::= <NULL> ¡!                                                     ...(254)
<FPCOMPS>       ::= <FPDELIM> <NAME> $77 <FPCOMPS> ¡@3@2@1!                       ...(255)
                ::= <FPDELIM> <NAME> $77 ¡@2@1!                                   ...(256)
<FPSPFN20>      ::= ( $78 <FPLIST> $68 ) ¡(@1)!                                   ...(257)
                ::= <DUM1> $79 ¡!                                                 ...(258)
<STR⊥EXPRN>     ::= <SOPERAND> <RO⊥STR⊥EXPRN> <PARAM80> ¡?43@3@2!                 ...(259)
<RO⊥STR⊥EXPRN>  ::= . $81 <SOPERAND> <RO⊥STR⊥EXPRN> ¡.@2@1!                       ...(260)
                ::= <NULL> ¡!                                                     ...(261)
<SOPERAND>      ::= <OQSVNAME> $40 ¡@1!                                           ...(262)
                ::= ! <PARAM82> ¡A?11!                                            ...(263)
<QUAL2>         ::= <NAME> $32 <SUBSLIST⊥B0> ⊥ <REC⊥SEQB0> ¡@3@2⊥@1!              ...(264)
                ::= <NAME> $91 <PARAM7> ⊥ <REC⊥SEQB0> ¡@3?25A?12⊥@1!              ...(265)
<REC⊥SEQB0>     ::= <NAME> $92 <PARAM7> ⊥ <REC⊥SEQB0> ¡@3?25A?12⊥@1!              ...(266)
                ::= <NULL> ¡!                                                     ...(267)
<FAULTLIST>     ::= <NLIST> %GOTO <PARAM1> <NAME> , <FAULTLIST> ¡@4?93@2,@1!      ...(268)
                ::= <NLIST> %GOTO <PARAM1> <NAME> ¡@3?92@1!                       ...(269)
<NLIST>         ::= <INTEGER> , <NLIST> ¡@2,@1!                                   ...(270)
                ::= <INTEGER> ¡@1!                                                ...(271)
<COMP>          ::= £ ¡£!                                                         ...(272)
                ::= = ¡=!                                                         ...(273)
                ::= > = ¡>=!                                                      ...(274)
                ::= < = ¡<=!                                                      ...(275)
                ::= < ¡<!                                                         ...(276)
                ::= > ¡>!                                                         ...(277)
<PARAM1>        ::= $1 ¡!                                                         ...(278)
<PARAM2>        ::= $2 ¡!                                                         ...(279)
<PARAM3>        ::= $3 ¡!                                                         ...(280)
<PARAM4>        ::= $4 ¡!                                                         ...(281)
<PARAM5>        ::= $5 ¡!                                                         ...(282)
<PARAM6>        ::= $6 ¡!                                                         ...(283)
```

```
<PARAM7>       ::= $7 !!                                          ...(284)
<PARAM8>       ::= $8 !!                                          ...(285)
<PARAM9>       ::= $9 !!                                          ...(286)
<PARAM21>      ::= $21 !!                                         ...(287)
<PARAM51>      ::= $51 !!                                         ...(288)
<PARAM80>      ::= $80 !!                                         ...(289)
<PARAM82>      ::= $82 !!                                         ...(290)
<PARAM96>      ::= $96 !!                                         ...(291)
<PARAM102>     ::= $102 !!                                        ...(292)
<PARAM110>     ::= $110 !!                                        ...(293)
<PARAM153>     ::= $153 !!                                        ...(294)
<PARAM154>     ::= $154 !!                                        ...(295)
<DUM1>         ::= <NULL> !!                                      ...(296)
<DUM2>         ::= <NULL> !!                                      ...(297)
<DUM3>         ::= <NULL> !!                                      ...(298)
<NAME1>        ::= <NAME> !@1!                                    ...(299)
<NAME2>        ::= <NAME> !@1!                                    ...(300)
<NAME3>        ::= <NAME> !@1!                                    ...(301)
<ANAME>        ::= <NAME> !@1!                                    ...(302)
<TFN>          ::= <NAME> !@1!                                    ...(303)
```

SYNTAX DEFINITIONS OF IMP(INT)

Definitions (1) to (55) correspond exactly to those given in appendix (3) with the exception that ⟨S14⟩ is no longer an alternative, having been reinterpreted in the form ⟨S5⟩.

```
<S1>          ::= <TYPE> <NAMELIST> ;                                             ...( 36)
              ::= <TYPE> %ARRAY <ARRAYLIST> ;                                     ...( 37)
<S2>          ::= <TYPE> %NAME <NAMELIST> ;                                       ...( 38)
<S4>          ::= %LIBRARY <RT> %SPEC <NAME> <FPSPFN0> ;                          ...( 39)
              ::= %EXTERNAL <RT> %SPEC <NAME> <FPSPFN0> ;                         ...( 40)
<S5>          ::= <SYS0> <RT> %SPEC <NAME> <FPSPFN0> ;                            ...( 41)
<S6>          ::= <EXTRN0> <RT> <NAME> <FPDEFN0> ;                                ...( 42)
<S7>          ::= %OWN <TYPE> <NAME> = <SGND_CONST> ;                             ...( 44)
<S8>          ::= %OWN <TYPE> %ARRAY <NAME> ( <INTEGER> : <INTEGER> ) = <ILIST> ;  ...( 45)
<S9>          ::= %SWITCH <SWITCHLIST> ;                                          ...( 46)
<S10>         ::= %RECORD %FORMAT <NAME> ( <RFDEFN> ) ;                           ...( 47)
<S11>         ::= %RECORD <NAMELIST> ( <NAME> ) ;                                 ...( 48)
              ::= %RECORD %ARRAY <ARRAYLIST> ( <NAME> ) ;                         ...( 49)
<S12>         ::= %RECORD %NAME <NAMELIST> ( <NAME> ) ;                           ...( 50)
<S13>         ::= %RECORD %SPEC <NAME> _ <NAME> ( <NAME> ) ;                      ...( 51)
              ::= %RECORD %SPEC <NAME> ( <NAME> ) ;                               ...( 52)
<S15>         ::= %BEGIN ;                                                        ...( 54)
<S16>         ::= %END <MARK0> ;                                                  ...( 55)
<S17>         ::= %END <MARK0> %OFPROGRAM ;                                       ...( 56)
              ::= %END <MARK0> %OFFILE ;                                          ...( 57)
<S18>         ::= <COMMENT_TEXT> ;                                                ...( 58)
<S19>         ::= %IF <UBSC> %THEN %START ;                                       ...( 59)
<S20>         ::= %IF <RESLN_HEAD> <RTAIL0> %THEN %START ;                        ...( 61)
<S22>         ::= %FINISH ;                                                       ...( 62)
<S23>         ::= %CYCLE <OQSVNAME> = <IEXPRN> , <IEXPRN> , <IEXPRN> ;            ...( 63)
<S24>         ::= %REPEAT ;                                                       ...( 64)
<S25>         ::= %FAULT <FAULTLIST> ;                                            ...( 65)
<S26>         ::= <NAME> :                                                        ...( 66)
<S27>         ::= <NAME> ( <INTEGER> ) :                                          ...( 68)
<S28>         ::= <RESLN_HEAD> <RTAIL0> ;                                         ...( 69)
<S29>         ::= <QUALIFICATION0> <NAME> <TYPE_MARK> %PTS <OQSVNAME> ;           ...( 70)
<S30>         ::= <OQSVNAME> = <TYPED_EXPRN1> ;                                   ...( 74)
              ::= <OQSVNAME> %FRMSTR <STRING_LENGTH> <STR_EXPRN> ;                ...( 75)
              ::= <OQSVNAME> %FRMBYT <IEXPRN> ;                                   ...( 76)
              ::= <OQSVNAME> %FRMSHT <IEXPRN> ;                                   ...( 77)
              ::= <OQSVNAME> %FRM <TYPED_EXPRN2> ;                                ...( 78)
<S31>         ::= <NAME> <ACTUAL_PARMS0> ;                                        ...( 79)
```

```
<S32>           ::= %GOTO <NAME> ;                                              ...( 8Ø)
<S33>           ::= %GOTO <NAME> %LB <IEXPRN> %RB ;                             ...( 82)
<S35>           ::= %RETURN ;                                                   ...( 83)
<S36>           ::= %RESULT = <TYPED_EXPRN1> ;                                  ...( 84)
                ::= %RESULT = %MP <TYPED_EXPRN1> ;                              ...( 85)
<S37>           ::= %STOP ;                                                     ...( 86)
<S38>           ::= %MONITOR <INTEGER> ;                                        ...( 87)
                ::= %MONITOR ;                                                  ...( 88)
<TYPE>          ::= %BYTE %INTEGER                                              ...( 89)
                ::= %SHORT %INTEGER                                             ...( 9Ø)
                ::= %INTEGER                                                    ...( 91)
                ::= %LONG %REAL                                                 ...( 92)
                ::= %REAL                                                       ...( 93)
                ::= %STRING ( <STRING_LENGTH> )                                 ...( 94)
                ::= %STRING                                                     ...( 95)
<NAMELIST>      ::= <NAME> , <NAMELIST>                                         ...( 96)
                ::= <NAME>                                                      ...( 97)
<ARRAYLIST>     ::= <NAMELIST> ( <BPLIST> ) , <ARRAYLIST>                       ...(1Ø1)
                ::= <NAMELIST> ( <BPLIST> )                                     ...(1Ø2)
<BPLIST>        ::= <IEXPRN> : <IEXPRN> , <BPLIST>                              ...(1Ø3)
                ::= <IEXPRN> : <IEXPRN>                                         ...(1Ø4)
<IEXPRN>        ::= <IOPERAND> <RO_IEXPRN>                                      ...(1Ø5)
<REXPRN>        ::= <ROPERAND> <RO_REXPRN>                                      ...(128)
<RO_REXPRN>     ::= <AOP1> <ROPERAND> <RO_REXPRN>                               ...(129)
                ::= %RXP <ROPERAND> <RO_REXPRN>                                 ...(13Ø)
                ::= <NULL>                                                      ...(131)
<ROPERAND>      ::= <OQSVNAME>                                                  ...(134)
                ::= ( <ISUB_EXPRN> )                                            ...(136)
                ::= ( <REXPRN> )                                                ...(137)
                ::= %M <ISUB_EXPRN> %M                                          ...(138)
                ::= %M <REXPRN> %M                                              ...(139)
<AOP1>          ::= <AOP2>                                                      ...(14Ø)
                ::= /                                                           ...(141)
<AOP2>          ::= +                                                           ...(142)
                ::= -                                                           ...(143)
                ::= *                                                           ...(144)
<ISUB_EXPRN>    ::= <ISOPERAND> <RO_ISEXPRN>                                    ...(145)
<RO_ISEXPRN>    ::= <AOP2> <ISOPERAND> <RO_ISEXPRN>                             ...(146)
```

```
               ::= <NULL>                                                             ...(147)
<ISOPERAND>    ::= <INTEGER>                                                          ...(148)
               ::= <OQSVNAME>                                                         ...(149)
               ::= ( <ISUB_EXPRN> )                                                   ...(151)
               ::= %M <ISUB_EXPRN> %M                                                 ...(152)
<LREXPRN>      ::= <LROPERAND> <RO_LREXPRN>                                           ...(154)
<RO_LREXPRN>   ::= <AOP1> <LROPERAND> <RO_LREXPRN>                                    ...(155)
               ::= %DXP <LROPERAND> <LROPERAND> <RO_LREXPRN>                          ...(156)
               ::= <NULL>                                                             ...(157)
<LROPERAND>    ::= <OQSVNAME>                                                         ...(160)
               ::= ( <ISUB_EXPRN> )                                                   ...(162)
               ::= ( <REXPRN> )                                                       ...(163)
               ::= ( <LREXPRN> )                                                      ...(164)
               ::= %M <ISUB_EXPRN> %M                                                 ...(165)
               ::= %M <REXPRN> %M                                                     ...(166)
               ::= %M <LREXPRN> %M                                                    ...(167)
<SUBSLIST>     ::= <IEXPRN> , <SUBSLIST>                                              ...(196)
               ::= <IEXPRN>                                                           ...(197)
<ILIST>        ::= <SGND_CONST> <QUALIFIERØ> , <ILIST>                                ...(208)
               ::= <SGND_CONST> <QUALIFIERØ>                                          ...(209)
<SGND_CONST>   ::= <INTEGER>                                                          ...(210)
               ::= <MINUSØ> <REAL_CONST>                                              ...(211)
               ::= <NOTØ> <STRING>                                                    ...(212)
<MINUSØ>       ::= -                                                                  ...(213)
               ::= <NULL>                                                             ...(214)
<NOTØ>         ::= \                                                                  ...(215)
               ::= <MINUSØ>                                                           ...(216)
<QUALIFIERØ>   ::= ( <INTEGER> )                                                      ...(217)
               ::= <NULL>                                                             ...(218)
<SWITCHLIST>   ::= <NAMELIST> ( <INTEGER> : <INTEGER> ) , <SWITCHLIST>                ...(219)
               ::= <NAMELIST> ( <INTEGER> : <INTEGER> )                               ...(220)
<RFDEFN>       ::= <RFDEC> <RFDEFN>                                                   ...(221)
               ::= <RFDEC>                                                            ...(222)
<RFDEC>        ::= <TYPE> <NAMELIST>                                                  ...(223)
               ::= <TYPE> %ARRAY <ARRAYLIST>                                          ...(224)
               ::= <TYPE> %NAME <NAMELIST>                                            ...(225)
               ::= %RECORD %NAME <NAMELIST>                                           ...(226)
<RESLN_HEAD>   ::= %RSLV <RESLN_HEAD> . <OQSVNAME> . ( <STR_EXPRN> )                  ...(227)
```

```
                ::= <OQSVNAME> %GOTO <RHEADØ> ( <STR_EXPRN> )                   ...(228)
<RTAILØ>        ::= , <OQSVNAME>                                                ...(229)
                ::= <NULL>                                                      ...(230)
<RHEADØ>        ::= <OQSVNAME> .                                                ...(231)
                ::= <NULL>                                                      ...(232)
<FPSPFNØ>       ::= ( <FPLIST> )                                                ...(233)
                ::= <NULL>                                                      ...(234)
<FPLIST>        ::= <FPDELIM> <NAME> <FPLIST>                                   ...(235)
                ::= <FPDELIM> <NAME>                                            ...(236)
<FPDELIM>       ::= <RT>                                                        ...(237)
                ::= <RTYPE> <ARRAYØ> %NAME                                      ...(238)
                ::= <TYPE>                                                      ...(239)
                ::= <ARRAYØ> %NAME                                              ...(240)
<RTYPE>         ::= %RECORD                                                     ...(241)
                ::= <TYPE>                                                      ...(242)
<EXTRNØ>        ::= %EXTERNAL                                                   ...(243)
                ::= <SYSØ>                                                      ...(244)
<SYSØ>          ::= %SYSTEM                                                     ...(245)
                ::= <NULL>                                                      ...(246)
<RT>            ::= %ROUTINE                                                    ...(247)
                ::= <TYPE> %FN                                                  ...(248)
                ::= <TYPE> %MAP                                                 ...(249)
<ARRAYØ>        ::= %ARRAY                                                      ...(250)
                ::= <NULL>                                                      ...(251)
<FPDEFNØ>       ::= <FPSPFNØ>                                                   ...(252)
<STR_EXPRN>     ::= %CNCT <STR_EXPRN> , <OQSVNAME>                              ...(258)
                ::= <OQSVNAME>                                                  ...(259)
<FAULTLIST>     ::= <NLIST> %GOTO <NAME> <FAULTLIST>                            ...(267)
                ::= <NLIST> %GOTO <NAME>                                        ...(268)
<NLIST>         ::= <INTEGER> , <NLIST>                                         ...(269)
                ::= <INTEGER>                                                   ...(270)
<IOPERAND>      ::= <OQSVNAME>                                                  ...(303)
                ::= <INTEGER>                                                   ...(304)
                ::= ( <IEXPRN> )                                                ...(305)
                ::= %L <LOG_EXPRN> %R                                           ...(306)
                ::= %M <IEXPRN> %M                                              ...(307)
<RO_IEXPRN>     ::= <AOP2> <IEXPRN> <RO_IEXPRN>                                 ...(308)
                ::= %IXP <IEXPRN> <RO_IEXPRN>                                   ...(309)
```

```
                  ::= <NULL>                                                    ...(310)
<LOG_EXPRN>       ::= <IOPERAND> <RO_LOG_EXPRN>                                 ...(311)
<RO_LOG_EXPRN>    ::= <LOG_OP> <IOPERAND> <RO_LOG_EXPRN>                        ...(312)
                  ::= <NULL>                                                    ...(313)
<LOG_OP>          ::= %NEV                                                      ...(314)
                  ::= !                                                         ...(315)
                  ::= &                                                         ...(316)
                  ::= %BY                                                       ...(317)
                  ::= %LSH                                                      ...(318)
                  ::= %RSH                                                      ...(319)
<OQSVNAME>        ::= <NAME> %MP <TYPE_MARK2> <ACTUAL_PARMS0>                   ...(320)
                  ::= <QUALIFICATION0> <NAME> <TYPE_MARK2>                      ...(321)
                  ::= <QUALIFICATION0> <NAME> %LB <SUBS_LIST> %RB               ...(322)
                  ::= <QUALIFICATION0> <NAME> <ACTUAL_PARMS0>                   ...(323)
<QUALIFICATION0>
                  ::= <NAME> %LB <SUBS_LIST> %RB _ <REC_SEQ0>                   ...(324)
                  ::= <NAME> _ <REC_SEQ0>                                       ...(325)
                  ::= <REC_SEQ0>                                                ...(326)
<REC_SEQ0>        ::= <NAME> %REC <NAME> _ <REC_SEQ0>                           ...(327)
                  ::= <NULL>                                                    ...(328)
<ACTUAL_PARMS0>::= ( <PARM> <RO_PARMS> )                                        ...(329)
                  ::= <NULL>                                                    ...(330)
<PARM>            ::= <OQSVNAME>                                                ...(331)
                  ::= %NM <OQSVNAME>                                            ...(332)
                  ::= <TYPED_EXPRN1>                                            ...(333)
<RO_PARMS>        ::= , <PARM> <RO_PARMS>                                       ...(334)
                  ::= <NULL>                                                    ...(335)
<TYPED_EXPRN1> ::= %BYT <IEXPRN>                                                ...(336)
                  ::= %SHT <IEXPRN>                                             ...(337)
                  ::= %STR <STRING_LENGTH> <STR_EXPRN>                          ...(338)
                  ::= <TYPED_EXPRN2>                                            ...(339)
<TYPED_EXPRN2> ::= %INT <IEXPRN>                                                ...(340)
                  ::= %RL <REXPRN>                                              ...(341)
                  ::= %LNG <LREXPRN>                                            ...(342)
<TYPE_MARK>       ::= %REC <NAME>                                               ...(343)
                  ::= <TYPE_MARK2>                                              ...(344)
<TYPE_MARK2>      ::= %BYT                                                      ...(345)
                  ::= %SHT                                                      ...(346)
```

```
                ::= %INT                                                    ...(347)
                ::= %RL                                                     ...(348)
                ::= %LNG                                                    ...(349)
                ::= %STR <STRING_LENGTH>                                    ...(350)
<UBSC>          ::= <CEXPRN> <COMP> <CEXPRN> <COMP> <CEXPRN>                ...(351)
                ::= <CEXPRN> <COMP> <CEXPRN>                                ...(352)
<CEXPRN>        ::= <TYPED_EXPRN2>                                          ...(353)
                ::= %STR <STRING_LENGTH> <STR_EXPRN>                        ...(354)
<MARK0>         ::= %MP                                                     ...(355)
                ::= <TYPE_MARK2>                                            ...(356)
                ::= <NULL>                                                  ...(357)
<COMP>          ::= £                                                       ...(358)
                ::= =                                                       ...(359)
                ::= > =                                                     ...(360)
                ::= < =                                                     ...(361)
                ::= > =                                                     ...(362)
                ::= <                                                       ...(363)
<STRING_LENGTH>::= <INTEGER>                                                ...(364)
```

Syntax Definitions for Typing Conditional Statements

```
<UBSC>          ::= <EXPRN> $10 <COMP> <EXPRN> $11 <RO⊥COND> !!          ...( 1)
<RO⊥COND>       ::= <COMP> <EXPRN> $12 <DUMMY> !!                        ...( 2)
                ::= <NULL> !!                                            ...( 3)
<COMP>          ::= £ !!                                                 ...( 4)
                ::= = !!                                                 ...( 5)
                ::= < = !!                                               ...( 6)
                ::= < !!                                                 ...( 7)
                ::= > = !!                                               ...( 8)
                ::= > !!                                                 ...( 9)
<EXPRN>         ::= $14 <OPERAND> <RO⊥EXPRN> !!                          ...( 10)
<RO⊥EXPRN>      ::= <OPERATOR> <OPERAND> <RO⊥EXPRN> !!                   ...( 11)
                ::= <NULL> !!                                            ...( 12)
<OPERAND>       ::= <OQSVNAME> $23 !!                                    ...( 13)
                ::= 0 $24 !!                                             ...( 14)
                ::= R $30 !!                                             ...( 15)
                ::= ' $26 !!                                             ...( 16)
                ::= ( <EXPRN> $27 ) !!                                   ...( 17)
                ::= %M <EXPRN> $27 %M !!                                 ...( 18)
<OPERATOR>      ::= * $20 !!                                             ...( 19)
                ::= / $25 !!                                             ...( 20)
                ::= %DIV $28 !!                                          ...( 21)
                ::= + $20 !!                                             ...( 22)
                ::= - $20 !!                                             ...( 23)
                ::= %EXP $25 !!                                          ...( 24)
                ::= ! $28 !!                                             ...( 25)
                ::= %NEV $28 !!                                          ...( 26)
                ::= & $28 !!                                             ...( 27)
                ::= . $29 !!                                             ...( 28)
                ::= %LSH $28 !!                                          ...( 29)
                ::= %RSH $28 !!                                          ...( 30)
<OQSVNAME>      ::= <QUAL> <NAME> $17 <ARG0> !!                          ...( 31)
                ::= <TFN> $18 ( <EXPRN> ) $19 !!                         ...( 32)
                ::= <NAME> $21 <ARG0> !!                                 ...( 33)
<QUAL>          ::= <NAME> $15 <ARG0> ⊥ <REC⊥SEQ0> !!                    ...( 34)
<REC⊥SEQ0>      ::= <NAME> $16 ⊥ <REC⊥SEQ0> !!                           ...( 35)
                ::= <NULL> !!                                            ...( 36)
<NAME>          ::= A $22 !!                                             ...( 37)
```

```
<TFN>            ::= A $22 !!                                           ...( 38)
<ARG0>           ::= ( <EXPRN2> <RO⊥ARG> ) !!                           ...( 39)
                 ::= <NULL> !!                                          ...( 40)
<RO⊥ARG>         ::= , <EXPRN2> <RO⊥ARG> !!                             ...( 41)
                 ::= <NULL> !!                                          ...( 42)
<DUMMY>          ::= <NULL> !!                                          ...( 43)
<EXPRN2>         ::= <OPERAND2> <RO⊥EXPRN2> !!                          ...( 44)
<RO⊥EXPRN2>      ::= <OP2> <OPERAND2> <RO⊥EXPRN2> !!                    ...( 45)
                 ::= <NULL> !!                                          ...( 46)
<OP2>            ::= * !!                                               ...( 47)
                 ::= / !!                                               ...( 48)
                 ::= + !!                                               ...( 49)
                 ::= - !!                                               ...( 50)
                 ::= %EXP !!                                            ...( 51)
                 ::= %DIV !!                                            ...( 52)
                 ::= ! !!                                               ...( 53)
                 ::= %NEV !!                                            ...( 54)
                 ::= & !!                                               ...( 55)
                 ::= . !!                                               ...( 56)
                 ::= %LSH !!                                            ...( 57)
                 ::= %RSH !!                                            ...( 58)
<OPERAND2>       ::= 0 $13 !!                                           ...( 59)
                 ::= R $13 !!                                           ...( 60)
                 ::= ' $13 !!                                           ...( 61)
                 ::= <OQSVNAME2> !!                                     ...( 62)
                 ::= ( <EXPRN2> ) !!                                    ...( 63)
                 ::= %M <EXPRN2> %M !!                                  ...( 64)
<OQSVNAME2>      ::= <NAMEB> <ARG0> <RECORD0> !!                        ...( 65)
<NAMEB>          ::= A $13 !!                                           ...( 66)
<RECORD0>        ::= ⊥ <NAMEB> <RECORDPTR0> !!                          ...( 67)
                 ::= ⊥ <NAMEB> <ARG0> <RECORDPTR0> !!                   ...( 68)
                 ::= <NULL> !!                                          ...( 69)
<RECORDPTR0>     ::= ⊥ <NAMEB> <RECORDPTR0> !!                          ...( 70)
                 ::= <NULL> !!                                          ...( 71)
```

IMP(SYS) PHRASE STRUCTURES THAT ARE NOT TRANSLATED

The following syntax constructions are forbidden by this implementation of the translator because they are difficult to simulate in PL/1. They are identified during the second pass and rejected. The forbidden constructions are best discussed by reference to the syntax definitions listed in appendix (3).

27, 29 and 133

%ARRAY %NAME declarators are not permitted in this implementation of the translator; nor are %STRING %NAME (the length qualification must be included).

30

Routine pointers are forbidden.

31 and 206

The extended formal parameter definition (i.e. the use of type-general parameters) is only permitted in the context of %EXTERNAL routine specifications and the inclusion of type-general parameters in other contexts is faulted (by constructor-routine 01 writing an error code to the output stream).

55

%RECORD %ARRAY %NAME is not permitted in this implementation as a declarator. (This results in a form for the meta-variable <OQSVNAME> which is less general to that given in the current IMP language manual (ERCC 70).

236 and 237

The bounds of <CBPAIR> may only be defined by integers in this implementation.

292 to 297

A restricted class of record format delimiters is permitted in this implementation, consistent with the limitations in declarators already noted.

THE DESCRIPTOR

In the symbol tables, each identifier has associated with it a descriptor which describes the attributes of that identifier. The attributes are recorded by using a 56-bit field with the following structure:

1-3	4-11	12-14	15	16-17	18	19	20-24	25-56
TYPE	QUAL	DIM	AP	VP	USED	INTNC	ANO	AOFF

The meaning of each sub-field is described below under the appropriate heading.

TYPE

is set '000'B for type-general formal parameters

'001'B	"	%INTEGER	variables
'010'B	"	%REAL	"
'011'B	"	%STRING	"
'100'B	"	%RECORD	"
'101'B	"	%ROUTINE	"
'110'B	"	label	"

QUAL

is set '00000000'B to describe the arguments of transparent functions,

'00000001'B	to denote the identifier is stored in a	byte,
'00000010'B	"	half-word,
'00000100'B	"	full word,
'00001000'B	"	double word

location. When qualifying a %STRING variable, QUAL defines the string length (0 -> 255).

QUAL

indicates the dimensions of an identifier

set	dimension(s)
'000'B	0 (scalar)
'001'B	1
'010'B	2
'011'B	3
'100'B	4
'101'B	5
'110'B	6

AP

is set '1'B for pointer variables; '0'B for atoms.

VP

is set '01'B for functions; '10'B for map routines and '00'B for all other identifiers.

USED

is set to '1'B if the intrinsic or implicitly specified IMP library routine to which this descriptor relates is used in the IMP program being processed; else set to '1'B.

INTNC

is set to '1'B for all intrinsic and implicitly specified IMP library routines; else set to '0'B.

ANO and AOFF

are set to (5)'0'B and (32)'0'B for all but records, routines, maps and functions, when they point to the start of lists which describe the elements of a record or parameters.

The general significance of ANO and AOFF is described in section (4.3.1.2). ANO is an (unsigned) index to an array of pointers, AOFF is the bit representation of an address.

THE FOURTH PASS

The fourth pass reads the IMP(INT) text output from the third pass and translates it to PL/1. At the time of writing this report, the pass was in the process of being written by T. Nonweiler of the Glasgow University Aeronautics Department.

The structure of the pass is to be similar to that of the third pass. Using the syntax analyser and generating routine described in chapter 2., the IMP(INT) text is analysed and PL/1 text subsequently generated. The syntax definitions controlling this analysis together with the target language constructions producing the PL/1 text are listed in a report by Nonweiler (Non 72). The analysis requires a certain amount of context-sensitive analysis and the symbol tables produced in the third pass are referenced. At the moment, the translation uses 9 analytic-routines and about 35 constructor-routines. Use is made of the PL/1 preprocessor to achieve the full translation to PL/1.

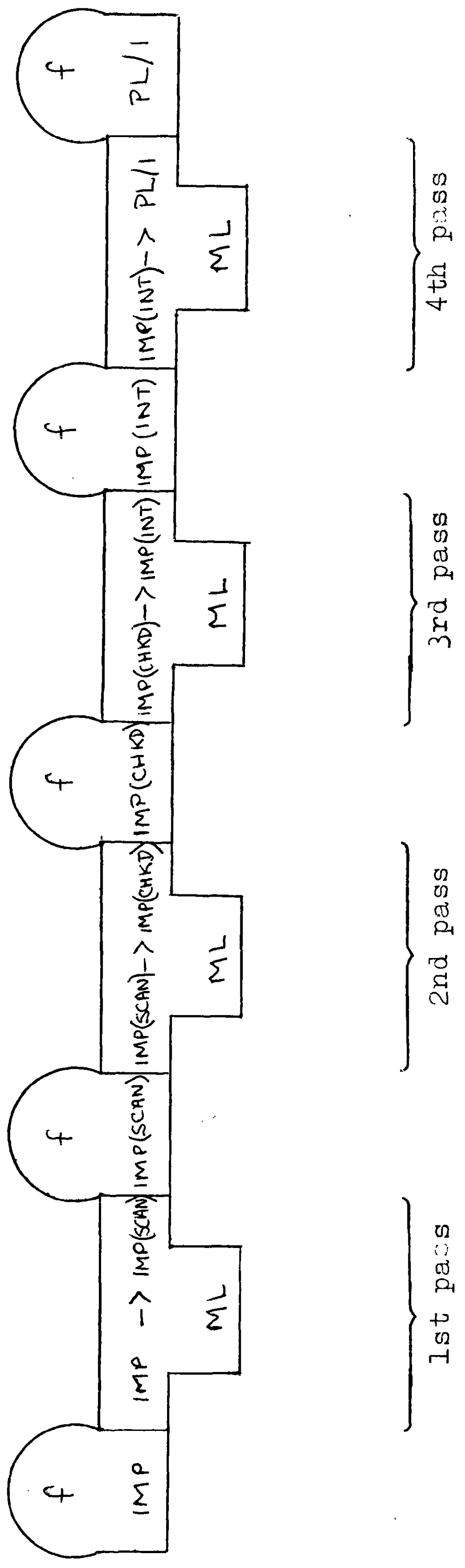
Nonweiler has written a program which will accept PL/1 programs and compose them in a form which is neat and readable. This program will be used to tidy the output from the fourth pass if so requested by a user. Alternatively the output may be compiled by the PL/1 preprocessor and compiler.

REFERENCES

Aho 72	AHO, A. V. and ULLMAN, J. D. The theory of parsing translation and compiling. Englewood Cliffs: Prentice Hall, 1972-1973.
Back 60	BACKUS, J. W. ET AL. Revised report on the algorithmic language ALGOL 60. Proc. International Conference on Information Processing.
Cap 73	CAPON, P. C. and ARGENT, G. D. Effective syntax analysis. Datafair 1973 Conference papers.
Don 67	DONOVAN, J. J. and LEDGARD, H. F. A formal system for the specification of the syntax and translation of computer languages. Proc AFIPS, 1967 Fall Joint Computer Conference pp 553-569.
Ear 70	EARLEY, J. and STURGIS, H. A formalism for translator interactions. CACM Vol 13 No. 10, Oct. 1970.
ERCC 70	Edinburgh IMP language manual. Edinburgh Regional Computing Centre July 1970.
Gries 68	GRIES, D. and FELDMAN, J. Translator writing systems. CACM Vol 11 No. 2 pp 77-113, 1968.
Gries 70	GRIES, D. Compiler construction for digital computers. John Wiley and Sons, Inc. 1970.
Hop 69	HOPGOOD, F. R. A. Compiling techniques. London: Macdonald. 1969.
IBM 68	IBM System/360 Conversion Aids. ALGOL-to-PL/1 language conversion program- User's Guide. IBM 1968.

IBM 68a	IBM System/360 Conversion Aids. ALGOL-to-PL/1 language conversion program - Program logic manual. Form Y33-7006-0. IBM 1968.
IBM 70	IBM System/360 Operating System PL/1 Language Reference Manual. File No. S360-29. IBM 1970.
Irons 70	IRONS, E. T. Experience with an extensible language. CACM Vol 13 No. 10, Jan 1970.
McE 67	McEWAN, A. T. An Atlas Autocode to Algol 60 translator. Computer Journal Vol 9 pp 353-359, Feb. 1967.
Mill 66	MILLARD, G. E. A syntax-directed compiling technique. RAE Technical Report No. 66154 - June 1966.
Mul 69	MULHOLLAND, K. A. Software to translate TELCOMP programs into KDF9 ALGOL. Computer Journal Vol 12 pp 221-224, Aug 1969.
Non 72	NONWEILER, T.R.F. A systematic translation of the IMP language into PL/1: An interim report. Inter University Research Councils Research and Development Notes. No. 3 Nov. 1972.

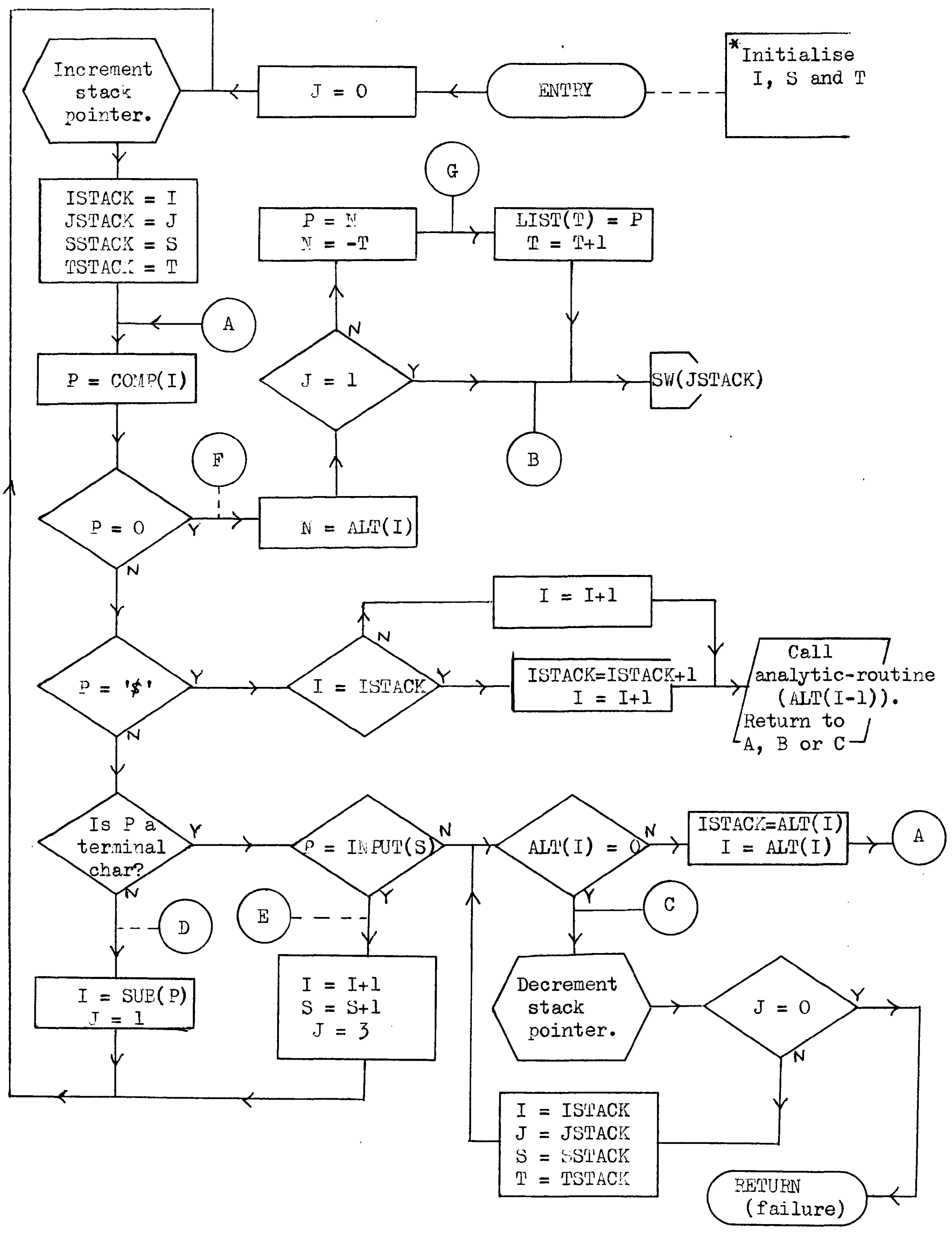
FIGURE 1.



THE IMP-to-PL/1 TRANSLATOR IN 'TOMBSTONE' NOTATION

FIGURE 2A

THE ANALYSER (First of two block diagrams)



THE ANALYSER (second of two diagrams)

Block diagram showing the build-up of the threaded list.

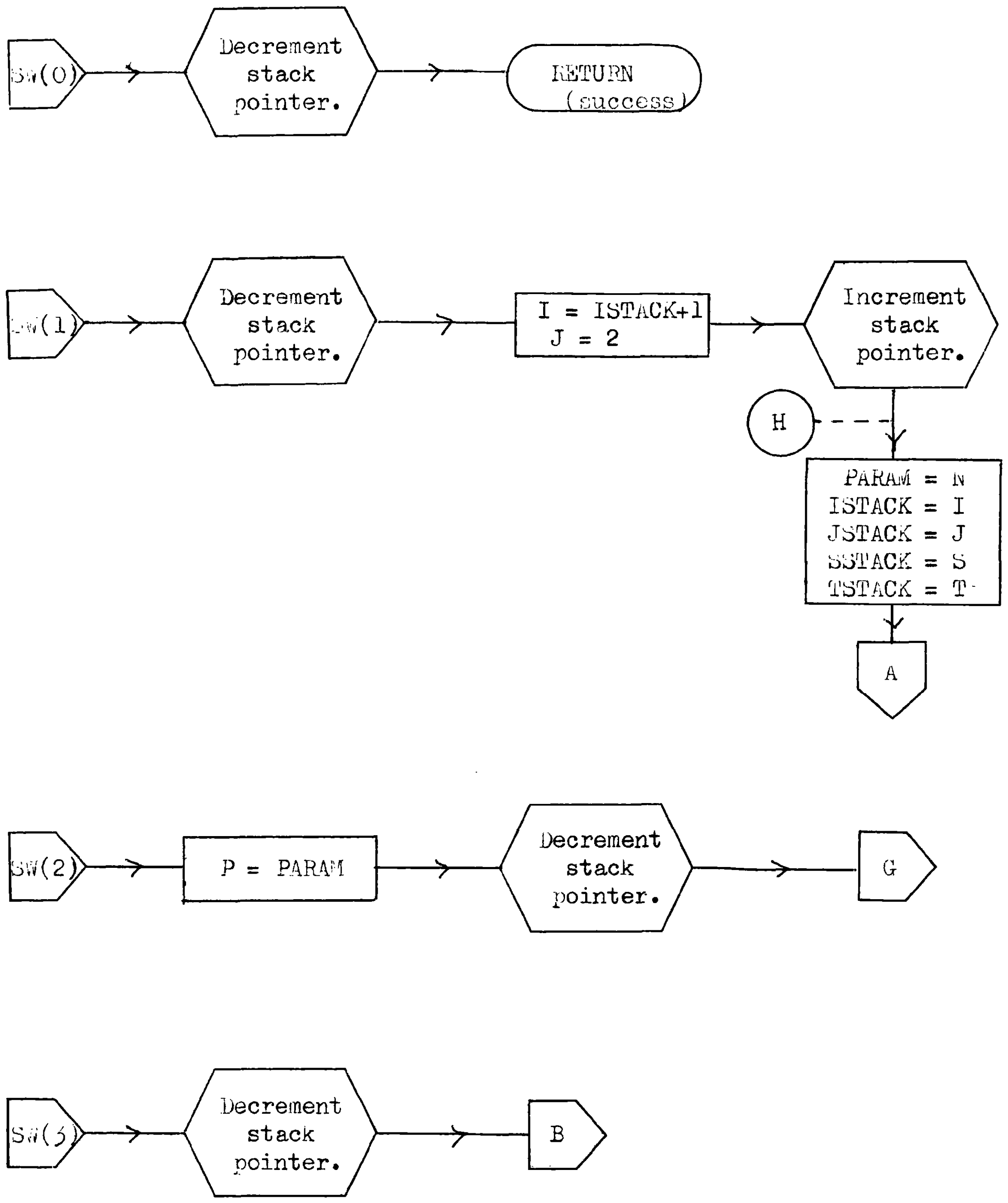
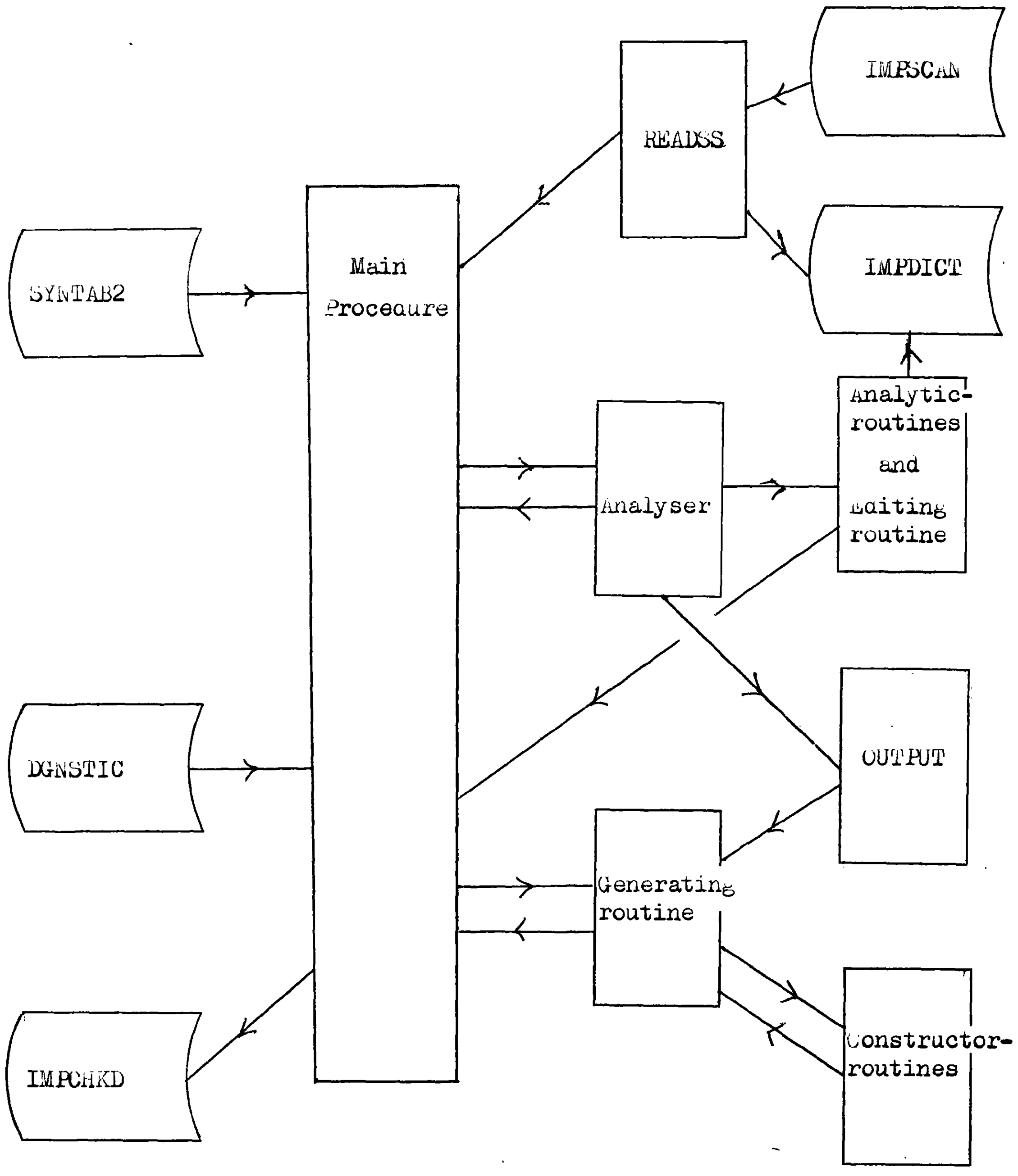
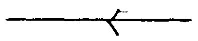


FIGURE 5.

THE SECOND PASS




Flow of data

THE EDITING ROUTINE

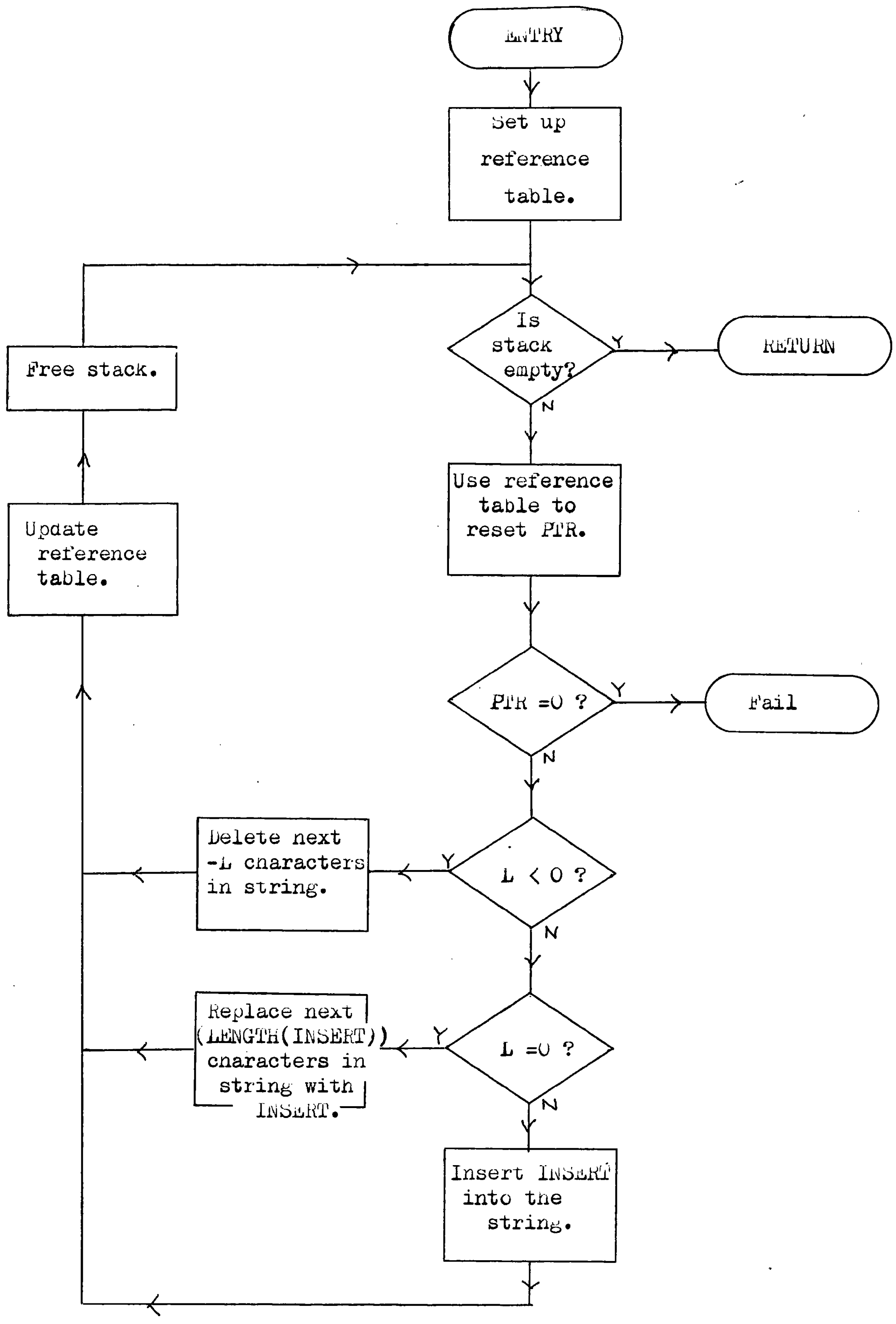
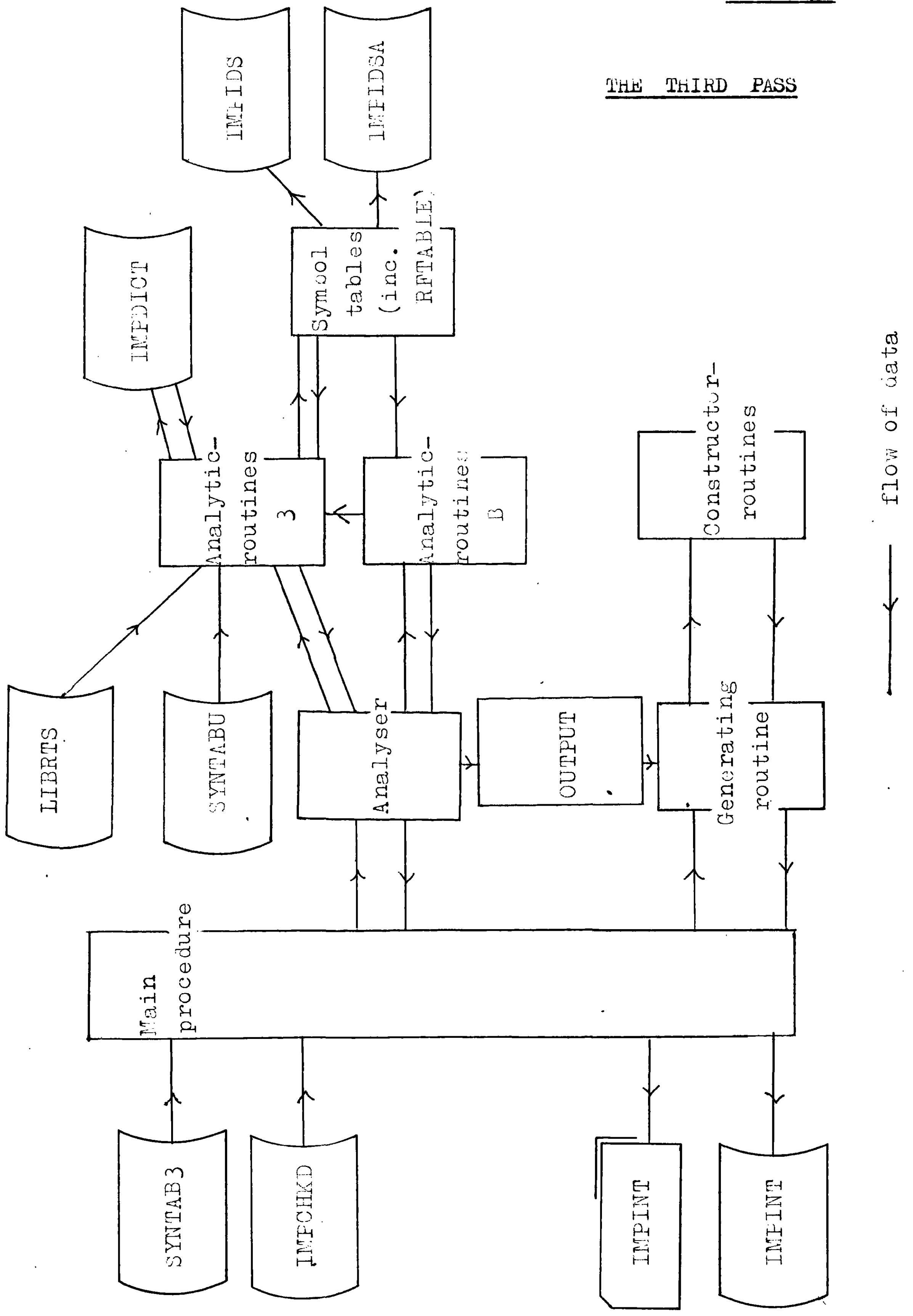

ENTRY
Set up reference table.
Is stack empty?
Y
N
RETURN
Free stack.
Use reference table to reset PTR.
Update reference table.
PTR =0 ?
Y
N
Fail
Delete next -L characters in string.
L < 0 ?
Y
N
Replace next (LENGTH(INSERT)) characters in string with INSERT.
L =0 ?
Y
N
Insert INSERT into the string.

FIGURE 5.

THE THIRD PASS



ABSTRACT

This thesis describes the author's contribution towards the development of a program (written in PL/1) by which a user's program written in a syntactically correct form of the IMP language (as implemented by the Edinburgh Regional Computing Centre) may be translated into a program of the PL/1 language; the implied intention being that the latter shall have in general the same effect upon it's computational environment when run under the control of an IBM system/360 operating system (that includes the PL/1 (F) compiler), as does the IMP source.

The translator makes four passes of the IMP source text. The first pass is concerned mainly with lexical analysis. The second and third passes translate the IMP source text into a form known here as IMP(INT). IMP(INT) is an internal form of a subset of the IMP language augmented by statement-description markers. It is not directly related to the generation of a PL/1 object text. The fourth pass translates the IMP(INT) text into PL/1.

It was the author's responsibility to program the second and third passes. Both of these passes (and the fourth pass) are controlled by a top-down syntax analyser. The methods used to analyse the syntax and perform most of the text transformations required during a pass are based on a syntax directed compiling technique noted by G. Millard. Various modifications and

improvements to Millard's original algorithms made by the author are described within.

During the second pass a text editor is used. This text editor was devised by the author and is also described within.

The third pass constructs symbol tables. These tables are referenced during the third and fourth passes. A description of the construction of these tables and the methods used to access them is given.

A number of programs designed to test all the characteristics of the translation to IMP(INT) have been successfully processed by the first, second and third passes. It is expected that the programming of the fourth pass will soon be completed, after which time an automatic translation from IMP to PL/1 will be possible.