TR80-62

THE AUTOMATIC GENERATION OF CODE GENERATORS WITH PARTICULAR REFERENCE TO COBOL

Thesis submitted in Fulfilment of the

Requirements for the Degree of

MASTER OF SCIENCE

Rhodes University

by

ALLAN ROY BULMER

December 1979

ACKNOWLEDGEMENTS

I wish to express my gratitude to my supervisor Prof. M.H. Williams who provided the idea for this project and assisted in the conceptual understanding of the method by means of which the system is able to propagate itself. The guidance I received during the project was greatly appreciated.

Also appreciated was the cooperation of the Officer in Charge of the Rhodes University Computer Centre, Mr. Martin Urry, particularly when the computing services were in heavy demand. In addition I wish to thank Dr. P.D. Terry for his encouragement.

Finally I acknowledge the financial assistance of I.C.L. and the C.S.I.R. from whom I received bursaries while this work was being carried out.

C O N T E N T S

1. INTRODUCTION 1

2. SYSTEM OVERVIEW 3

3. THE INVESTIGATION OF CODE GENERATOR REQUIREMENTS 6

3.1 GENERAL CODE GENERATOR FUNCTIONS 6

3.2 ICL 1900 SYSTEM COBOL CODE GENERATOR ACTIONS 7

3.3 CODE SKELETON ACTIONS 9

3.3.1 PSEUDO-CODE ACTION PARAMETERS 9

3.3.2 ADDRESS TABLE ACCESSES 9

3.3.3 EXPRESSIONS 10

3.3.4 RELOCATABLE INSTRUCTIONS 12

3.3.5 CONDITIONAL INSERTION OF CODE 13

3.3.6 FETCH, FREE AND USE REGISTERS 14

3.4 DEFINITION ACTIONS 17

3.4.1 DEFINE STORE 17

3.4.2 DEFINE CONSTANT 18

3.4.3 ALIGN 19

3.4.4 TRANSFER ADDRESS 19

3.4.5 DEFINE ADDRESS CONSTANT 21

3.4.6 DEFINE LOCAL ROUTINE 24

3.4.7 DEFINE ADDRESS TABLE ELEMENT 25

3.4.8 FINAL ACTION 26

3.4.9 STACK AND UNSTACK THE CURRENT ADDRESS 27

3.4.10 BRANCH FORWARD AND RESOLVE FORWARD BRANCH 29

3.5 THE INITIALIZATION ACTION 33

3.6 USER TRANSPARENT ACTIONS 33

3.7 FORMALIZATION OF THE CODE GENERATOR REQUIREMENTS 34

4. PSEUDO-CODE ACTION DEFINITION PROCESSING 36

4.1 INTRODUCTION 36

4.2 INPUT 36

4.2.1 THE OPCODE TO INTERNAL CODE MAPPING 36

4.2.2 THE PSEUDO-CODE ACTION DEFINITIONS 38

4.3 OUTPUT 38

4.4 PSEUDO-CODE ACTION TABLE OPTIMIZATION 39

4.5 THE PSEUDO-CODE ACTION DEFINITION PROCESSING PROGRAM 40

4.6 PROCESSED EXAMPLES 42

5. THE CREATION OF A COMPLETE SYSTEM 45

5.1 THE PRINCIPLE OF THE INTERPRETER OPERATION 45

5.1.1 AN OVERVIEW OF THE INTERPRETER 45

5.1.2 THE GENERAL SYSTEM PSEUDO-CODES 48

5.1.3 SYSTEM REQUIRED LOCAL ROUTINES 59

5.1.4 CODE GENERATOR PRODUCTION 63

5.2 IMPLEMENTATION FOR THE 1900 68

5.2.1 THE BOOTSTRAP PROGRAM 68

5.2.2 A LOADER FOR THE BOOTSTRAPPED CODE GENERATOR GENERATOR 70

5.2.3 TESTING THE CODE GENERATOR GENERATOR 73

6. THE COBOL COMPILER - PART I FORMAL SPECIFICATION AND THE DATA DIVISION 77

6.1 INTRODUCTION 77

6.2 THE GENERAL IMPLEMENTATION APPROACH 78

6.3 THE FORMAL SPECIFICATION NOTATION 79

6.3.1 DATA STRUCTURES 80

6.3.2 ACTIONS 81

6.4 THE FORMAL SPECIFICATION OF THE DATA DIVISION 83

6.5 IMPLEMENTATION OF A PARSER FOR THE DATA DIVISION 83

6.6 SYMBOL TABLE DATA STRUCTURES 84

6.6.1 RECORDS 86

6.6.2 ELEMENTARY RECORDS 87

6.6.3 UNIQUE VARIABLES 89

6.6.4 FILES 91

6.6.5 MNEMONICS 93

6.6.6 NON-UNIQUE VARIABLES 93

6.6.7 PARAGRAPHS 93

6.6.8 THE FILE DESCRIPTION INFORMATION LIST 94

6.7 OTHER DATA DIVISION DATA STRUCTURES 97

6.7.1 THE DATA NAMES LIST 97

6.7.2 THE FIELD LIST 98

6.7.3 THE GROUP ITEMS LIST 99

6.7.4 THE OCCURS CLAUSE LIST 99

7. THE COBOL COMPILER - PART II

THE PROCEDURE DIVISION AND CODE GENERATOR 101

7.1 INTRODUCTION 101

7.2 PARAGRAPHS AND SYSTEM LABELS 101

7.3 IMPERATIVE STATEMENT HANDLING 104

7.4 THE IF STATEMENT IMPLEMENTATION 105

7.5 THE PERFORM STATEMENT IMPLEMENTATION 106

7.6 OTHER IMPLEMENTATION FEATURES 113

7.7 THE IMPLEMENTED SYNTAX ANALYZER 115

7.8 THE SYNTAX ANALYZER - CODE GENERATOR INTERFACE 116

7.9 THE COBOL CODE GENERATOR 119

7.10 EXTENDING THE COBOL COMPILER 122

8. CONCLUSION 124

REFERENCES 127

APPENDIX 1 THE ADDRESS TABLE AND DATA AREA REQUIREMENTS OF THE CODE GENERATOR GENERATOR 129

APPENDIX 2 THE FORMAL DEFINITION OF THE ACTION ANALYZING ROUTINES 134

APPENDIX 3 THE CODE SKELETON ACTION IDENTIFIERS 147

APPENDIX 4 THE PSEUDO-CODE OR PSEUDO-CODE ACTION TABLE IDENTIFIERS 148

APPENDIX 5 THE SYNTAX OF THE COBOL SUBSET 152

APPENDIX 6 THE FORMAL SPECIFICATION OF THE DATA DIVISION 157

APPENDIX 7 THE FORMAL SPECIFICATION OF THE PROCEDURE DIVISION 163

APPENDIX 8 THE DECIMAL TO BINARY CONVERSION CODE SKELETONS 182

APPENDIX 9 THE ALGORITHM AND CODE SKELETONS FOR THE CONVERSION OF BINARY NUMBERS TO DECIMAL NUMBERS 183

APPENDIX 10 LISTINGS OF PROGRAMS, DATA AND DECOMPILED GENERATED SYSTEMS Appended

1. INTRODUCTION

The task of constructing a formal specification of a subset of COBOL and implementing part of a parser for this subset was attempted as a previous Honours project. This current project continues from the point at which the Honours project left off and has two prime objectives, viz.

1) To produce a code generator generator system which is capable of systematically producing code generators for any language on any machine given the appropriate set of parameters.
2) To complete the formal specification and partial implementation of a parser for the COBOL subset and to use the code generator generator to produce a code generator for this COBOL compiler.

A great deal is known about the design of lexical and syntax analysers for programming language compilers[1]. The machine independent techniques are reasonably well understood and lend themselves to mechanization. It is possible and usual to write parser generators in high level programming languages and some systems have been designed to be completely portable[16].

The automation of the process of producing a code generator generator has lagged behind that of parser generators. The chief reasons for this are that, besides depending on the language of which it is a part, a code generator generator

a) is much more dependent on the architecture of the target machine than a parser generator, and
b) also depends on the degree of optimization to be performed.

It is much more difficult to produce a system which is general enough to be applied to a wide range of architectures and which is easily transportable from one machine to another.

While the system has not attempted to solve the problem in the manner described by Fraser[12] in which he uses the machine description language developed by Bell and Newell[3], an attempt has been made to overcome some of the problems associated with code generation and to produce a system for code generation without optimization. This system is described in chapters 2 to 5.

The original intention in the case of the earlier Honours project was to produce a fast COBOL compiler for use by undergraduate students at Rhodes University. This was necessary because of the inefficiency of the ICL COBOL compiler and the large number of students compiling and testing COBOL programs causing the system to be severely congested for lengthy periods. Because of the size of the project and the time constraints involved not a great deal was accomplished.

Another intention of the earlier project was to investigate the effect of producing a formal specification of the syntax of the COBOL subset to be implemented. As some formal specification methods (eg. two-level grammars[15]) are of no use to the compiler writer a suitable method of formal specification had to be found.

The method chosen was a form of extended BNF[25] which reflects the methods used in the implementation of a compiler and thus serves to simplify the actual implementation effort as well as defining the problem areas in the compiler.

The COBOL compiler and the formal specification are described in chapters 6 and 7.

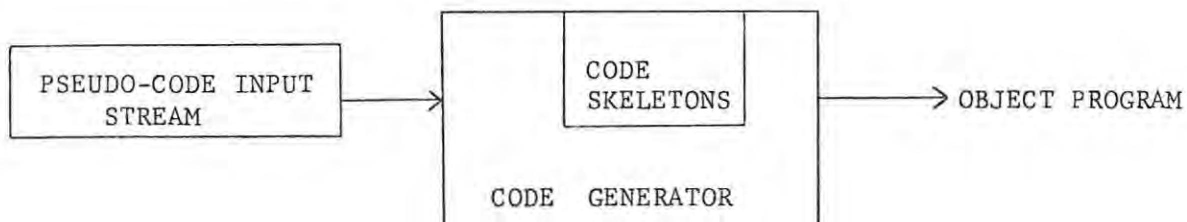
2. SYSTEM OVERVIEW

In constructing a code generator it is necessary to consider the type of interface which exists within a compiler between the parser and the code generator. Although there is no general agreement concerning this, one of two approaches is usually taken :

a) calls to the code generator routines are built into the parser so that the correct routines are called and code is generated at the appropriate points in the parsing process, or

b) the parser produces a sequence of pseudo-codes which are stored either in main memory or in a file on some backing storage device. On completion of the parsing process control is passed to the code generator which scans the pseudo-codes and generates the final object code [11,24].

A simplified view of a code generator of the latter type may be represented diagramatically as depicted below.



Basically the code generator accepts as input a stream of pseudo-codes and generates an object program which is assembled on the basis of the pseudo-codes from the set of built-in code skeletons.

In constructing a system which will generate code generators for different languages one of the first decisions which had to be taken was that concerning the structure of the code generator produced by such a system. Of the various possibilities it was decided to use a table driven approach in this system. A code generator produced by the system is made up of three major components, namely

a) The pseudo-code action table. This contains information on what action is to be performed on encountering each pseudo-code. It includes code

skeleton information and information on how to assemble the code skeletons. This information is stored in the table in a suitably encoded form.

b) The interpreter. This is a program which accepts as input the pseudo-codes produced by the parser and performs the corresponding actions either directly, by means of routines within the interpreter, or by reference to the pseudo-code action table.

c) Local routines. These are machine dependent and language dependent routines used by the interpreter.

The actions coded into the pseudo-code action table include fetching addresses, assembling instructions, storing code in buffers, etc. The local routines might include routines to read data from backing store, write buffers to backing store, allocate general purpose registers, etc.

In order to generate a code generator for a particular language the user must

a) create the pseudo-code action table relevant to the language and machine,

b) write whatever local routines may be necessary, and

c) create an interpreter in the machine code of the particular machine incorporating the pseudo-code action table and local routines.

In order to be able to produce a suitable pseudo-code action table it was first of all necessary to investigate the requirements of a code generator. To do this the facilities which would be required by a particular language code generator for a particular machine were considered. The particular code generator considered was for a COBOL compiler on an ICL 1902T machine running GEORGE 2/MK9F and MAXIMOP. This investigation of code generator requirements is the subject of chapter 3.

Chapter 4 describes the high level language program which generates a pseudo-code action table from a specification of the opcode to internal machine code

mapping and the code skeleton definitions. The code skeleton definitions may include any of the facilities described in chapter 3.

A high level formal definition of the code generator requirement actions was written and then expressed in terms of a set of general system pseudo-codes. Thefunctions of these general system pseudo-codes are described in detail in chapter 5. At this stage a pseudo-code definition of the interpreter existed but no interpreter capable of processing these pseudo-codes had been produced.

In order to produce an interpreter a high level bootstrap program allowing only a subset of the actions was written. Because of the limited set of facilities allowed by the bootstrap program it was necessary to produce a pseudo-code action table where each code generator action was defined by means of a single locally optimized code skeleton. This procedure is described in chapter 5.

The local routines, of which in general there will be very few, will need to be written afresh for each new machine, and possibly for different languages on the same machine. There are three local routines required by every code generator produced by the system. Descriptions of the functions of these routines can be found in chapter 5.

3. THE INVESTIGATION OF CODE GENERATOR REQUIREMENTS

3.1 GENERAL CODE GENERATOR FUNCTIONS

Before considering the general code generator generator system it is first necessary to consider the requirements of a code generator. Typical operations which a code generator performs include:

(a) Manipulation of code skeletons

The code generator will contain the code skeletons for the language which must be assembled in the correct order to produce the final object program.

(b) Computation of address fields

The addresses may be passed to the code generator in the stream of pseudo-codes produced by the parser or they may be looked up in tables. It may be necessary to perform simple arithmetic operations on the addresses before they are inserted into the appropriate fields within the instructions in the final object program. Besides simple addresses both forward and backward branches must be catered for.

(c) Assignment of registers

The code generator may be responsible for the assignment of general purpose registers, index registers, etc. to the machine code instructions in the code skeletons.

(d) Allocation of data areas and constants

Besides handling the instructions, the code generator will also be responsible for allocating areas of storage as data areas and for creating preset constants in some of these data areas.

(e) Relocation information

If the code generator produces relocatable binary then it will be responsible for setting up the relocation information for the loader program.

(f) Buffer management

As pseudo-codes are fetched from input buffers and generated object program

is placed in output buffers, the code generator will control the buffers and, on encountering the end of a buffer, be required to read information into or write information from the appropriate buffer.

Although there will be variations on these basic functions for different machines and possibly, though to a lesser extent, for different languages on the same machine these are the main functions one would expect of a code generator.

3.2 ICL 1900 SYSTEM COBOL CODE GENERATOR ACTIONS

In order to get a more detailed idea of the operations to be performed by a code generator the features required by a COBOL code generator for the ICL 1900 system were considered. A COBOL code generator was considered for two main reasons, viz.

(a) A formal specification of a COBOL syntax analyser, as far as the end of the DATA DIVISION, had been produced in the earlier project and an attempt had been made to implement this formal specification. Thus the COBOL code generator requirements were, to some degree, recognized and understood.

(b) A character based language such as COBOL should require most code generator features which would be required by other language code generators.

In addition to the COBOL code generator features provided by the system a number of additional facilities not specifically required by COBOL but which may be useful in code generators for other languages have been included.

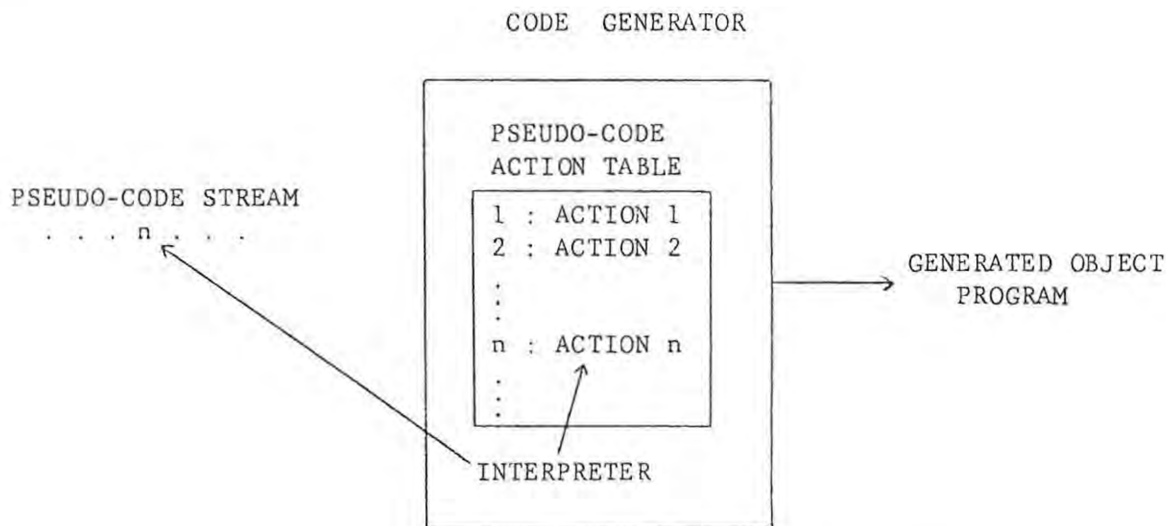
The major feature not required by a COBOL code generator but required by code generators for languages such as ALGOL or PASCAL is a means for handling recursive calls to procedures. An attempt has been made to cater for the data area requirements of a compiler allowing recursion.

The features or code skeleton actions described below are divided into four sections, viz. those which may only appear in the code skeletons as

specified by the user, those which provide definitions of data areas, constants, addresses, etc., an initialization action and actions transparent to the user but inserted by the code skeleton processing program into the pseudo-code action table.

A code generator produced by the system operates as a table driven interpreter. This means that it will fetch a pseudo-code from the pseudo-code stream and interpret the appropriate section of the table to provide the action for that particular pseudo-code.

Initially each pseudo-code together with its corresponding set of instructions which must be performed on encountering the pseudo-code is set up in the pseudo-code action table. When the interpreter reads a pseudo-code from the pseudo-code stream it will look up the corresponding entry in the pseudo-code action table and interpret it.



Every entry in the pseudo-code action table definition will consist of the following.

a) A heading of the form

SK pseudo-code number, number of parameters.

b) The pseudo-code action or code skeleton body which may incorporate the features described later in this chapter.

3.3 CODE SKELETON ACTIONS

The actions in this category may only appear within the code skeletons which are used for the definition of the pseudo-code action table. In addition to the code skeleton actions described in this section two further actions in this category, viz. unstack address and branch forward, are described in sections 3.4.9 and 3.4.10 respectively.

3.3.1 PSEUDO-CODE ACTION PARAMETERS

In the definition of each pseudo-code action the number of parameters is specified. When a pseudo-code action requiring n parameters is encountered in the input stream the next n items in the input stream are taken as the actual parameters. Within a pseudo-code action definition %i is used to indicate an occurrence of the ith parameter.

Given the following pseudo-code input sequence and pseudo-code action definition the indicated code will be generated in binary form.

```
Pseudo-code input sequence :      20,3,100
Pseudo-code action definition :  #SK   20,2
                                  LDN   1  %1
                                  STO   1  %2
Generated code :                  LDN   1  3
                                  STO   1  100
```

Pseudo-code action parameters may be used in conjunction with some of the other actions to be described presently and will be mentioned at the appropriate points in the action descriptions.

3.3.2 ADDRESS TABLE ACCESSES

Within the code generator a table of addresses is used for passing addresses from one routine to another. In particular it is used for passing addresses from one pseudo-code action to another, for linking to local routines, etc. Whenever the address table is accessed the element number, or index, of the address table element must be given.

Thus the address table constitutes the fundamental means of reference or access to storage defined within the data areas or the instruction area. See Section 3.4.

The address table element number used in the access may be either a literal or a pseudo-code action parameter, in either case the value (the literal value or the value of the parameter) is taken as the table index. An address table access returns the value in the referenced address table element to a standard location. Within a pseudo-code action the symbol ! is used to indicate an address table access.

As an example of the use of the address table access facility consider the following.

Pseudo-code input sequence   20,2,1

Pseudo-code action definition : #SK 20,2

```
LDN 1  %1
LDX 2  !%2(1)
STO 2  !2
```

Address table :

3	
2	200
1	100

Generated code :

```
LDN  1  2
LDX  2  100(1)
STO  2  200
```

3.3.3 EXPRESSIONS

Operand fields within instructions in the code skeletons used in the definition of the pseudo-code actions may contain simple expressions which may include addition, subtraction, multiplication, a relative branch or a call to a user function. The following BNF definition describes the form of an expression.

```
<expression>::=<expression1>|<expression2>

<expression1>::=<term>|<term>+<integer>|<term>-<integer>|
                <term>*<integer>|<integer>

<expression2>::=*+<integer>|*-<integer>|<user function call>

     <term>::=<parameter>|<address table element>

<address table element>::=! <integer>|!<parameter>

     <parameter>::=%<integer>

<user function call>::=↑<integer>{(<parameter list>)}^1_0

<parameter list>::=<param>{,<param>}^∞_0

     <param>::=<expression1>
```

In addition to user function calls it is also possible to have user procedure calls which do not return a value to be inserted into an instruction operand field. They have the same form as a user function call except that they appear on their own instead of as an instruction operand. A procedure will generally be called in the situation where one wishes to retain some information about the current state of the system for some future reference. An example of the use of procedures can be seen in section 7.2.

For the implementation of user functions and procedure calls in the code generator it is necessary to have an area of storage set aside for the passing of parameters to those routines which require them.

An example of the use of the described features is given below.

```
Pseudo-code input stream      :  20,2,2,4
Pseudo-code action definition :  #SK   20,3
                                 LDN 1 %1+2
                                 LDX 2 !%2+4(1)
                                 STO 2 !4+5
                                 BNZ 2 *+2
                                 BRN   ↑4(%3+1,!%2)
```

Address table :

4	280
3	200
2	100
1	50

Code generator output : (Assume the current location count is initially 400)

LOCATION	CODE
400	LDN 1 4
401	LDX 2 104(1)
402	STO 2 285
403	BNZ 2 405
404	BRN x
405	. . .

Where x is the value returned by user function 4 when passed parameters with the values 5 and 100.

3.3.4 RELOCATABLE INSTRUCTIONS

If an address field of an instruction is to contain a value which is relocatable with respect to the start of any data area or the start of the instruction area then the instruction is preceded by

Ø <integer>

where the integer denotes the data area number. The instruction area is specified as area 0.

If one is generating relocatable code it will be necessary to mark all instructions containing relative branches as being relocatable with respect to

the start of the instruction area.

If this facility is used then the code generator must contain a relocation buffer and an associated data block. See section 3.4.4. The local routine to write a buffer to a disc file will be required to mark the buckets or sectors of relocation information in an appropriate manner for identification by the loader program. Each piece of information inserted into the relocation buffer will contain two components, viz. the address of the instruction whose operand is to be relocated and the data area number with respect to which it must be relocated.

3.3.5 CONDITIONAL INSERTION OF CODE

Within a pseudo-code action it is possible to specify that alternative actions are to be taken depending on the truth or falsity of a condition. To achieve this one uses a statement of the form :

#IF <condition> #THEN<statement>#FI , or

#IF <condition> #THEN<statement>#ELSE<statement>#FI

where <condition>::=<exp><relational operator><exp>

<relational operator>::= =|≠|>|>=|<|<=

<exp>::=<expression1> <user function call>

<expression1> and <user function call> are defined in section 3.3.3 and <statement>is any sequence of actions and instruction specifications which are to be carried out or inserted depending upon the evaluation of the condition. <statement> may not contain another conditional code insertion sequence.

The implementation of this action requires the definition of four locations within the code generator to hold the operands of the condition, the type of the relational operator and an operand flag.

An example of the usage of this facility is as follows.

Pseudo-code input stream : 20,5,4,1,2

Pseudo-code action definition : #SK 20,4

LDN 1 %1

#IF !%2 > 4095

#THEN SMO !%3

```
            LDX   3    !3+2
    #ELSE   LDX   3    !3+3
    #FI
      STO   3    !%4,1
```

Address table:

4	5000
3	500
2	400
1	100

Generated code :

```
LDN   1   5
SMO       100
LDX   3   502
STO   3   400(1)
```

3.3.6 FETCH, FREE AND USE REGISTERS

These three actions are used to assign and access registers within a pseudo-code action. In any system one may have different types of registers to be allocated, e.g. ordinary general-purpose registers, index registers, pairs of general-purpose registers (required for example in multiply and divide operations [19,20]), floating point registers, double precision floating point registers, etc. At present the system caters for three types of register allocation although this could easily be extended. For each class of registers one has two stacks :

(a) the stack of registers currently in use (USE STACK), and

(b) the stack of free registers (FREE STACK).

The fetch register command has the form

#FETCH(i)

When such a command is given, a register of type i is taken from the appropriate FREE STACK and placed on the corresponding USE STACK. If the FREE STACK is empty when this action occurs, it will be necessary to select one of the registers in the corresponding USE STACK and to store its contents in some temporary location.

The free register command has the form

FREE (i)

and causes the register at the top of USE STACK i to be transferred to FREE STACK i.

Access to the nth register from the top of the USE STACK i may be represented as follows.

```
        # REG(i)-n            (n ≠ 0)
or      # REG(i)              (n = 0)
```

As an example of the use of these facilities consider the following.

Assume that type 1 registers are index registers, type 2 registers are general purpose registers and type 3 represents a pair of general purpose registers. As types 2 and 3 are both concerned with general purpose registers they will employ the same FREE and USE STACKS.

```
Pseudo-code input stream        : 14,10,15,11,16,17,2,13
Pseudo-code action definitions  : # SK    14,1
                                  # FETCH(3)
                                   LDX # REG(2)-1  !%1
                                  # SK   15,1
                                  # FETCH(2)
                                   LDX # REG(2)    !%1
                                  # SK   16,0
                                   MPY # REG(2)-2 # REG(2)
                                  # FREE(2) # FREE(2)
                                  # SK   17,2
                                  # FETCH(1)
                                   LDN  #REG(1)    !%1
```

```
STO  #REG(2)  !%2  (#REG(1))
#FREE(1)
#FREE(2)
```

Address table:

13	105
12	102
11	101
10	100

Free stack (types 2 and 3)

0 ← TOP OF STACK
7
6
5
4

Free stack (type 1)

3 ← TOP OF STACK
2
1

Generated code :

```
LDX   0   100
LDX   6   101
MPY   0     6
LDN   3     2
STO   0   105(3)
```

On completion of the generation of the above code the free stacks will again be in the states depicted above.

3.4 DEFINITION ACTIONS

The definition actions together with their required parameters may appear in the pseudo-code stream to be input to the code generator or they may appear within a pseudo-code action sequence. The action taken by the code generator is the same in both cases. In the first case the parameters for the action are to be found in the pseudo-code input stream and in the second case in the pseudo-code action table.

Certain of the definition actions cannot be logically included within a code skeleton and are trapped as illegal in the program which produces the pseudo-code action table from the code skeleton definitions.

3.4.1 DEFINE STORE

This action permits the definition of an area of storage of a specified size in a specified data area. The size of the area is specified as a certain number of basic units, where a basic unit is the unit of storage for the machine concerned, e.g. a word, a byte, a half-word, etc.

A define store operation has the form shown below.

#DS data area number, address table element number, size

For example on the 1900 system

#DS 3,10,1000

defines a 1000 word area of storage in data area 3 whose starting address will be inserted into address table element 10.

It is possible to specify that a code generator is to have more than one data area in which storage may be defined. For this reason it is necessary to specify in which data area the storage is to be defined. In the simplest case a user will probably only employ a single data area and an instruction area and have the data area entirely specified before the program instructions, or alternatively a single area with sections of data and program instructions intermingled.

However, in general, one may have to deal with more than one data area and accordingly the system allows for a variable number of buffers. Thus, for example, on an ICL 1900 where one has an instruction area and two data areas (#LOWER and #UPPER) a code generator may require three separate buffers.

Associated with each area of storage defined by each individual storage definition is an address table element which gives the location of the first word of the defined storage area.

If data is defined simultaneously in two data areas then the address table will not contain the absolute addresses of the storage areas as they will appear in the final object program. This problem is solved by means of the relocatable instruction facility (3.3.4).

Within a compiler for a language which allows recursion it has been assumed that the store management will be handled in a manner similar to that described by Wirth[27] and Ammann[2]. By allocating a sufficiently large area of working storage this method of recursive procedure calling should be adequately handled.

3.4.2 DEFINE CONSTANT

The purpose of this action is to permit the user to define constants within a generated object program. As in the define store operation the action identifier is followed by the data area number in which the constants are to be defined and an address table element number which will contain the address of the first of the defined constants in the constant list. In addition, the number of constants to be defined and the constants themselves are specified.

The current implementation of this action assumes that the constants are each to occupy one basic unit (i.e. one word) in the final object program. For machines with a byte type architecture it may be desirable to allow the user to specify, within the action, the number of basic storage units to be occupied by each constant within a generated object program.

The define constant action has the form

#DC data area number, address table element number, $n, c_1, c_2, \ldots, c_n$

where c_1, c_2, ..., c_n are the n constants.

3.4.3 ALIGN

This action was designed primarily for a byte oriented system in which constants and/or instructions are required to start on particular basic storage unit boundaries, e.g. half word or full word boundaries. The action requires two parameters specifying the data area number in which the alignment is to take place (for this action the instruction buffer is considered to be data area 0) and an integer value. The pointer to the current position within the specified data area is set to the next multiple of the specified integer value.

This action has not been found to be necessary in any of the ICL applications considered, however it will certainly be necessary in a byte oriented system.

It has the form :

ALIGN data area number, integer

3.4.4 TRANSFER ADDRESS

Each buffer within the system, i.e. data area buffers, the instruction buffer and the relocation information buffer, has an associated data block containing the following information.

(i) A pointer to the current position within the buffer relative to the start of the buffer. In the case of the data area buffers and the relocation buffer this will be a word or byte pointer while in the case of the instruction buffer a bit pointer is required since the code generator will be assembling instructions from the information within the pseudo-code action table and the parameters associated with the relevant pseudo-codes.

(ii) The buffer size in words or bytes. This is required to reset component iii of the data block after the contents of the buffer have been written to an output disc file.

(iii) The amount of free space remaining within the buffer. Once again in the case of the data buffers and the relocation buffer this is a word or byte count while in the case of the instruction buffer it is the number of unused bits remaining within the buffer.

(iv) An absolute buffer counter which contains the total number of words or bytes which have already been written to the output file for the buffer concerned.

This action has been designed for a code generator which will generate absolute code. The action makes it possible to transfer the absolute current location count from one buffer (i.e. the value contained in the sum of components 1 and 4 of the corresponding data block) to another. This would be required in the situation where a user wished to generate absolute instructions after the definition of his data areas.

The two parameters required by the action are the data area number from which the address is to be transferred and the data area number to which it is to be transferred.

An example of the use of the transfer address function can be seen in the data used for generating the COBOL code generator. All the storage areas and constants are defined initially. The current location count is then transferred from the data area buffer to the instruction buffer enabling absolute code for the local routines to be generated. After the definition of the local routines the current location count is transferred back to the data area for the definition of the start addresses of the local routines. The absolute location count is then transferred back to the instruction buffer for the definition of the rest of the code generator. See appendix 10.1.

Obviously it is not necessary to produce a code generator or any other object program in the way described above as the instructions having relative addresses as operands could be marked as being relocatable with respect to the start of the appropriate area and have their addresses resolved by the loader program.

The form of the transfer address action is :

#TRANSFER sending data area number, receiving data area number

3.4.5 DEFINE ADDRESS CONSTANT

The purpose of this action is to enable the user to initialize a location within the final object program with the contents of an address table element. As the address table in the code generator (or code generator generator) contains the absolute addresses of the first words of defined storage areas or defined constants this action provides a means for defining these addresses within the generated object program.

The action takes the following four parameters.

1. The number of the data area in which the address constants are to be defined.
2. An address table element number which will be set to the address of the first defined address constant in the data list.
3. The number of elements in the data list.
4. The data list consisting of the address table element numbers whose values are to be defined.

The action has the form :

DAC data area number, address table element number, $n, c_1, c_2, \ldots, c_n$

where $c_1, c_2, \ldots, c_n$ are address table element numbers.

As an illustration of the difference between #DC, #DS and # DAC consider the following example.

Assume that the current location counter has the value 100 before the following 3 definitions are processed.

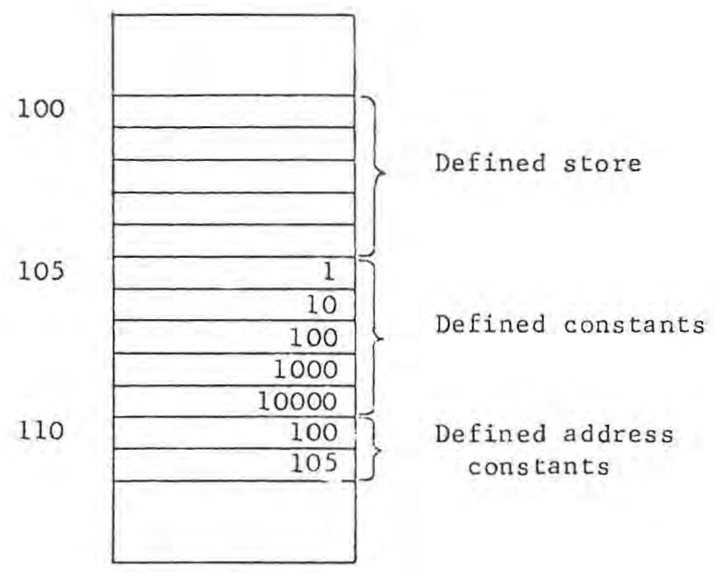
```
# DS     1,10,5
# DC     1,11,5,1,10,100,1000,10000
# DAC    1,12,2,10,11
```

After these definitions have been processed the address table within the code generator and the generated object program will be as shown below.

ADDRESS TABLE IN CODE GENERATOR

10	100
11	105
12	110

GENERATED OBJECT PROGRAM



As an example of the use of this facility consider the routine within a code generator whose function is to write buffers of data to a disc file. Since several different buffers exist it would be a waste of storage space to have a separate instruction sequence and control area for the writing of each buffer.

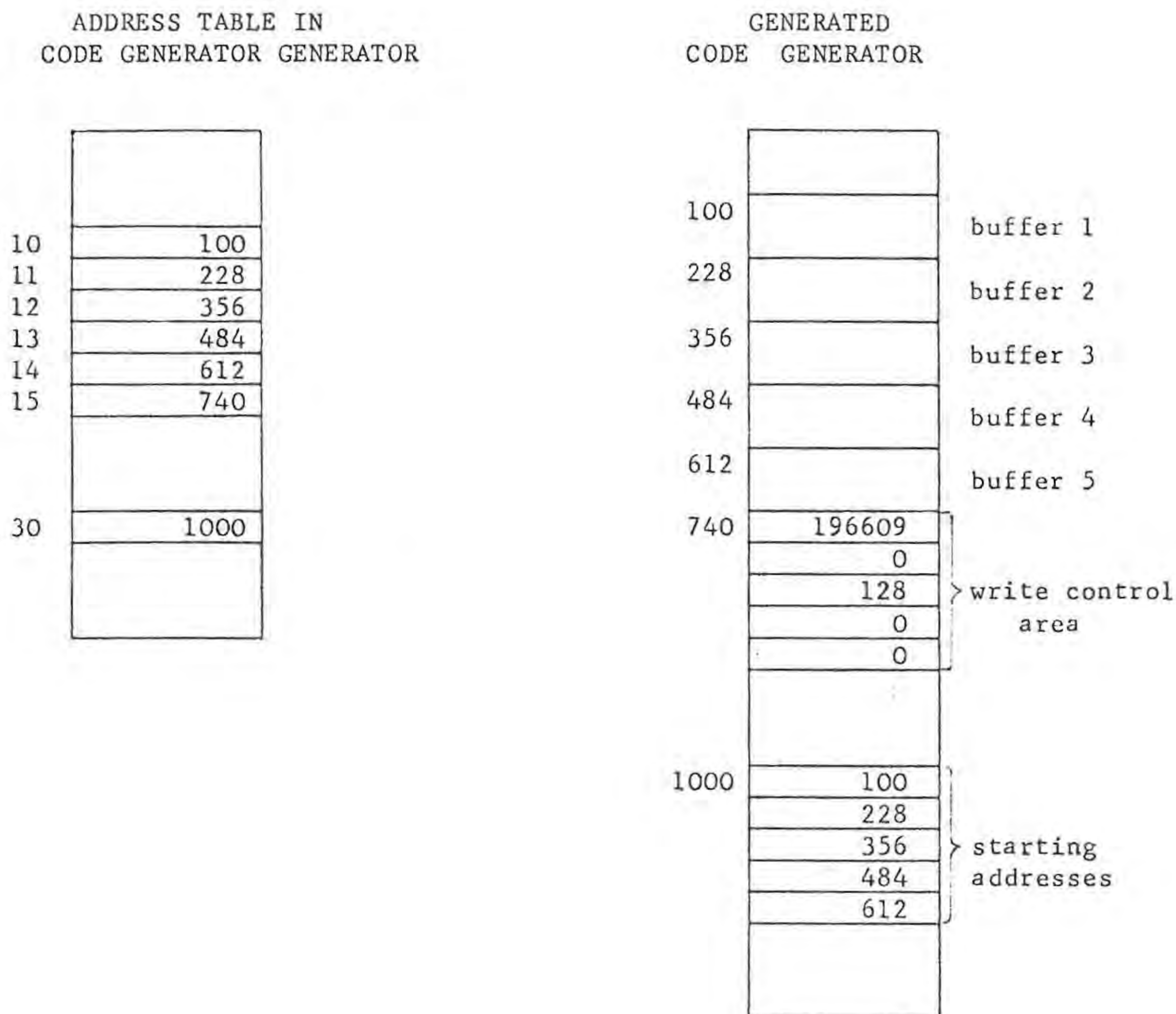
The routine will have one control area and is passed a parameter indicating which buffer is to be written to the disc file. Assume that address table element 30 was defined in the code generator definition data to be the starting address of an area containing the starting addresses of the various buffers. The parameter passed to the routine will be the relative position in this list of the required start address.

The control area for a disc file write operation occupies 5 words with the fourth word containing the starting address of the area whose contents are to be written. The third word contains the number of words to be written.

Assume that the code generator definition data contained the following definitions.

```
# DS     1,10,128                      }
# DS     1,11,128                      }
# DS     1,12,128                      } buffers
# DS     1,13,128                      }
# DS     1,14,128                      }
# DC     1,15,5,196609,0,128,0,0       control area
# DAC    1,30,5,10,11,12,13,14         address definitions
```

If the current location counter has the value 100 before the first define store action is processed and 1000 before the define address constant action is processed then the address table and the object program will be as shown below.



Assume that 1. the parameter is passed to the routine in accumulator 1, and

2. channel 2 is associated with the output file.

The routine might be defined as follows.

```
LDX    2   !30(1)     load buffer address
STO    2   !15+3      store address in control area
LDN    3   1          increase bucket number
ADS    3   !15+4          by one
PERI   2   !15        write buffer
check reply words, etc.
```

The routine would appear in the generated program as shown below.

```
LDX   2   1000(1)
STO   2   743
LDN   3   1
ADS   3   744
PERI  2   740
etc.
```

3.4.6 DEFINE LOCAL ROUTINE

This action makes possible the insertion of any local routines into the final object program. Local routines may be required to get around system dependent features as well as to avoid the unnecessary duplication of code within an object program. The action has the form:

#DLR address table element number, routine length, routine code

The address table element number will be set to the starting address of the routine in the final object program and may be used in instructions which call the routine.

The routine length and the routine code are obtained from the pseudo-code action table as produced by the pseudo-code action definition processing program when it is given the routine in pseudo assembler instructions as input. (See chapter 4).

As local routines are generally quite specifically written for only one system it was decided to limit the permissible range of actions within the code skeletons constituting the local routines. The allowable code skeleton actions are address table accesses (3.3.2), arithmetic expressions and relative branch expressions (3.3.3).

This action may not appear within a code skeleton as it makes use of the pseudo-code action buffer for its interpretation. As the binary code for a local routine is created in the instruction buffer it is necessary, if one is generating absolute code, to precede the definition of a local routine by a

transfer address function from the current data area buffer to the instruction buffer if the system is not already using the instruction buffer for output.

There are three local routines which require certain fixed address table element numbers to contain their start addresses. These routines are required to be included in any code generator generated by the system. The functions of these routines are described in detail in section 5.1.3 and are briefly as follows.

ROUTINE FUNCTION	ADDRESS TABLE ELEMENT NUMBER
fetch the next source symbol	31
fetch a pseudo-code action	32
write a buffer to a disc file	33

Any other local routines may be included in the final object program with the proviso that if they are to be called by means of the call user routine action then their start addresses must be defined in the correct order in a define address constant action having address table element 30 associated with the list of defined address table elements.

3.4.7 DEFINE ADDRESS TABLE ELEMENT

The function of this action is to assign a specific value to an address table element. The value to be assigned to an address table element may be an integer value or it may be the current location counter.

The primary use of this function will be to assign the current location counter to an address table element. In this way address table elements may be associated with the start addresses of the various local routines within a generated object program. This enables these routines to be called from the control segment or any segment appearing later in the object program.

This action is used chiefly when using the system to generate a code generator and will seldom be used by a particular code generator. A use of this action, as employed by the COBOL code generator, is the definition of the start

address of the instructions after the data area definitions, i.e. defining an entry point to the program.

The implementation of this action for the definition of the current address assumes that the buffer currently being used is the instruction buffer and thus the address is calculated from the absolute buffer pointer and the current position within buffer pointer components of the instruction buffer data block.

The action has the form :

TABULATE address table element number, value

3.4.8 FINAL ACTION

This action is invoked by means of a pseudo-code which should be the last one in the pseudo-code input stream for the code generator. This system dependent routine is required to empty any system buffers which still contain parts of the final object program and to close any disc files which are required to be closed.

In addition to the above functions the routine may be required to write loader information contained within tables in the code generator to a disc file. Some examples of typical loader information are as follows.

A map giving the disc file addresses of data buckets or sectors and the addresses of buckets or relocation information. If relocatable code was generated then the absolute start addresses of the various data areas and the instruction area will be required.

The contents of tables containing the terminal nodes of branch ahead chains and the addresses to be inserted into the operand fields for all instructions in the chains. A simple example of this can be seen in the COBOL code generator.

The entry point to the generated object program.

The routine may terminate either with an appropriate message indicating that the code generation has been successfully completed or it may call a loader program to load the generated binary program if this is desired.

3.4.9 STACK AND UNSTACK THE CURRENT ADDRESS

These two actions are described together as they constitute a means for handling backward branches. For this purpose the system employs a backward branch address stack.

Stack the current address is a definition action which when encountered by the code generator causes the current location count, which is obtained from the instruction buffer absolute address and the current position within the buffer, to be pushed down onto the backward branch address stack.

Unstack the current address is a code skeleton action and thus occurs only within a code skeleton. The action will appear as part of an instruction and causes the entry at the top of the branch backward address stack to be taken off the stack and inserted into a standard location whence it may be picked up and inserted into the appropriate operand field of the branch instruction concerned.

Stack the current address and unstack the current address are represented as #STACK and #UNSTACK respectively.

An example of the use of the backward branching system is as follows.

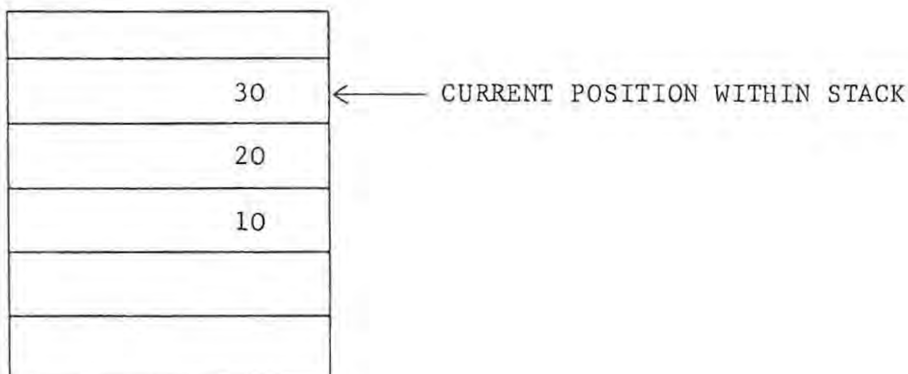
	PSEUDO-CODE SEQUENCE
a	#STACK
	pseudo-codes not containing stack or unstack operations.
b	#STACK
	pseudo-codes not containing stack or unstack operations.
c	#STACK
	pseudo-codes not containing stack or unstack operations.
d	20,1
	20,2
	20,3
e	

Pseudo-code action definition :

```
# SK     20,1
LDN      1   %1
TXU      2   1
BCC          # UNSTACK
```

Assuming that the current addresses at points a, b and c are 10, 20 and 30 respectively then at point d the branch backward address stack will be in the form depicted below.

BRANCH BACKWARD ADDRESS STACK



On reaching point e in the above pseudo-code sequence the generated code will be

```
10   .
     .
20   .
     .
30   .
     .
         LDN    1    1
         TXU    2    1
         BCC         30
         LDN    1    2
         TXU    2    1
         BCC         20
         LDN    1    3
         TXU    2    1
         BCC         10
```

The pointer to the current position within the branch backward address stack will now point to the node below the one containing the value 10.

3.4.10 BRANCH FORWARD AND RESOLVE FORWARD BRANCH

Once again these two actions will be considered jointly as they constitute a means for handling forward branches.

Resolve forward branch is a definition action which has the format

#RESOLVE

and may appear on its own in the pseudo-code input stream or within a pseudo-code action. Branch forward is a code skeleton action which may only appear within a pseudo-code action as an instruction operand and has the format

#FORWARD.

It was recognized that forward branches would generally be required in IF-THEN-FI or IF-THEN-ELSE-FI type constructs. Bearing this in mind the system has been designed in the following way.

If this facility is to be used the instruction buffer should be specified as being two or more times the size that is written away in any one call to the "write buffer to disc" local routine. In the code generator generator the required size was found to be four times that written away in one write operation.

The second component of the data block for the instruction buffer should still indicate that the buffer size is 1 block or sector. The additional buffer space is required as there may be unresolved forward branches within the instruction buffer at the time when the buffer becomes full and requires emptying. Obviously the buffer cannot be emptied at this point and thus code is inserted into the adjacent buffer space until all the forward branches are resolved and the buffer can then be emptied.

The amount of additional buffer space required can be determined by estimating the maximum level of IF-THEN-ELSE-FI construct nesting and the sizes of the blocks of code to be inserted in each case.

It should be noted that two forward branch actions without an intervening resolve forward branch constitute an IF-THEN-ELSE-FI and that a construct of the type IF-THEN IF-THEN-FI FI is not possible.

The system has been implemented by making use of a forward branch stack (FBS). The algorithm for handling forward branches is as follows.

```
IF    FBS.INDEX = 0        (i.e. FBS empty)
THEN
      INDEX = 1;
      insert the current program location count into
      FBS [1,INDEX] ; FBS [2,INDEX] = 'THEN';
      increase the amount of free space left in the instruction
      buffer by the additional allocated buffer size
ELSE
IF    FBS [2,INDEX] = 'THEN'
THEN
      insert the current program location count+1 into the
      operand field of the instruction pointed to by FBS [1,INDEX] ;
      insert the current program location count into FBS [1,INDEX] ;
      FBS [2,INDEX] = 'ELSE'
ELSE
      INDEX = INDEX+1;
      insert the current program location count into FBS [1,INDEX] ;
      FBS [2,INDEX] = 'THEN'
```

The algorithm for handling a resolve forward branch action is as follows.

```
insert the current location count into the operand field of
the instruction pointed to by FBS [1,INDEX] ;
INDEX = INDEX-1;
IF INDEX = 0 then reduce the instruction buffer
```

Reduce the instruction buffer causes buckets or sectors of object program to be written to the output disc file until the amount of used space left within the buffer is less than one bucket. The amount of usable space left within the buffer is also reset.

Some examples of the use of the forward branch system are as follows.

Pseudo-code input stream : 20, 50, 60, 21, 70, #RESOLVE

Pseudo-code actions 20 and 21 are defined as:

```
#SK       20,2                  #SK     21,1
LDX    1     %1                 STO   1    %1
TXU    1     %2
BCS          #FORWARD
```

Assuming that the current location count has the value 30 when pseudo-code 20 is encountered by the code generator, the following code will be generated.

LOCATION COUNT	CODE		
30	LDX	1	50
31	TXU	1	60
32	BCS		34
33	STO	1	70
34	...		

Consider a slightly more complicated example using the above two pseudo-code actions as well as pseudo-code action 22 defined as follows.

```
#SK    22,0
BRN    # FORWARD
```

PSEUDO-CODE INPUT STREAM	CONSTRUCT
20, 50, 60	IF THEN
21, 90.	
22	ELSE
20, 60, 70	IF THEN
21, 90	
22	ELSE
20, 70, 80	IF THEN
21, 90	
#RESOLVE	FI
# RESOLVE	FI
#RESOLVE	FI

The code generated would be the following.

LOCATION COUNT	CODE		
30	LDX	1	50
31	TXU	1	60
32	BCS		35
33	STO	1	90
34	BRN		44
35	LDX	1	60
36	TXU	1	70
37	BCS		40
38	STO	1	90
39	BRN		44
40	LDX	1	70
41	TXU	1	80
42	BCS		44
43	STO	1	90
44	...		

When the statement at location 43 is generated the forward branch stack will be in the state shown below.

FORWARD BRANCH STACK

42	THEN	← INDEX
39	ELSE	
34	ELSE	

3.5 THE INITIALIZATION ACTION

The initialization action routine is defined in the same way as the rest of the action analyzing routines and not as a local routine even though it is a completely system dependent routine.

The routine is not called in the same way as a local routine or on encountering a pseudo-code in the input stream but is automatically called by the controlling segment of the code generator as the first operation to be performed.

The functions of this action are to open all the disc files required by the code generator in the correct modes (e.g. input, output, etc.) and to read the pseudo-code action table index into the appropriate area of storage whence it may be accessed by the "fetch pseudo-code action" routine.

As this routine is completely system dependent no further description of it will be given. The initialization action routine for the ICL system can be found in appendix 10.2 as pseudo-code action 19.

3.6 USER TRANSPARENT ACTIONS

The actions in this category are inserted into the pseudo-code action table by the program which processes the pseudo-code action definitions. The actions are inserted to make the implementation of the code generator easier and to provide a means for inserting the values returned by some of the other actions into the instruction buffer at the appropriate points.

The four actions in this category are the following.

(a) Immediate bits

The specified number of bits are to be copied from the next character position into the instruction buffer.

(b) Immediate characters

The specified number of characters are to be copied from the pseudo-code action to the instruction buffer.

(c) Extract bits

Actions which produce some sort of result will place the result in a

standard location whence it may be extracted by this action and added to the instruction buffer. The actions which return results to the standard location are : a pseudo-code action parameter - %i, expressions, # UNSTACK, # FORWARD, address table accesses - !n, # REG and any user function calls.

(d) Literals

Literals appearing as address table element numbers, pseudo-code action parameter numbers, operands within condition, etc. are preceded by this action within the pseudo-code action table.

3.7 FORMALIZATION OF THE CODE GENERATOR REQUIREMENTS

Once the actions required by a code generator had been identified they were formalized in an ALGOL type notation which incorporated the additional features described below. In addition the data areas and constants required by the routines for the execution of the actions were specified.

The three local routines required by any code generator produced by the system and mentioned in section 3.4.6 have been called FETCHNEXT, FETCHACTION and WRITEBUFFER and will be referred to by these names within the action definitions. Calls to these routines in the definitions will appear as ordinary ALGOL procedure calls with an associated parameter which may subsequently be regarded as containing the value returned by the procedure.

In the implementation of these routines the parameters will be passed by means of standard locations used by the called and calling segments.

For example :

FETCHNEXT (DATAAREA)

is a call to the procedure to fetch the next symbol from the input stream and places the symbol in a variable called DATAAREA. The value returned may be an address table element number or some integer value.

In the case of an address table element number being returned by a procedure the use of the address table element will be denoted by preceding the appropriate variable name by an exclamation mark (!). Square brackets ([and]) have been used to denote 'the contents of'.

If ADDRESSVAR and DATAAREA contain the values 2 and 13 respectively and the address table is in the state depicted below

14	200
13	100

then the statement

!14 + ADDRESSVAR = [!DATAAREA + 3] + [!DATAAREA]

would be interpreted as :

location 202 = contents of location 103 + contents of location 100

A complete list of the data areas, their associated address table element numbers and functions is given in appendix 1. Appendix 2 contains the action definitions.

4. PSEUDO-CODE ACTION DEFINITION PROCESSING

4.1 INTRODUCTION

The primary function of the program which processes the code skeletons or pseudo-code action definitions is to produce the pseudo-code action table and an index to this table in a form suitable for input to the code generator. The program checks the syntax of the code skeletons while it is busy doing the processing and reports any errors which are encountered.

To achieve the above function the program requires a definition of the user's assembly language opcodes and the corresponding internal machine code versions with the parameter fields and constant fields appropriately specified.

4.2 INPUT

Input to the program consists of the following:

The byte or character size of the machine for which the pseudo-code action table is to be produced.

The number of code skeletons to be processed. This is specified merely to keep array sizes to a minimum.

The opcode to machine code mapping.

The pseudo-code action definitions.

An end of data marker.

4.2.1 THE OPCODE TO INTERNAL CODE MAPPING

The keyword OPCODES is used to identify the start of this section of the data and is followed by a number giving the length of the longest opcode. This has been included to enable the opcodes to be packed into words and not stored as individual characters. See section 4.5.

This is followed by the number of different instruction lengths which will appear in the following definitions and the lengths in bits of these instructions. In the case of the ICL 1900 system there is only 1 permitted instruction length, however in the case of the INTERDATA 8/32 system there are three possible

different instruction lengths. The instruction lengths are specified in order to enable the system to check that the machine code form for any instruction is of a valid length and to use the least amount of storage space for the storage of the machine code representations.

This specification is followed by the definition of each opcode and its corresponding machine code. The constant parts of the instructions are specified as zeroes and ones while the variable parts appear as letters - parameter 1 being denoted by the letter A, parameter 2 by B, etc. For example:

STO AAA0001000CCBBBBBBBBBBBB

specifies that the STO instruction has three parameters, the first occupying bits 0 to 2, the second bits 12 to 23, the third bits 10 to 11, while bits 3 to 9 are fixed.

No provision has been made for the insertion of default values into any of the fields and all the parameters must be specified when a particular opcode appears within a code skeleton.

Some examples of this opcode to machine code mapping are as follows.

For the ICL 1900 system :

```
OPCODES
5, 1, 24
STO   AAA0001000CCBBBBBBBBBBBB
CALL  AAA011100BBBBBBBBBBBBBBB
PERI  AAA1101111CCBBBBBBBBBBBB
MOVE  AAA1010110CCBBBBBBBBBBBB
MVCH  AAA1001110CCBBBBBBBBBBBB
SLC   AAA1001000CC00BBBBBBBBBB
NGN   AAA1000010CCBBBBBBBBBBBB
ANDN  AAA1010000CCBBBBBBBBBBBB
SUSWT 0001110001BBAAAAAAAAAAAA
```

For the INTERDATA 8/32 system :

```
OPCODES

4,3,16,32,48

LR      00001000AAAABBBB

STH     01000000AAAACCCCBBBBBBBBBBBBBBBB

LIS     00100100AAAABBBB

L       01011000AAAACCCC0100DDDDBBBBBBBBBBBBBBBBBBBBBBBB

EXHR    00110100AAAABBBB

CH      01001001AAAACCCC00BBBBBBBBBBBBBB
```

4.2.2 THE PSEUDO-CODE ACTION DEFINITIONS

The start of this section is identified by the keyword CODE-SKELETONS and is followed by the various pseudo-code action definitions. The start of a pseudo-code action is identified by :

#SK pseudo-code action number, number of parameters.

The end of one definition is identified by the start of the next definition or, as in the case of the last pseudo-code action definition, the end of data marker #EOD.

The various actions, described in chapter 3, which may appear within a pseudo-code action definition are identified by means of special characters. A list of these special characters and their meanings can be found in appendix 3. Some examples of these definitions are given in appendices 10.2 and 10.3.

4.3 OUTPUT

The standard ICL ALGOL output package has been used to write the pseudo-code action table plus an index to the table to the output disc files. The pseudo-code action table is written to the output file as a continuous character stream. For each pseudo-code action the index contains the following three components.

The pseudo-code action start address within the table.

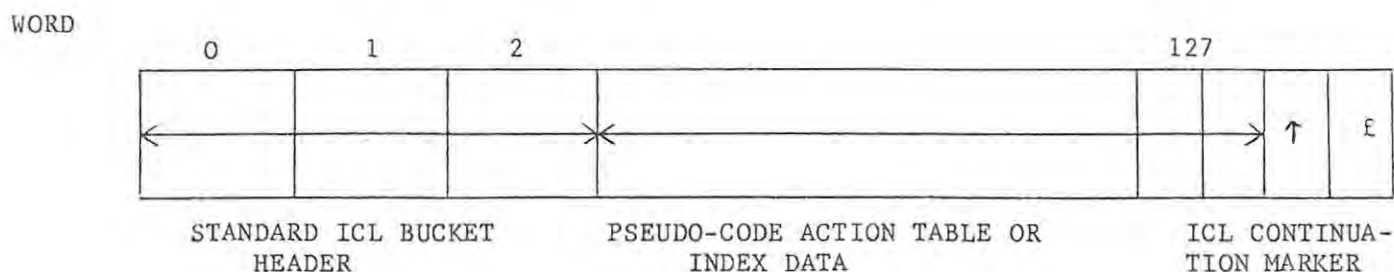
The length of the pseudo-code action.

The number of parameters required by the pseudo-code action.

Within a generated code generator the routine to fetch a pseudo-code action

is required to calculate the bucket number within which the action will appear and the starting position within the bucket from the start address in the index.

Both the pseudo-code action table and the index are written to disc files using one block (128 word) buckets with the format below.



The index file has been written with bucket one containing the number of entries in the index, while the index itself is contained in buckets 2, 3, etc. Each value in the index occupies 11 character positions being right justified and preceded by spaces.

An ALGOL program to list the pseudo-code action table together with its associated index has been written and can be found in appendix 10.4 together with some typical output from the program. As a user is required to insert the pseudo-code action for each local routine into a code generator specification this program has proved to be particularly useful.

In the pseudo-code action table the various actions, e.g. define store, address table access, etc., are identified by means of numbers as given in the table in appendix 4. All parameters described in the table are assumed to occupy 1 character position unless otherwise stated.

Several pseudo-code action definitions together with their corresponding pseudo-code action table data are given in section 4.6.

4.4 PSEUDO-CODE ACTION TABLE OPTIMIZATION

The idea of having the pseudo-code action table as a single character stream on a disc file is sufficient for the case where one is not particularly concerned with the amount of time taken by the code generator to locate a particular code skeleton. To improve the efficiency of the system for a particular code generator the user may wish to organize the pseudo-code action table in some

way so that faster access of any particular pseudo-code action can be obtained.

If this is done the user is obviously required to take this into consideration when reading the pseudo-code action table index into the appropriate area within the initial action routine and when fetching any particular pseudo-code action from the pseudo-code action table file.

An ALGOL program to organize the code skeletons in a more efficient manner for the ICL system has been written and can be found in appendix 10.5 together with a disc file dump of an optimized pseudo-code action table. The approach taken in this program was to store each pseudo-code action in such a way that each one starts at the beginning of a different bucket. If a skeleton extends to more than one bucket then this bucket will be the immediate successor of the previous one. The pseudo-code action table index field containing the start address now contains the bucket number of the bucket in which the skeleton resides.

This pseudo-code action table organization scheme has been used in the generated COBOL code generator.

4.5 THE PSEUDO-CODE ACTION DEFINITION PROCESSING PROGRAM

The structure and operation of the pseudo-code action definition processing program can best be seen by consulting the program listing in appendix 10.6.

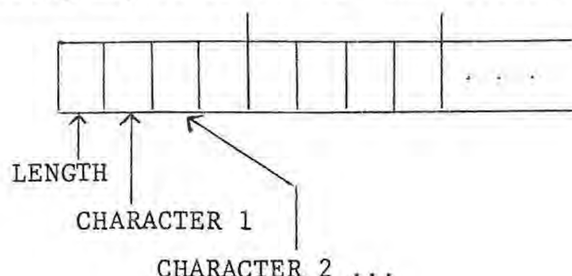
The program, written in ALGOL and incorporating a number of PLAN subroutines, consists basically of a section to read the assembly language opcodes used in the pseudo-code action definitions together with their corresponding internal machine code forms and a procedure to process the definitions. This latter procedure consists of a number of procedures to handle the various operations which may appear within the pseudo-code action definitions, e.g. fetch a register, a conditional code insertion statement, etc.

As the ICL ALGOL system does not provide a means for packing strings of characters a number of PLAN subroutines have been written and incorporated into the program to achieve this. The method of character string handling is

based on that used by BCPL as described by Richards[23] and Powell[22].

In addition to packing and unpacking strings there are routines for the comparison of two packed strings and the comparison of a packed string with a literal string. This system has been used to save storage space and to make string comparisons easier.

The ICL ALGOL system stores one character per word, thus wasting 75% of the storage space if characters are stored in this way. A packed character string may be up to 63 characters in length and is stored in the following way.



The character strings stored in this way are the opcodes as specified in the opcode to machine code mapping as well as the machine code representations themselves. It is necessary to be able to unpack strings for printing purposes and also for the analysis of instruction formats. The string comparison routines are useful for the identification of opcodes and keywords.

Another feature of this program is that constant data when located as one of the fields within an instruction is inserted into a buffer, the immediate buffer, where adjacent immediate data can be accumulated and from which immediate data may be copied to the pseudo-code action table.

If there is any data in the immediate buffer at the end of the analysis of an instruction which did not start at the beginning of a word or basic data unit then this information is extracted from the immediate buffer and inserted into the pseudo-code action table as an appropriate sequence of immediate characters and/or immediate bits. This ensures that one never has an immediate character containing information from two different adjacent instructions.

As an example of this consider the following.

OPCODES

LDX AAA0000000CCBBBBBBBBBBBB

STO AAA0001000CCBBBBBBBBBBBB

```
CODE-SKELETONS
#SK  12,3
STO   %1  1,2
LDX   3   %3(%2)
#EOD
```

The immediate part of the above pseudo-code action consists of the last 21 bits of the STO instruction and the first 10 bits of the LDX instruction. The immediate part is the binary pattern shown below.

0001000100000000000010110000000

Translating this into 5 immediate characters and 1 immediate bit would require a considerably more complex immediate character routine in the code generator. The parameters for the immediate characters action are the number of immediate characters and the characters themselves. The immediate bits action also takes two parameters, viz. the number of immediate bits and a single character containing the left aligned immediate bits. Thus the above immediate parts of the instructions would appear in the pseudo-code action table as

IMMED CHARS,3,4,0,IMMED BITS,3,8,IMMED CHARS,1,(,IMMED BITS,4,0

There is no error recovery within the program which halts with a self-explanatory error message on detection of an error.

4.6 PROCESSED EXAMPLES

Several short examples showing the code output from the pseudo-code action definition processing program are given below. The examples below cover most of the features allowed by the system.

The parameters for an opcode within a pseudo-code action definition may be separated by spaces or commas and have been written in the depicted manner merely for convenience.

```
OPCODES

4,1,24

LDX       AAA0000000CCBBBBBBBBBBBB

STO       AAA0001000CCBBBBBBBBBBBB

BCHX      AAA011010BBBBBBBBBBBBBBB

BCT       AAA011011BBBBBBBBBBBBBBB

SBN       AAA1000011CCBBBBBBBBBBBB

LDN       AAA1000000CCBBBBBBBBBBBB

SMO       0001001111BBAAAAAAAAAAAA
```

CODE-SKELETONS	PUBLICATION LANGUAGE EQUIVALENT
`#SK      12,2`	`#SK      12,2`
`£0`	`#FETCH(0)    type0=index register`
`LDX      &0  %1,0`	`LDX #REG(0)    %1,0`
`£2`	`#FETCH(2)    type2=2 adjacent general purpose registers`
`STO      @1 !%2,&0`	`STO #REG(1)-1  !%2,#REG(0)`

```
OUTPUT CODE :  (OBO?13!10 30=1?=<(2.1?13!14  1OBO?;2=2+?=<
```

`#SK      13,0`	`#SK      13,0`
`LDX      1    !5+2,0`	`LDX      1   !5+2,0`
`$0BCHX        1    *+2`	`$0  BCHX     1    *+2`
`STO      1    !6*2,0`	`STO      1   !6*2,0`

```
OUTPUT CODE : !280*0005+>002?=<"0!1;  3 -:?002!28@'000 6+,02?=<
```

`#SK      14,1`	`#SK      14,1`
`↑2(3,!%1+3)`	`↑2(3,!%1+3)`
`LDX      1    ↑3,0`	`LDX      1    ↑3,0`
`BCT      1    *-2`	`BCT      1    *-2`

```
OUTPUT CODE :  *0003<1=1+>003<2#2!280#3?=<!1; 3(-:?102
```

`#SK      15,0`	`#SK      15,0`
`BCHX        1    +`	`BCHX      1    #FORWARD`
`SBN         3    1,0`	`SBN       3    1,0`
`BCT         3    -`	`BCT       3    #UNSTACK`

```
OUTPUT CODE :  !1; 3 A?:?!5,<01+ 3(@?:?

      # SK    16,3                            # SK    16,3
       £2                                     # FETCH(2)
       LDN    @0      0,0                      LDN  # REG(1)      0,0
       LDN    @1      0,0                      LDN # REG(1)-1     0,0
       IF    !%2   >  4095                    # IF     !%2 > 4095
       THEN   SMO    !%3,0                    # THEN   SMO     !%3,0
              LDX    @0  !5+2,0                        LDX  # REG(1)    !5+2,0
       ELSE   LDX    @1  !5+2,0               # ELSE   LDX  # REG(1)-1 !5+2,0
              LDX    @0  !5+3,0                        LDX  # REG(1)    !5+3,0
       FI     STO    @0  !%2,0                # FI     STO  # REG(1)    !%2,0
              STO    @1  !%2+1,0                       STO  # REG(1)-1 !%2+1,0
```

```
OUTPUT CODE :  (2.0?13!3@00  30.1?13!3@00  30£3=2+%*
               00←←%@0E!24$=3+?=<.0?13!10 30*0005+>0
               02?=<'0Q.1?13!10 30*0005+>002?=<.0?13!
               10 30*0005+>003?=</.0?13!14 30=2+?=
               <.1?13!14 30=2+>001?=<
```

5. THE CREATION OF A COMPLETE SYSTEM

5.1 THE PRINCIPLE OF THE INTERPRETER OPERATION

5.1.1 AN OVERVIEW OF THE INTERPRETER

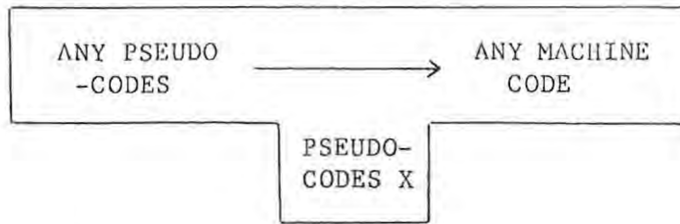
The function of the interpreter is to accept as input the pseudo-code stream from the parser and to perform the corresponding actions either directly or by reference to the pseudo-code action table.

There are a number of differentways of writing an interpreter to achieve the above function. These include writing the system in a systems programming language such as BCPL [22,23]. In this case the systems programming language should be of a sufficiently high level to be removed from any machine dependent features while simultaneously providing the low level facilities required by a system of this nature. Furthermore, a compiler for this systems programming language is required to exist on the target machine. For the mini- and micro-computers currently available very few systems programming language compilers exist and, as it was intended that the system should be useful in these cases as well, the idea of writing the interpreter in one of these languages was discarded.

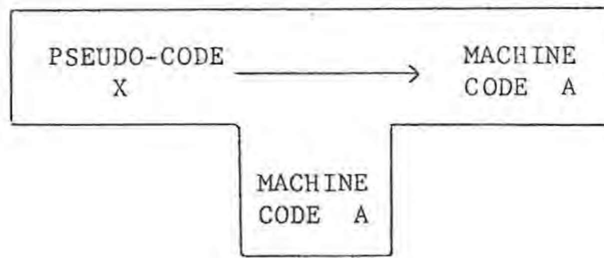
Another alternative was to write the interpreter in a suitable high level language which provides the required facilities, e.g. ALGOL, however the same problems as stated above still exist.

As it was one of the aims of the project to produce a system which would be as portable [4,21] as possible, it was decided to write the system in such a way that it would be able to generate itself or any other code generator from a set of input parameters and pseudo-codes. In order to accomplish this a high level definition of the actions required by a typical code generator was written. See chapter 3 and appendix 2.

This high level definition was then translated into a set of pseudo-codes (pseudo-code X) and a set of corresponding code skeletons or pseudo-code X actions. Thus an interpreter specified in terms of pseudo-codes existed but no interpreter capable of processing these pseudo-codes had been written. The T-diagram below shows the state of the system at this point.

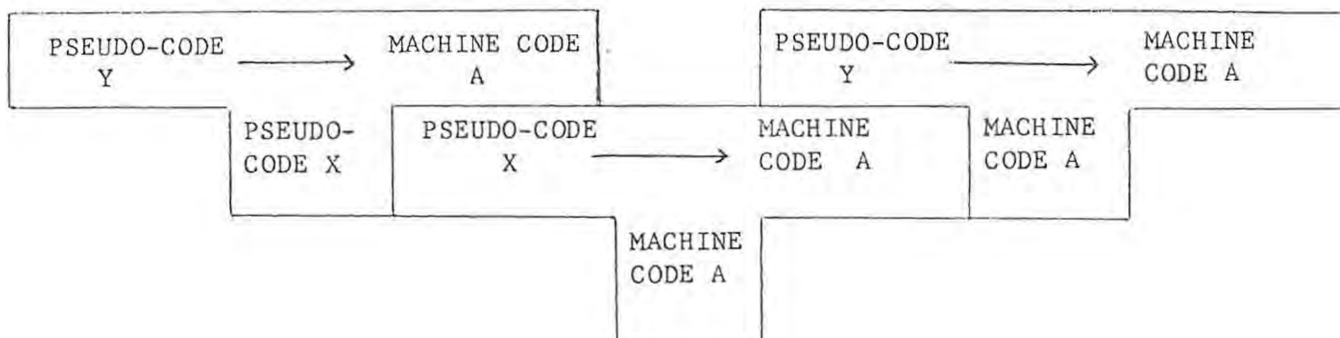


The next step in generating the system was to write a bootstrap program which would be capable of producing a version of the interpreter in machine code (machine code A). The interpreter produced by the bootstrap program can be expressed as the T-diagram below. To be able to perform the conversion of pseudo-code X to machine code A requires the inclusion of the appropriate pseudo-code action table plus the required local routines, e.g. fetch the next symbol, write a buffer, etc.

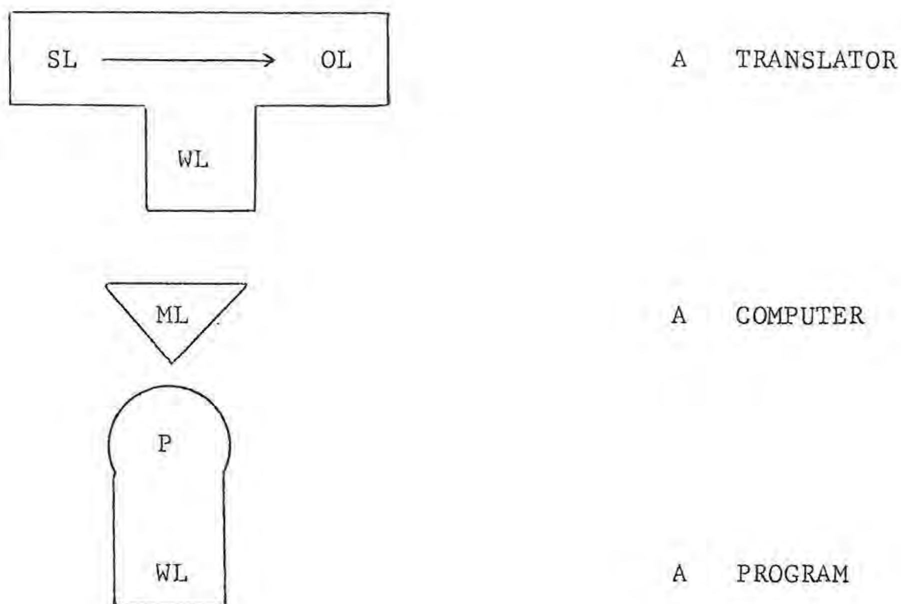


This bootstrapped code generator generator is capable of accepting any set of pseudo-codes and producing any machine code provided that the appropriate pseudo-code action table is available.

To produce a code generator for another language on the same machine involves the definition of the pseudo-code (pseudo-code Y) to machine code (machine code A) mapping and a suitable adjustment to the input parameters for the system. The input parameters consist of the definitions of areas such as the instruction buffer, data buffers, pointers, etc. This definition is used as input to the code generator generator as depicted below.



Using the notation as described by Earley and Sturgis[10] and used by Lecarme and Peyrolle-Thomas[17] in their description of the PASCAL boot-strapping system, viz.



where P = program

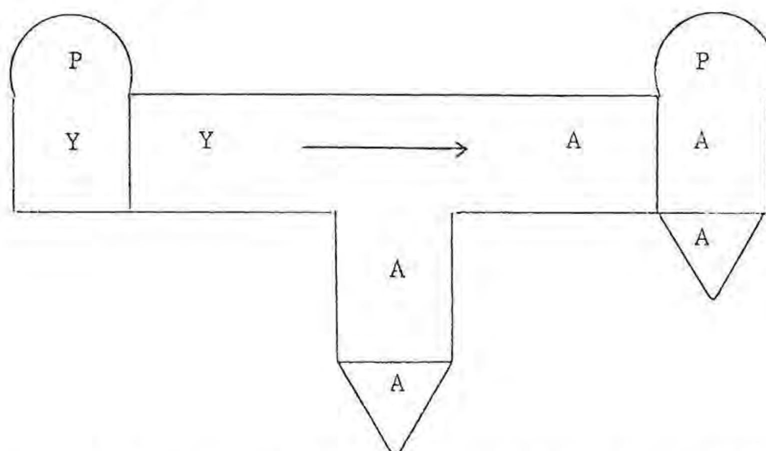
WL = writing language

SL = source language

OL = object language

ML = machine language

we now have the following.



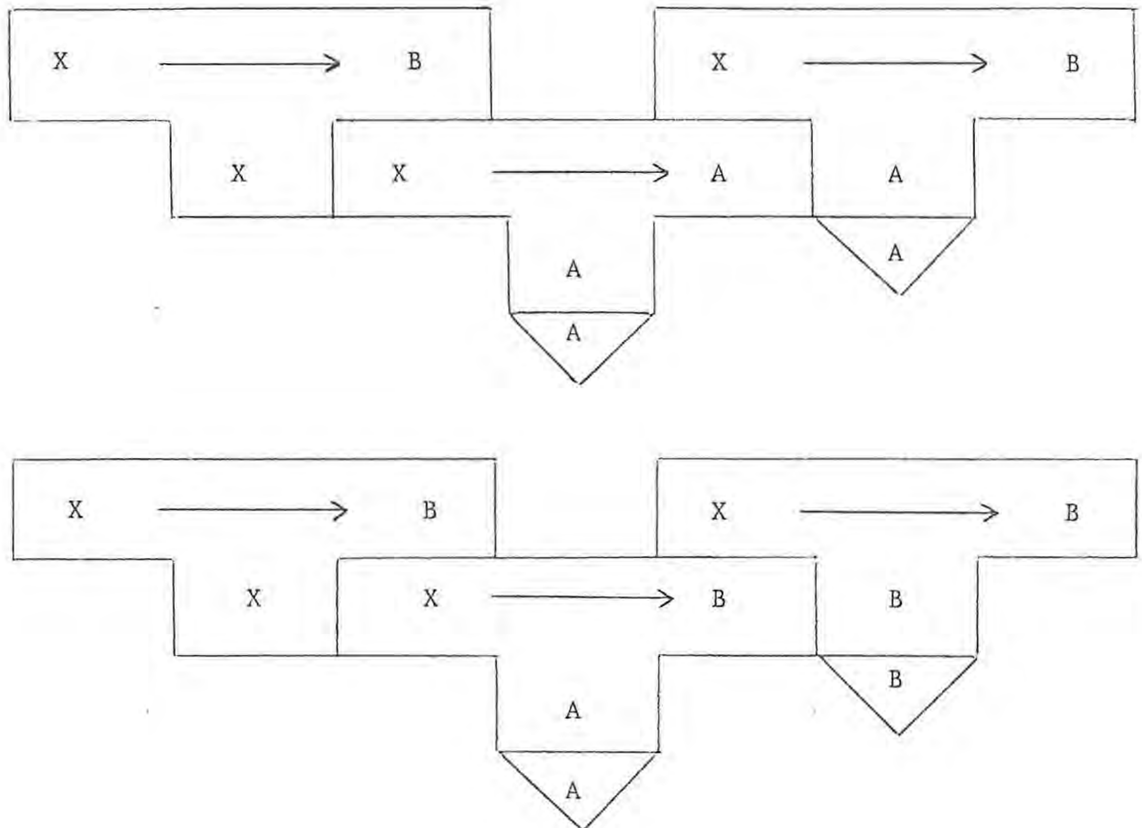
The program in pseudo-code Y is translated into machine code A.

To transport the code generator generator system to a different computer a bootstrapping system[10] is used.

X = system definition pseudo-codes

A = current system machine code

B = new system machine code



The parameters for the pseudo-code X to machine code B in pseudo-code X translator is defined in appendix 10.7 as a list of general system pseudo-codes which can be processed by the original bootstrapped code generator generator giving as output the machine code A translator which converts pseudo-code X to machine code B. A version of this translator on the 1900 system has been generated and its decompiled code can be found in appendix 10.8.

A detailed description of the transportation process and the process required to generate a particular code generator can be found in section 5.1.4.

5.1.2 THE GENERAL SYSTEM PSEUDO-CODES

The general system psuedo-codes are defined as those in terms of which the action analyzing routines are defined in order to give the system its transportable nature.

In designing the set of general pseudo-codes there were two prime objectives, viz.

(i) to produce a set of pseudo-codes in terms of which the action analyzing routines could be easily specified, and

(ii) to keep the number of registers in use at any one time to a minimum. This was set as one of the objectives to make the system useful for computers with only a few registers.

In some systems several of the pseudo-codes may be found to be redundant and have exactly the same effect as some other pseudo-code. Quite obviously this situation does not matter at all and is preferable to a system in which there exist fewer but more system dependent pseudo-codes. A few of the pseudo-codes may be found to be partially system dependent in their definitions below, e.g. the multiply and divide operations requiring multiple register operands. For the cases considered this presented no problems at all.

A list of the pseudo-codes with their functions and a description of any parameters is given below. Examples have been provided in a number of cases in order to clarify the descriptions. A complete list of the pseudo-code definitions in PLAN is given in appendix 10.3.

For ease of reference the pseudo-code definitions have been numbered starting from 12 as the definition actions are considered to be pseudo-codes 1 to 11.

12. Save the return address in the specified address table element. This pseudo-code is used on entry to a routine to save the return address for exit from the routine. In the PLAN pseudo-code action definitions one particular register is always used to contain the return address in calls to routines. This is not strictly required by the system but serves mainly to demonstrate that certain registers may have predefined functions and is useful for communication with local routines.

13. Call the routine whose start address is contained within the specified address table element. As mentioned above the system makes use of a predefined register number for this purpose. The pseudo-code action is defined as :

```
CALL    7    !%1
```

but may equally well be defined as :

```
# FETCH(2)
  CALL    # REG(2)    !%1
```

where type 2 registers are general purpose registers. The term register in the following descriptions will be taken to mean a general purpose register.

14. Fetch a register and load it with the value in the location specified by the address table element number used as the parameter.

15. Add the literal parameter to the contents of the register at the top of the register use stack.

16. Perform an arithmetic left shift on the contents of the register at the top of the register use stack. The number of places by which the value must be shifted is specified as a literal parameter.

17. Store the contents of the register at the top of the register use stack in the location whose address is contained in the address table element specified as the parameter and free the register at the top of the use stack.

18. Store the contents of the register at the top of the register use stack in the location specified by the address table element number appearing as parameter two plus the value of parameter one. Free the register at the top of the register use stack. In the ICL case this has been implemented as :

```
# FETCH(1)
 LDN    # REG(1)    %1,0
 STO    # REG(2)    !%2,# REG(1)
# FREE(1)
# FREE(2)
```

where type 1 registers are index registers.

19. Fetch an index register and load it with the value in the location specified by the address table element number used as the parameter.

20. Add the contents of the index register at the top of the index register use stack to the contents of the index register second from the top of the index register use stack and free the register at the top of the stack.

```
e.g.    ADX    #REG(1)-1    #REG(1),0
        #FREE(1)
```

21. Load an index register with the value from the location contained within the index register at the top of the index register use stack and free the index register containing the address.

e.g. LDX #REG(1) 0,#REG(1)

22. Store the contents of the index register at the top of the index register use stack in the location whose address is contained in the address table element specified as the parameter and free the index register at the top of the use stack.

23. Fetch a register and load it with the value from the location contained within the index register at the top of the index register use stack and free the index register containing the address.

24. Add the literal parameter to the contents of the index register at the top of the index register use stack.

25. Add the contents of the top two registers in the register use stack, leaving the result in the register at the top of the stack while freeing the registers containing the values to be added.

26. Subtract the literal parameter from the contents of the register at the top of the register use stack.

27. Add the contents of the register at the top of the register use stack to the index register at the top of the index register use stack and free the register at the top of the register use stack.

28. Advance the character or byte pointer in the location specified by the address table element number used as the parameter to the next character or byte position.

In the ICL 1900 system it is possible to store four characters to a word and one uses the concept of character index words for accessing individual characters[20]. The code for this action is :

```
#FETCH(1)
  LDX    #REG(1)    !%1,0
  BCHX   #REG(1)    *+1
  STO    #REG(1)    !%1,0
#FREE(1)
```

In the byte orientated INTERDATA system the code for this action is much the same.

```
# FETCH(1)
  L      #REG(1), !%1,0
  AIS    #REG(1),1
  ST     #REG(1), !%1,0
# FREE(1)
```

29. Store the contents of the register at the top of the register use stack in the location specified by the address table element number used as the parameter plus the contents of the index register at the top of the index register use stack. Return the registers at the top of both stacks to their respective free register stacks.

```
  STO    #REG(2)      !%1,#REG(1)
# FREE(1)
# FREE(2)
```

30. Fetch a register and load it with the contents of the location at the address table element number passed as the parameter plus the contents of the index register at the top of the index register use stack. Free the top index register.

31. Subtract the contents of the register at the top of the register use stack from the contents of the register second from the top of this stack. Leave the result in the register at the top of the stack and return the registers containing the operands to the free register stack.

32. Set to zero the location whose address appears in the location specified by the address table element number passed as the parameter.

```
e.g.  # FETCH(1)
        LDX   # REG(1)     !%1,0
        STOZ        0,#REG(1)
      #FREE(1)
```

33. Branch backwards unconditionally.

```
e.g.      BRN   #UNSTACK
```

34. Exit from a routine to the address in the address table element number specified as the parameter.

```
e.g.     #FETCH(2)
           LDX   #REG(2)    !%1,0
           EXIT  #REG(2)    0
         # FREE(2)
```

35. Load an index register with the value from the location specified by the address table element number used as parameter two plus the literal value used as parameter one.

36. Store the contents of the register at the top of the register use stack in the location whose address is contained in the index register at the top of the index register use stack. Free the top register and index register.

37. Fetch a register and load it with the value in the location specified by the address table element number used as parameter two plus the literal value used as parameter one.

```
e.g.     # FETCH(2)
         # FETCH(1)
           LDN   #REG(1)     %1,0
           LDX   # REG(2)    !%2, # REG(1)
         # FREE(1)
```

38. Test if the value in the register second from the top of the register use stack is less than the value in the register at the top of the stack. Free both registers.

```
ICL 1900 example   :      TXL    # REG(2)-1    # REG(2),0
                        # FREE(2)
                        # FREE(2)
INTERDATA example  :      CLR    # REG(2)-1, #REG(2)
                        # FREE(2)
                        # FREE(2)
```

39. Branch forward on a false condition.

e.g. BCC #FORWARD

40. Load an index register with the contents of the location at the address table element used as the parameter plus the contents of the index register at the top of the index register use stack and free this latter index register.

e.g. LDX #REG(1) !%1,#REG(1)

41. Subtract the literal parameter from the contents of the index register at the top of the index register use stack.

42. Branch forward unconditionally.

43. Branch forward if the register at the top of the register use stack contains a non-zero value and free this register.

44. Branch forward if the register at the top of the register use stack contains the value zero and free this register.

45. Store the literal value passed as parameter one in the address specified by the address table element number passed as parameter two.

```
e.g.    #FETCH(2)
        LDN   # REG(2)   %1,0
        STO   # REG(2)   !%2,0
        # FREE(2)
```

46. Fetch an index register and load it with the literal value specified as the parameter.

47. Fetch a register and load it with the literal value specified as the parameter.

48. Fetch an index register and load it with the contents of the register at the top of the register use stack. Return the register at the top of the register use stack to the free register stack.

49. Perform a logical left shift on the contents of the location specified by the address table element number used as the parameter. The number of places by which the value must be shifted is contained within the index register at the top of the index register use stack. Free the index register.

e.g.

```
# FETCH(2)
  LDX   # REG(2)   !%1,0
  SLL   # REG(2)   0,#REG(1)
  STO   # REG(2)   !%1,0
# FREE(2)
# FREE(1)
```

50. Perform a logical right shift on the contents of the location specified by the address table element number used as the parameter. The number of places by which the value must be shifted is contained within the index register at the top of the index register use stack. Free the index register.

51. Logically OR the value in the register second from the top of the register use stack into the storage location at the address specified by the address table element number passed as the parameter plus the contents of the register at the top of the register use stack. Free the top two registers.

ICL 1900 example:

```
#FETCH(1)
  LDX  #REG(1)   #REG(2),0
# FREE(2)
  ORS  #REG(2)   !%1,#REG(1)
# FREE(2)
# FREE(1)
```

INTERDATA example:

```
  O    #REG(2)-1,   !%1,#REG(2)
  ST   #REG(2)-1,   !%1,#REG(2)
# FREE(2)
# FREE(2)
```

In the two systems considered the multiply and divide instructions both required a two word operand or dividend, however the placing of the actual operands in adjacent registers was slightly different and users should be particularly careful in their interpretation of the following 8 pseudo-codes.

52. At this point the register use stack is considered to contain the operands for a divide instruction. On completion of the pseudo-code action the register at the top of the register use stack must contain the quotient while the registers

containing the operands are considered to have been freed.

The 1900 implementation regards the contents of the register at the top of the register stack as the divisor while the dividend is contained in the two adjacent registers immediately below this one.

```
e.g.      DVS    #REG(2)-1  #REG(2),0
          #FREE(2)
          #FREE(2)
```

The following three pseudo-code actions are concerned with the loading of a register(s) for a multiply operation which is to follow. The 1900 system requires two adjacent registers.

53. Fetch register(s) for a multiply operation and load the contents of the location specified by the address table element number used as the parameter.

```
ICL 1900 example :         #FETCH(3)
                           LDX   #REG(2)    !%1,0
```

Type 3 registers are two adjacent general purpose registers.

```
INTERDATA example :        #FETCH(3)
                           L    #REG(2)-1,  !%1,0
```

Type 3 registers are two adjacent registers with the first of the pair being an even numbered register.

54. Fetch register(s) for a multiply operation and load the contents of the location whose address is in the index register at the top of the index register use stack. Free the index register.

55. Fetch register(s) for a multiply operation and load the contents of the location specified by the address table element number used as parameter two plus the literal value used as parameter one.

56. The operands for a multiply operation are considered to be in the register use stack. On completion of the pseudo-code action the result of the multiplication is assumed to be in the single register at the top of the register use stack while the registers containing the operands are considered to have been freed.

```
e.g.      MPY    #REG(2)-1    # REG(2),0
          # FREE(2)
          # FREE(2)
```

The following three pseudo-codes are concerned with loading the dividend for a divide operation into one or more registers. The 1900 system requires that the dividend be contained within two adjacent registers.

57. Fetch register(s) and load the contents of the location specified by the address table element number used as the parameter.

```
ICL 1900 example :        # FETCH(3)
                           LDX    # REG(2)-1    !%1,0

INTERDATA example :       # FETCH(3)
                           LIS    # REG(2),0
                           L      # REG(2)-1,   !%1,0
```

58. Fetch register(s) and load the contents of the location whose address is contained within the index register at the top of the index register use stack. Free the index register.

59. Fetch register(s) and load the contents of the location specified by the address table element number used as parameter two plus the literal value used as parameter one.

60. Branch backwards if the register at the top of the register use stack contains a non-zero value and free the register.

```
e.g.      BNZ    # REG(2)  # UNSTACK
          # FREE(2)
```

61. Call the routine whose starting address is currently in the index register at the top of the index register use stack and then free this index register.

```
e.g.      SMO    # REG(1),0
          CALL   7   0
          # FREE(1)
```

62. Branch backwards if the contents of the location specified by the address table element number used as parameter one is less than the contents of the location specified by the address table element number used as parameter two.

ICL 1900 example:

```
#FETCH(2)
 LDX    #REG(2)    !%1,0
  SLC   #REG(2)    2,0      necessary because of
  ANDN  #REG(2)    4095,0   the use of character
                            index words.
  TXL   #REG(2)    !%2,0
  BCS   #UNSTACK
#FREE(2)
```

INTERDATA example:

```
#FETCH(2)
  L     #REG(2),   !%1,0
  C     #REG(2),   !%2,0
  BTC      8,#UNSTACK
# FREE(2)
```

63. Test if the contents of the registers at the top and second from the top of the register use stack are unequal. Return both registers to the free register stack.

64. Branch forward on a true condition.

65. Halt and display the literal used as the parameter.

66. Fetch a register and load it with the single character in the location specified by the address table element number used as the parameter plus the contents of the index register at the top of the index register use stack. Free the index register.

67. Load an index register with the single character in the location specified by the address table element number used as the parameter plus the contents of the index register at the top of the index register use stack. Free this latter index register.

68. Negate the contents of the register at the top of the register use stack.

69. Branch backwards if the register at the top of the register use stack contains a positive value and free the register.

70. Branch forwards if the register at the top of the register use stack contains a positive value and free the register at the top of the register use stack.

71. Move the number of words specified as the literal parameter from the address in the register at the top of the register use stack to the address in the register second from the top of the stack and return both registers to the free register stack.

5.1.3 SYSTEM REQUIRED LOCAL ROUTINES

In this section the local routines required by any code generator or code generator generator for input and output are described together with their data area requirements.

An implementation of these local routines for the ICL 1900 can be found in appendix 10.9. Within these routines the return address to any calling routine is always located in accumulator 7. As mentioned in section 5.1.2 this has been used as a standard throughout the 1900 implementation.

FETCH A PSEUDO-CODE ACTION

The local routine to fetch a pseudo-code action makes use of the pseudo-code action table index and a buffer which is used to contain the appropriate action whose interpretation is required. The index should be initialized with the appropriate values either by means of a define constant action in the data area specifications or by means of some procedure within the initialization action routine whereby the index is read from the appropriate disc file.

As mentioned in section 5.1.2 the user-defined pseudo-code actions should be numbered from twelve as the directly executed definition actions are considered to be pseudo-codes 1 to 11.

The index and the pseudo-code action buffer, which must obviously be large enough to contain the largest defined pseudo-code action, should have their start addresses in address table elements one and two respectively.

The pseudo-code action which is to be fetched has its number inserted into the location specified by address table element 104 while the routine is required to return the length of the pseudo-code action sequence in the location specified by address table element 113. On exit from the routine the location specified by address table element 104 need no longer contain the pseudo-code action number and thus this location may be used as a temporary storage area within the routine.

As the pseudo-code action is inserted into the buffer the start address of this buffer is required. For this purpose address table element number 45 is required to be an address constant for address table element 2. Furthermore a control area for disc file read operations from the file containing the pseudo-code action table is required. This is required to begin at the location specified by address table element 44.

In the case of fetching a pseudo-code action from an optimized table (see section 4.4) it will probably be possible to read the action directly into the pseudo-code action buffer. However, in the case of a non-optimized table this may not be possible and an additional buffer may be required into which the data is first read then manipulated in some way and finally moved to the pseudo-code action buffer. If such a buffer is required its start address should be contained within address table element 105 while address table element 106 should be defined as an address constant containing address 105.

FETCH THE NEXT SYMBOL

This routine is used to fetch pseudo-codes and their parameters from the input stream, but as the definition actions are regarded as pseudo-codes and may themselves appear within a pseudo-code action the routine must first check the flag indicating the input source, i.e. the input stream or the pseudo-code action

buffer, and fetch the next symbol from the appropriate source. The value returned by the routine is inserted into a standard location whence it may be accessed by the calling routine.

The required data areas and their associated address table element numbers are as given below.

DATA AREA FUNCTION	ADDRESS TABLE ELEMENT NUMBER
A disc buffer	46
The amount of unreferenced data currently remaining within the disc buffer	47
A pointer to the current position within the buffer	48
A control area for the read from disc file operations	49
The bucket number of the bucket currently in the disc buffer	50
Flag indicating source	51
The standard location into which the symbol is to be inserted	58
The start address of the disc buffer	103

WRITE A BUFFER TO A DISC FILE

The function of this routine is to write the contents of a buffer to the disc file which is to be used as input to the loader program. In general there will be several different buffers whose contents will have to be written to the output disc file. In the simplest case there would be 2 buffers, i.e. an instruction buffer and a data area buffer. (See code skeleton 4 in appendix 10.9).

The routine is also responsible for the setting up of loader information

for each buffer which is written to the output file. The complexity of the routine will obviously depend upon the complexity of the system involved. A system having several data area buffers, an instruction buffer and a relocation buffer will require a far more complex routine than the one in appendix 10.9.

The routine defined in appendix 10.9 is probably the simplest possible case as the start address in the final object program of the contents of the buffer is obtained from the fourth component of the data block for the required buffer and inserted into the first word of the bucket of output data.

When the data areas and constants for a code generator are defined the user is required to define address table element 35 as containing the start address of an area containing the following addresses.

The instruction buffer data block

The instruction buffer

The relocation information buffer data block

The relocation buffer

The data block for data area 1

Data area 1 buffer

The data block for data area 2

etc.

The parameter for this routine, which is stored in the location specified by the address table element 96, is the position of the data block for the data buffer within the area associated with address table element 35. For example, if the parameter contains the value 4 then this indicates that the routine is required to write away the contents of the data area 1 buffer.

The data areas and the associated address table element numbers are as follows:

DATA AREA FUNCTION	ADDRESS TABLE ELEMENT NUMBER
An output control area for the disc file write operations	40
The bucket number of the last output bucket	41
A disc buffer	42
The start address of the above disc buffer	43

Other system dependent routines include the initial and final action routines which can be found in appendix 10.2 as pseudo-code actions 19 and 20 respectively. A different final action routine has been used in the case of the COBOL code generator and can be found in appendix 10.10 as pseudo-code action 18.

The concept of local routines being included within a generated object program lends a vast amount of flexibility to the system as a user wishing to employ some feature not provided by the system may simply include the feature in one or more local routines. An example of this can be found in the COBOL code generator where routines have been included for the handling of paragraph names and system labels.

5.1.4. CODE GENERATOR PRODUCTION

This section defines the procedure required in order to be able to generate a code generator using the bootstrapped code generator generator. Two different cases will be considered. These are:

(a) generating a code generator for the ICL 1900 system, and

(b) generating a code generator for some other system.

Using the bootstrapped code generator generator on the 1900 system it is possible to generate code generators for the 1900 system or for any other system. This latter case implies that the complete system, i.e. the code generator generator, can be propagated to some other machine since the production of a code generator generator can be regarded as a special case of the production of a code generator.

a) GENERATING A CODE GENERATOR FOR THE ICL 1900 SYSTEM

CREATION OF THE CODE GENERATOR DEFINITION DATA

The first step in the generation of a code generator for the 1900 system is the creation of the code generator definition data. In order to set up this definition data for the code generator generator to be able to produce the desired code generator the user should take note of the following points.

1. Only those facilities and corresponding data areas which are required in the code generator to be produced need be included in the code generator definition data. Thus for a code generator not using the register allocation system the data areas and pseudo-code actions concerned with register allocation should be omitted.

2. Routines for fetching a pseudo-code action, writing a buffer to a disc file and fetching the next symbol have been written for the 1900 system and can be found in appendix 10.9. If no changes to these routines are required then the corresponding pseudo-code actions can be incorporated directly. If changes are required then the routines must be written and processed by the pseudo-code action definition processing program and the appropriate pseudo-code action sequences inserted into the data for the code generator generator.

3. If any local routines are required within the code generator then these routines must be written and then processed by the pseudo-code action definition processing program. The pseudo-code action sequences produced by this program must be inserted into the code generator definition data. Any data areas required by the routines must also be defined. An example of this can be seen in the case of the COBOL code generator which incorporates local routines for the handling of paragraph names, system labels, etc. (See appendix 10.1.)

4. As the code generator under consideration is to be generated for the 1900 system the user may define the actions in terms of the locally optimized pseudo-code actions or in terms of the general pseudo-code actions. Two different COBOL code generators which produce identical results have been defined. One is defined in terms of the optimized pseudo-code actions and the other in terms of the general pseudo-code actions.

PROCESSING

At this stage it is assumed that the user has created a set of data which when used as input to the code generator generator will result in the generation of the appropriate code generator. In order to do this the following procedure should be followed.

1. The code generator definition data must be inserted into a disc file for input to the code generator generator. This is done by means of the ALGOL program described in section 5.2.3. Examples of such data can be found in appendices 10.1 and 10.12 for the COBOL code generator defined in terms of the locally optimized pseudo-code actions and the general pseudo-code actions respectively.

2. Create the pseudo-code action table corresponding to the pseudo-codes used in the definition of the code generator on the correct disc file. In order to achieve this the appropriate pseudo-code action definitions are used as input to the pseudo-code action definition processing program. Examples of this data can be found in appendices 10.10 and 10.13. The former contains the locally optimized pseudo-code action definitions and the latter the general pseudo-code action definitions. If different initial and final actions are required then they should be written and substituted for the current initial and final actions in the pseudo-code action definitions.

3. Load and run the code generator generator. The defined code generator will be produced as output on the specified file in a loadable form as described in section 5.2.2.

4. A loader program for the 1900 system has been written (appendix 10.14) and can be run to load the code generator into store whence it may be filed if so desired.

5. Before the generated code generator can be run the appropriate pseudo-code action table must be set up on a file. In order to do this the user is once again required to run the pseudo-code action definition processing program. The input will be the pseudo-code action definitions which are to be referenced in the input to the code generator. Depending upon the fetch pseudo-code action routine within the generated code generator it may be required to optimize the pseudo-code action table by means of the pseudo-code action table optimizing program in appendix 10.5.

At this stage the user has a code generator as defined by the input parameters which will produce code as defined by the pseudo-code action table.

b) GENERATING A CODE GENERATOR FOR A DIFFERENT SYSTEM

The procedure to be carried out in order to generate a code generator for a different system is similar to that described in the previous section.

Once again the user is required to select those actions which will be required in the code generator to be produced together with the data areas required for the implementation of the actions. The actions should be defined in terms of the general pseudo-codes described in section 5.1.2.

The routines to fetch the next symbol, fetch a pseudo-code action, write a buffer to a file and any other local routines which may be required in the code generator must be written by the user in his defined assembly language. These routines together with the opcode to machine code mapping are then used as input to the pseudo-code action definition processing program. The pseudo-code action sequences obtained as output are then incorporated into the rest of the code generator definition data described above.

At this point the user now has the data which is to be input to the code generator generator. This data consists of the required actions defined in terms of the general pseudo-codes, the appropriate data area definitions and the pseudo-code action sequences for the routines to be included in the code generator to be produced. This data must be set up in the input file to the code generator generator by the program described in section 5.2.3.

Before the code generator generator can be run the code skeletons for the general pseudo-code actions must be written by the user in his defined assembly language together with code skeletons for the initial and final actions and any other actions which are to be defined differently. These definitions and the appropriate opcode to machine code mapping are then used as input to the pseudo-code action definition processing program in order to produce the correct pseudo-code action table for the processing of the code generator definition data.

Since the register allocation system for a different machine may well be different to that of the 1900 system the user may be required, as an intermediate step,to generate a different code generator generator on the 1900 system which incorporates routines to perform the particular type of register manipulation required.

In order to produce the required code generator the user is simply required to load and run the code generator generator. The defined code generator will be produced on the 1900 system and must be transferred to the new machine.

There are two ways in which the pseudo-code action table required by the code generator can be produced.

1. The pseudo-code action definition processing program can be transferred to the new machine and the pseudo-code action table required by the code generator can be produced locally.

2. The pseudo-code actions may be processed by the pseudo-code action definition processing program on the 1900 system and the output transferred to the new system. The appropriate character conversions will be required to be made.

Once the preceding process has been carried out the user will be in possession of a code generator and a suitable pseudo-code action table on the new system.

PRODUCTION OF A CODE GENERATOR GENERATOR

As mentioned previously the generation of a code generator generator is just a special case of the generation of a code generator for a different system. Thus to generate a code generator generator for a different system simply requires the procedure as for generating a code generator, but this time the input to the code generator generator consists of a definition of a code generator generator in terms of the general pseudo-codes, i.e. all the functions now provided by the system must be included in the definition.

Such a definition of the system for the generation of a code generator generator on the 1900 system can be found in appendix 10.7 and is described in section 5.2.3.

5.2 IMPLEMENTATION FOR THE 1900

5.2.1 THE BOOTSTRAP PROGRAM

From the definition of the code generator generator in terms of a set of pseudo-codes a bootstrap program was written which produced a machine code version of the code generator generator from this definition. In order to achieve this the pseudo-code action table corresponding to the pseudo-codes used in the definition was set up on a file by the pseudo-code action definition processing program.

The bootstrap program was written in ALGOL and incorporated a number of routines written in PLAN. ALGOL was chosen as the programming language chiefly because the index to the pseudo-code action table would have to be read from the disc file to which it had been written by the pseudo-code action definition processing program. As the index had been written to the file by means of the ALGOL output package it would be most easily read from the file by means of the ALGOL input system.

The ALGOL input system has not been used for the input of the pseudo-code action table as it is stream oriented and thus to get to a particular character position within the input stream it is required that all the preceding characters be read. To avoid this problem two routines from a disc handling package, written as an earlier project, have been used. These routines are called ASINALG which is a procedure to open a disc file from an ALGOL program and READSR which is a procedure to read a specified bucket from a disc file.

Other PLAN routines included within the program are used for writing buckets of data to an output file in a format suitable for input to a loader, for performing left and right logical shifts and for unpacking characters.

A listing of the program can be found in appendix 10.15.

INPUT

Since this program is merely a bootstrap program the range of facilities allowed as input to the program and the range of actions which may appear in the pseudo-code action definitions are limited to a set which is just sufficient to enable a code generator generator to be defined.

The set of allowable definition actions consists of those used to define store, constants, address constants, local routines and the current address. No definition actions may appear within a pseudo-code action definition. The set of code skeleton actions which may appear within a pseudo-code action definition is limited to the expressions defined in section 3.3.3 with the exception of the user function calls.

The data input to the program consists of :

the absolute location at which the data area definitions are to begin

store, constant and address constant definitions

local routine definitions

current address definitions and pseudo-codes

the data terminator.

The format of the pseudo-codes is the same as that required by the output code generator generator with the exception that no data area numbers are specified. The data for the program must be specified in the above mentioned order as the system makes use of only one buffer in which all data items and code skeletons are assembled. The actual data for the bootstrap program can be found in appendix 10.16.

Since the set of actions which may appear within a pseudo-code action is limited to those mentioned above, it is obvious that the pseudo-code actions used by the bootstrap program had to be system dependent, for exmple, no automatic register allocation system exists. The pseudo-code actions referenced in the bootstrap program data were in fact locally optimized ones where one code skeleton existed for each of the actions as specified in chapter 3. These code skeletons can be found in appendix 10.2. A user wishing to produce a more efficient code generator generator or more efficient code generators for some other system may find it useful to write equivalent code skeletons in his defined assembly language.

The sequences of general pseudo-codes in appendix 10.7 used to define the action analyzing routines in the machine independent code generator generator data definition produce exactly the same effect as the single pseudo-codes in appendix 10.11 which contains the data for the generation of a locally optimized code generator generator on the 1900 system.

OUTPUT

The output from the bootstrap program consists of a list of address table element numbers together with their corresponding addresses in storage. This list was output as an aid for the checking of addresses in the output binary program. In addition to this a binary program in a form suitable for input to a loader program is output to a disc file.

There is no error recovery procedure within the bootstrap program and the program halts with a self-explanatory error message on detection of an error.

The operation of the bootstrap program can be represented diagramatically as illustrated overleaf.

5.2.2 A LOADER FOR THE BOOTSTRAPPED CODE GENERATOR GENERATOR

In order to load the binary program output from the bootstrap program a loader is required. The input to the loader is the information on the disc file output from the bootstrap program. The loader produced falls into the category of absolute loaders as defined by Donovan[9].

The format of the information on the disc file is as follows.

Bucket 1. Word 0 : The four character program name.

Word 1 : The address of the start of the control segment of the binary program. To produce the start address the system requires the definition of address table element 108 immediately before the start of the control segment. This is done by means of an address table element definition for the current address at the appropriate point in the input parameters.

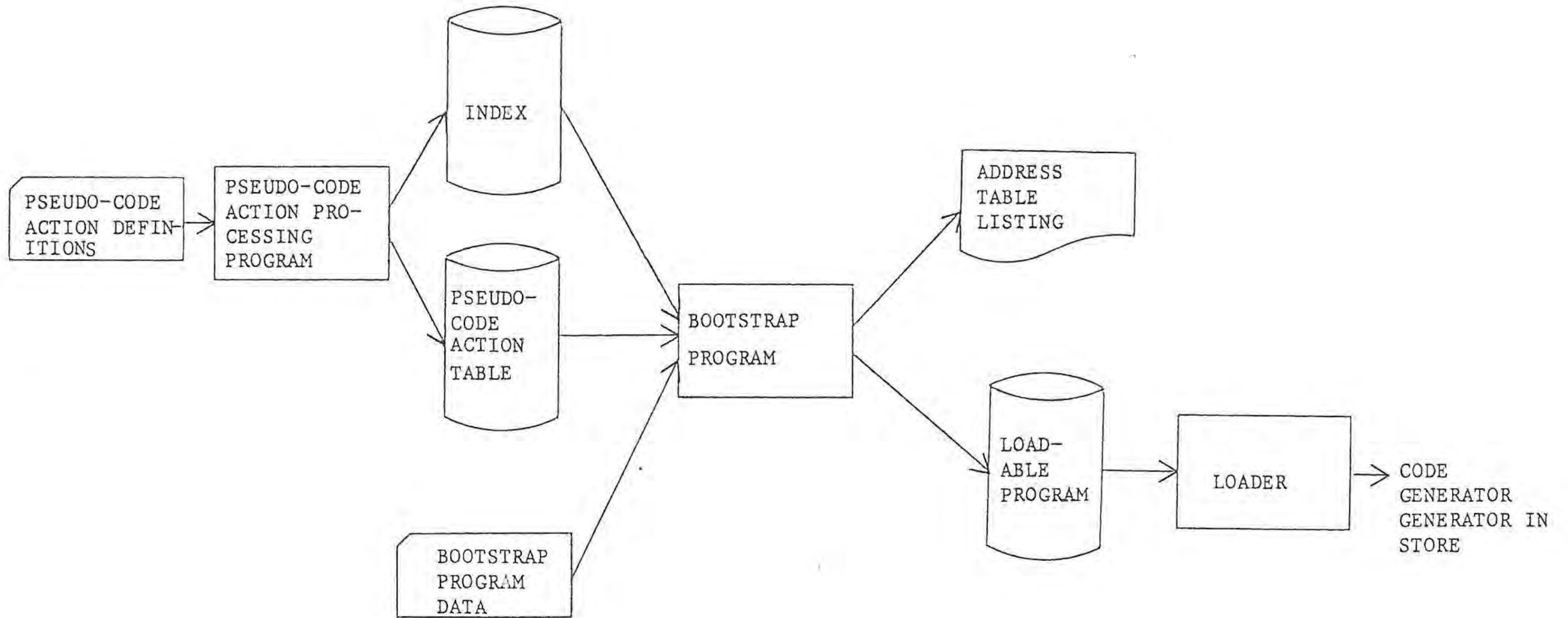
Words 2-127: 0

Bucket 2. etc. Word 0 : The start address of the data in the bucket in the binary program when the binary program is in store.

Words 1-127: The data to be inserted into storage.

Final Bucket.Word 0 : -1

THE BOOTSTRAP PROCESS



The loadable program as output from the bootstrap program can be found in appendix 10.17.

A listing of the loader can be found in appendix 10.14. The loader is a relatively simple one and operates in the following way. On entry to the loader program the output file is opened and the part of the program which performs the reading of the loadable binary program is moved up to the top of storage such that it begins at location 18000. Control is then transferred to the instruction at location 18000.

The idea of moving the program to the top of storage results in problems as far as relative branches within the moved code are concerned since the absolute addresses contained within the moved instructions will no longer be correct. This problem was overcome by calculating the difference between the address of the start of the moved code and 18000 and using this value as a supplementary modifier to the branch instructions.

The code for performing the loading is divided into two sections. The first section merely reads buckets, starting from bucket two, and moves the associated data into the correct place in storage until a bucket containing a negative value in word 0 is encountered.

The second part of the code reads bucket one which contains the program name and the start address of the control segment. The program name is inserted into the program request slip[5] and the start address of the control segment is inserted into the operand field of an unconditional branch instruction which is subsequently inserted into word 20 of the program. Word 20 of a program produced by the ICL system is considered to contain an instruction which when executed constitutes the entry point zero[20]. On completion of this operation the loader overwrites itself with zeroes and halts with the message HALTED LD. At this point the binary program exists in store in a state ready to be run.

A disadvantage of a loader of this type is that it is not possible to change the program size in the request slip and thus the request slip will contain a storage requirement considerably larger than necessary. However, this does not affect the execution of the program in any way and can thus be ignored.

5.2.3 TESTING THE CODE GENERATOR GENERATOR

A number of systems have been generated by the bootstrapped code generator generator and although not every possible can has been tested or can be tested in a reasonable amount of time a fairly wide range of test examples has been successfully produced.

1) A code generator generator defined in terms of the general pseudo-code actions described in section 5.1.2.
2) A code generator generator defined in terms of the locally optimized pseudo-code actions used by the bootstrap program. (See section 5.2.1)
3) A code generator for the COBOL subset defined in terms of the general pseudo-code actions.
4) A code generator for the COBOL subset defined in terms of the locally optimized pseudo-code actions.

With respect to its processing characteristics the code generator generator was designed in such a way as to be indistinguishable from any object program it produced. Since a parser would produce pseudo-codes on a disc file for input to the code generator, the input for the code generator generator was also defined as being on a disc file.

A program to store the parameters on a disc file for the code generator generator has been written and a listing is contained in appendix 10.18. In order to be able to run this program the pseudo-code action definitions to be used by the code generator generator must have been processed by the pseudo-code action definition processing program. This is required as the pseudo-code action table index is used by the data set-up program. The index is required to enable the program to determine how many parameters are required by any particular pseudo-code action when the pseudo-code is encountered in the code generator definition data.

This program, written in ALGOL, operates in the following way. First the pseudo-code action table index is read into an array. Then the pseudo-codes together with their parameters are read and inserted into an output buffer which is written to a disc file. This disc file becomes the input to the code generator

generator. Execution ends on encountering the data terminator.

1) A GENERAL PSEUDO-CODE CODE GENERATOR GENERATOR

The purpose of this example was to show that the code generator generator is capable of reproducing itself, i.e. of generating itself, when defined in terms of the general pseudo-codes.

As this system has been successfully generated, it means that to generate the system for any other machine simply involves writing the local routines and the definitions of the general pseudo-codes. See section 5.1.4.

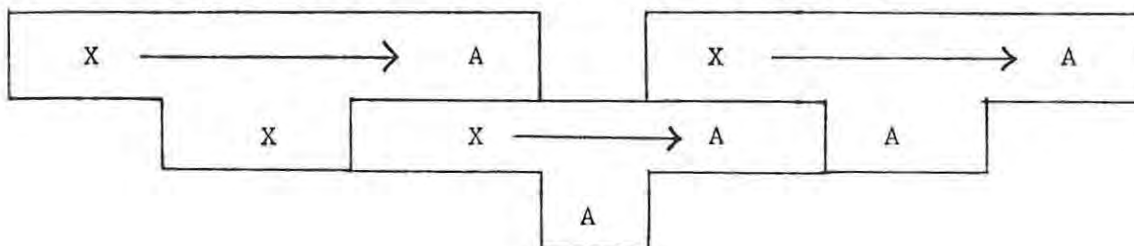
As a further test the generated code generator generator was run using the data used for generating itself to check that this version was working correctly. An identical code generator generator was produced. The decompiled version of this code generator generator can be found in appendix 10.8.

2) A LOCALLY OPTIMIZED CODE GENERATOR GENERATOR

This test case was constructed from the data used by the bootstrap program and in all respects was the same except that the initial start address was omitted. This could be done as the start address was now defined as the current absolute position within the data block for data area 1. In addition, data area numbers were inserted, as the second parameter, into all define store, constant and address constant definitions and a reference to the final action routine was included. The definition data for the code generator generator can be found in appendix 10.11.

The purpose of this test case was to prove that the system is capable of reproducing itself, i.e. of generating a code generator generator. The output code generator generator was identical to the one which produced it thus proving the above conjecture.

In T-diagram notation this situation can be represented as :



where X represents the pseudo-codes used and

A represents the ICL 1900 machine code.

A decompiled version of this code generator generator can be found in appendix 10.19.

The produced system used an optimized pseudo-code action table for improved efficiency and can be represented as in the diagram overleaf.

3) A GENERAL PSEUDO-CODE COBOL CODE GENERATOR

This test case was identical to case 4 as far as the code skeletons used by the output code generator were concerned. However, the pseudo-code action processing routines, with the exception of the system dependent routines such as the initialization action, write a buffer, etc, were now defined in terms of the general pseudo-codes.

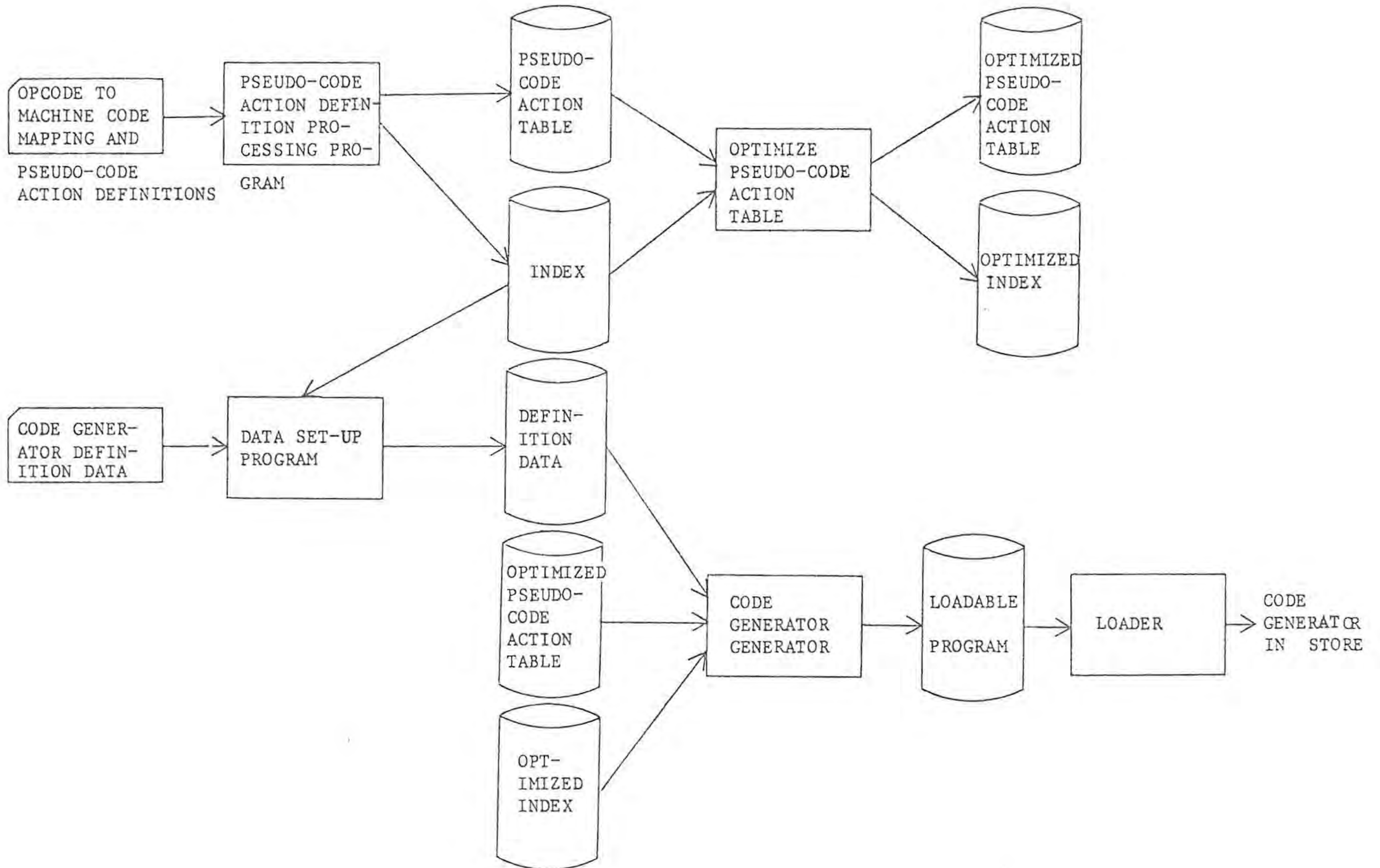
The code generator definition data for this example can be found in appendix 10.12 and appendix 10.21 contains the decompiled version of the code generator.

The input and output for this code generator and the one described in section 4 were identical.

4) A LOCALLY OPTIMIZED COBOL CODE GENERATOR

This test case was concerned with the production of a COBOL code generator to interface with the syntax analyzer described in chapters 6 and 7. The locally optimized pseudo-code action table was used in generating the system thus ensuring that the code generator would be as efficient as possible. The code generator produced functions correctly producing valid and correct object programs corresponding to the COBOL source programs input to the parser. This system is described in greater detail in section 7.9. A decompiled version of the code generator can be found in appendix 10.22.

THE OPERATION OF AN OPTIMIZED CODE GENERATOR GENERATOR



6. THE COBOL COMPILER-PART I

FORMAL SPECIFICATION AND THE DATA DIVISION

6.1 INTRODUCTION

In the previous project, when work was originally started on the COBOL compiler, the intention was to produce a COBOL compiler which would be faster than the one currently in operation on the ICL 1900 system at Rhodes University. The compiler would be implemented for a subset of COBOL and was intended to be used chiefly for the compilation of COBOL programs written by undergraduate students.

After several months work on this project it was realized that because of the time constraints and the problems which were encountered in the preliminary analysis of the compiler requirements it would not be possible to complete the project or in fact to get much further than producing a formal specification of the IDENTIFICATION, ENVIRONMENT and DATA DIVISIONS, writing a lexical analyzer and possibly writing the routines to implement the formal specification of the first three divisions of the COBOL compiler.

An extended BNF[25] formal specification of the COBOL compiler as far as the end of the DATA DIVISION was produced, however, this later required several alterations when the PROCEDURE DIVISION requirements were considered. The lexical analyzer and the routines to analyze the IDENTIFICATION and ENVIRONMENT DIVISIONS were written and tested and found to be working as required. In addition a set of routines to analyze PICTURE clauses was written and tested. It should be noted that the picture clause options have been slightly restricted, however, the restrictions are few and should not affect student users at all. For example, DB, CR and CS have been excluded from the set of allowable symbols which may appear within a picture clause character string.

An attempt was made to implement the syntax analyzer specification for the DATA DIVISION, however, because of the time constraints, this was not done properly and necessitated the rewriting of a large portion of the code.

The implemented COBOL subset was based on the article 'Mini Cobol' by P. Giles[13] and also included a number of additional facilities which were considered to be particularly useful for student type programs. The subset can be found in appendix 5. The subset includes most of the frequently used parts of COBOL, but by avoiding the full implementation the efficiency could be improved. The implementation of the subset was based on COBOL 60 as defined by Maginnis[18], Chai and Chai[6] and the ICL 1900 COBOL manual[7].

In particular the compiler does not have the facility to include any external subroutines or segments existing in a semi-compiled form on some library[8]. Thus there is no consolidation to be performed before the compiled program can be loaded into storage and run. This has resulted in a fairly large time saving as compared with the amount of time taken by the ICL COBOL compiler XEKB.

6.2 THE GENERAL IMPLEMENTATION APPROACH

The compiler consists of three basic parts, viz. a lexical analyzer, a syntax analyzer and a code generator, with one central module in overall control. The syntax analyzer is called by the control module on completion of the initialization process.

The syntax analyzer checks the syntax of the COBOL program and reports any errors present in the COBOL source. If no errors are detected by the syntax analyzer it will generate the appropriate pseudo-codes which will later be converted into actual code by the code generator.

When a source symbol is required by the syntax analyzer it calls the lexical analyzer which, in the process of extracting the next symbol, also does a certain amount of error checking. For example, it checks that identifiers do not exceed 30 characters in length.

During the syntax analysis of the ENVIRONMENT and DATA DIVISIONS a symbol table containing identifier names, etc. is built up for use in that part of the syntax analyzer concerned with the analysis of the PROCEDURE DIVISION.

If errors are detected during the syntax analysis then on completion of this phase no further action is taken by the compiler which halts with an appro-

priate message. However, if the source program has been found to be error free then the code generator is loaded into store and entered at the appropriate point. At this stage neither the syntax analyzer, the lexical analyzer nor any of the used data areas are required and thus the code generator is loaded into the storage space previously occupied by these routines and data areas.

From the list of pseudo-codes output by the syntax analyser and the pseudo-code action table, which exists in an appropriate disc file, the code generator produces a binary program as defined by the pseudo-code list. This binary program is output in a loadable form to a disc file whence it may be loaded by means of a suitable loader program.

6.3 THE FORMAL SPECIFICATION NOTATION

The syntax analyzer was designed using a finite state machine approach and by specifying the actions to be taken, i.e. the syntax interpretation rules[25] or static semantics[14], at various points in the finite state machine on recognition of the various syntactic structures.

The actions have been specified in a modified form of extended BNF as described by Williams[25]. Extended BNF was chosen as the means for formally specifying the syntax analyzer as it reflects the methods which are used by a compiler, for example, the searching of stacks, the insertion of items into stacks, etc. Using this notation it was possible to formally specify the entire syntax analyzer, and in this way all the problem areas were located and analyzed completely, before an implementation of the syntax analyzer was attempted.

Thus time was saved by avoiding extensive changes to already implemented parts or complete system rewrites necessitated by changing requirements in parts of the syntax analyzer not initially considered.

In addition to this obvious advantage it also means that this complete compiler oriented definition can be implemented with relative ease on any system for which the student COBOL compiler is desired.

A description of the notation is given below.

6.3.1 DATA STRUCTURES

The data structures used by the notation may be of two types, viz. simple and complex items.

1. SIMPLE ITEMS

These may take the following forms:

1) The strings represented by the metalinguistic variables in the BNF specification, e.g. <identifier> , <paragraph-name> .

2) String constants enclosed within quotes, e.g. "RECORD".

3) Numeric constants, e.g. 10.

4) The null item λ .

5) Replacement items. These are items enclosed within single quotes which are given some meaningful name which will be replaced by some value, e.g. 'SYSTEM LABEL'.

2. COMPLEX ITEMS

1) Tree structured stacks which may be regarded as stacks with links from one component to the one immediately above it. The elements of the stack generally contain n-tuples of pointers which may point to single values, i.e. items of information, or to another stack. A stack name will be represented by a string of upper case letters, e.g. ST.

2) System variables which may contain pointers to simple items or to stack entries. System variables consist of a string of lower case letters.

3) Component pointers. Components of stack entries or system variables may be individually accessed by means of the following :

   COMP n refers to the nth component of the current stack entry.

   COMP n(S) refers to the nth component of the n-tuple pointed to by S, where S may be a system variable or a component pointer.

6.3.2 ACTIONS

The actions which appear in the formal specification are as follows.

1) Push down an item i onto stack S : $i \downarrow S$.

2) Access the current element of stack S, inserting the resulting item into variable i : $S \uparrow i$.

3) Pop up the whole of stack S inserting the result as a single item into variable i : $i \Uparrow S$.

4) Add all the items in stack S to T as a single element : $S \Downarrow T$. T may be a stack or a single variable.

5) Search a stack S for a particular item i : $\mathcal{S}$(S,i, success action, fail action). This action causes stack S to be searched for item i. If item i is found in the stack then the success action is performed otherwise the fail action is performed. The success and fail actions may be any single action or sequence of actions.

6) Assign a value to a variable i : i = value, where value may be any item, e.g. 5, COMP 3(x).

7) Perform some action a on all elements of stack S. Unlike Williams' description of this action stack S is considered to be unchanged on completion of the action: $\forall S:(x \uparrow S \; ; \; a)$.

   e.g. $\forall$ IFSTACK:(x↑IFSTACK; <u>if</u> x <0 <u>then</u>('SYSTEM LABEL', IFSTACK(ifcntr)+1) ↓ IS)

8) Conditional actions : <u>if</u> Boolean expression <u>then</u> action <u>else</u> action. The '<u>else</u> action' part of the conditional statement is optional and the boolean expression is of the form encountered in an ALGOL 60 IF statement.

9) Perform the action specified by the action number : An. The actions associated with the recognition of particular syntactic structures in a finite state diagram have in general been numbered and this action makes it possible to specify that some defined action sequence should be performed at the current point. The action is in fact merely a means for shortening the specification and for improving the clarity for a reader and the

structure for a compiler writer.

10) Indicate that an error has occurred : En (n an integer).

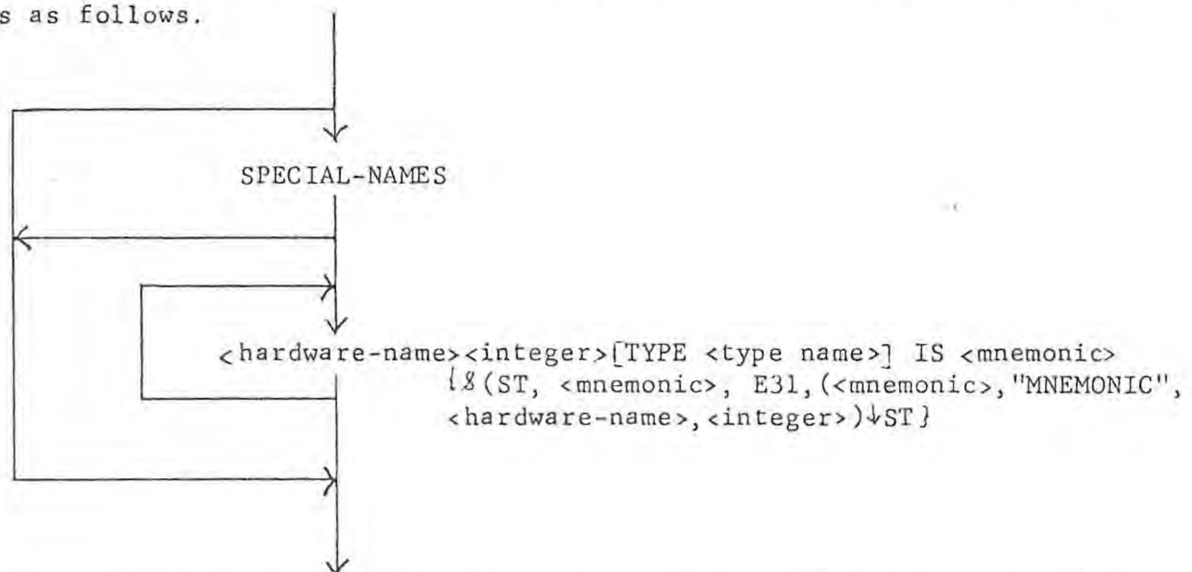
11) The null action : - .

12) Insert a call to a run-time routine into the implementation :

r(p) (r = routine name, p = parameter list).

In using a general notation for specifying a compiler one will wish to avoid the problems associated with system dependence. For example, in the COBOL compiler when reference is made to a particular variable in the PROCEDURE DIVISION one requires at run time the start address and length of the variable to be able to extract its value from some data area. In the 1900 system the approach taken was to set up these addresses in a single character index word in the data area and to pass the address of the location containing this value, as a parameter, to the code generator. In addition, some sections within a compiler are more easily and practically specified by means of a finite state diagram or flowchart. An example of this is the picture clause character string analyzer whose specification in some formal notation would be somewhat ridiculous. For the above reasons this facility has been included and its use can be seen fairly extensively in the COBOL formal specification.

An example of the use of this notation in conjunction with a finite state recognizer is as follows.



The meaning of the action in the above diagram is that on recognition of a special name definition the symbol table (ST) is searched for the mnemonic name to be given to the hardware name and if it is found error 31 is reported

otherwise the 4-tuple described above is inserted into the symbol table.

6.4 THE FORMAL SPECIFICATION OF THE DATA DIVISION

As mentioned previously the DATA DIVISION was specified as a finite state machine with the syntax interpretation rules specified in extended BNF. Since the only pseudo-code generation which takes place during the analysis of this division is that involved with VALUE clauses occurring in the WORKING-STORAGE SECTION it was possible to specify almost the entire operation of the syntax analyzer for the DATA DIVISION using the notation described in section 6.3.

The formal specification of the DATA DIVISION can be found in appendix 6 together with the data names options analyzer. The data names options analyzer is that section of the syntax analyzer concerned with the analysis of data names and the clauses used to describe the characteristics of the fields.

6.5 IMPLEMENTATION OF A PARSER FOR THE DATA DIVISION

The syntax analyzer for the DATA DIVISION has been written in PLAN from the formal specification and operates in the following way. The available storage is assumed to begin at location 45. Thus addresses are allocated starting from this point for the defined data areas but no storage is actually defined by means of a define store pseudo-code until the end of the DATA DIVISION is encountered. At this point one define store pseudo-code for all the required storage as defined in the DATA DIVISION is generated for the code generator.

The system uses a different file to contain the pseudo-codes produced by the PROCEDURE DIVISION analyzer and it is to this file that pseudo-codes for the creation of data area initial values as defined by value clauses will be written. The entire system is described in greater detail in section 7.8.

During the analysis of the DATA DIVISION a number of problems were encountered. The more interesting of these were :

1) The representation of data names which occur more than once in the DATA DIVISION and the problems associated with the handling of such data

items as qualified data names. This presents a problem since a data name appearing more than once may be qualified by any enclosing group or record identifier. This problem was solved by giving the symbol table entries for data items with more than one occurrence a data item type indicating this situation, i.e. that a qualifier is required, and including a link field from the symbol table entry for the enclosing group or record name to the data item entry concerned. This makes it possible to access the data item from its qualifier data item entry.

The standard COBOL rules concerning repeated names within a program have been imposed, viz. a data name may not appear more than once in a single record description, and, within a program the names of the following must be unique : files, mnemonics, records, elementary records and paragraphs.

2) The handling of multidimensional arrays where some field within the scope contains a synchronized clause which causes the mapping function for array element access to become complex. This problem is only particular to word machines and has been written up as a paper by Williams and Bulmer[26].

6.6 SYMBOL TABLE DATA STRUCTURES

The symbol table is built up chiefly during the analysis of the DATA DIVISION for use during the PROCEDURE DIVISION analysis when entries for paragraph names are added to the table when encountered or referenced.

Within the symbol table there are 7 different entry types.

1) Records - non-elementary data items with a level number of 01.
2) Elementary records - elementary data items with a level number of 01.
3) Unique variables - elementary or non-elementary data items which occur only once and which have a level number other than 01.
4) Files - as defined in the INPUT-OUTPUT SECTION by means of a select clause.
5) Mnemonics - as defined in the SPECIAL-NAMES paragraph.

6) Non-Unique variables - elementary or non-elementary data items with a level number other than 01 with a non-unique data name.

7) Paragraphs - as declared and referenced in the PROCEDURE DIVISION.

In addition to these entries there are also file description information lists which are built up from the information contained within a file description for each defined file. This information may be regarded as part of the symbol table entry for a file although it is not physically adjacent to the entry for a file.

Entries have been inserted into the symbol table in such a way that all entries begin on word boundaries. In the cases of the entries for records, variables and elementary records the attribute components also begin on word boundaries.

The seven types of symbol table entries mentioned above all have the same types of attributes for their first four components. These are :

1) A two character link field to the next symbol table entry which generates the same pointer table address for that particular identifier name. This link field is used in the symbol table lookup routine and is generated in the following way. When it is required to locate a particular data item in the symbol table a hashing function is applied to the data name to produce a value which is used as an index to a pointer table.

   If the indexed pointer table value is zero then the data item being sought does not occur in the symbol table, however, if it is non-zero then this value is the address of the first symbol table entry which generated the address in the pointer table. The data item name must now be compared with that in the symbol table entry. If the first two character positions of the address pointed to by the pointer table entry in the symbol table are non-zero then this value is a pointer to the next symbol table entry which generates the same value from the hashing function. In this way all the data items in the symbol table whose names generate the same value from the hashing function are chained together.

The hashing function used for symbol table lookup is :

HASH ADDRESS = if LENGTH = 1 then LETTER(1)

else (LENGTH*LETTER(2)+LETTER(1))MOD256

where LENGTH is the length of the data item in characters

LETTER(i) is the internal numerical value of the ith letter in the data item name.

2) The length of the data item in characters.

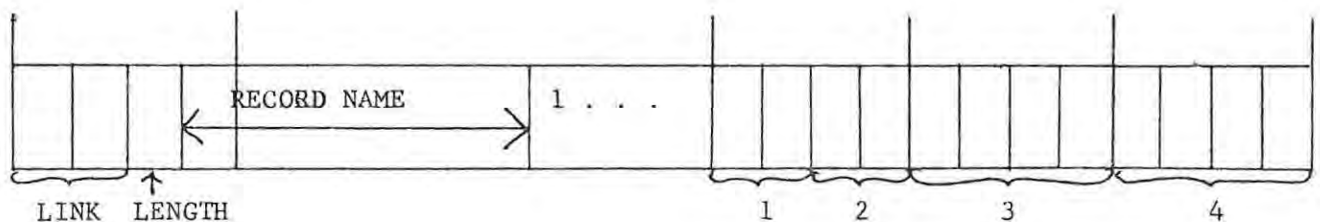
3) The data name.

4) A single character containing the entry type code.

6.6.1 RECORDS

Symbol table entries for records are made on encountering a record name in the FILE SECTION or WORKING-STORAGE SECTION. As mentioned before record names must be unique within a program.

The record name symbol table entry has the format below.



The final component, containing the four numbered fields, of the record name entry begins on a word boundary. The fields contain the following values.

1) A pointer to the file name entry in the symbol table with which the record is associated. In the case of working-storage records this entry is zero.

2) A pointer to the subordinate field. See section 6.5.

3) The record length in characters.

4) The absolute character start position of the record within the final object program.

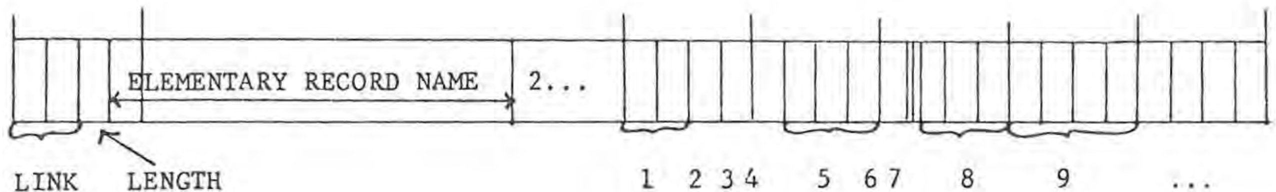
6.6.2 ELEMENTARY RECORDS

Elementary records are defined as data items with a level number of 01 which contain a picture clause.

e.g. 01 ELEMENTARY-RECORD PIC X(120).

Like records symbol table entries for elementary records are made on encountering an elementary record in the FILE SECTION or WORKING-STORAGE SECTION. Elementary record names must be unique within a program.

The elementary record symbol table entry has the format below.



As indicated in the diagram the first field of the final component of the elementary record symbol table entry begins on a word boundary. The numbered fields contain the attributes described below.

1) A pointer to the file name symbol table entry with which the elementary record is associated, if the entry is made from a declaration in the FILE SECTION, or zero if the entry is made from a declaration in the WORKING-STORAGE SECTION.

2) The field type as defined by the picture clause. Possible type codes and types which may be defined are as follows.

TYPE CODE	DATA TYPE
0	Unsigned numeric decimal
1	Signed numeric decimal
2	Unsigned binary
3	Signed binary
4	Edited numeric
5	Alphanumeric
6	Group item (i.e. alphanumeric)

TYPE CODE	DATA TYPE
7	Alphabetic
8	Record (used as a flag for the MOVE statement implementation)

3) A justified indicator indicating the presence or absence of a justified clause when the data type is alphanumeric or alphabetic.

4) The number of subfields within the data item as defined by the picture clause. V and S are not included in the count of the number of subfields.

   For example, £(2)9,9(3)V.99 has 6 subfields

   X(5)BX(5) has 3 subfields

   S9V9 has 2 subfields

5) The length of the data item in characters.

6) The amount of right shifting which must take place before the value in the field is used. Field values are considered to be integers which, when used for arithmetic purposes, must be shifted relative to an assumed decimal point at the right hand end of the field. Left shifting is considered to be negative right shifting.

   For example,

   99PP has an associated shift of -2, i.e. shift the value in the field left two decimal places before use.

   PP99 has a shift of 4, i.e. shift the value in the field right 4 places relative to the assumed decimal point before use.

   S9(3)V9(3) has a shift value of 3.

7) A two bit field indicating the presence or absence of a USAGE IS COMPUTATIONAL clause.

8) The absolute character start position of the elementary record within the final object program.

9) 10 etc. The subfield attributes. There will be one entry in this list for each subfield of the field as defined in the picture clause which adds to the length of the field, i.e. S and V are

excluded. The subfields are represented as character index words in the following way.

number of occurrences/type of subfield

For example, the picture clause character string £(2)9,9(3)V.99 has 6 subfields represented as shown below.

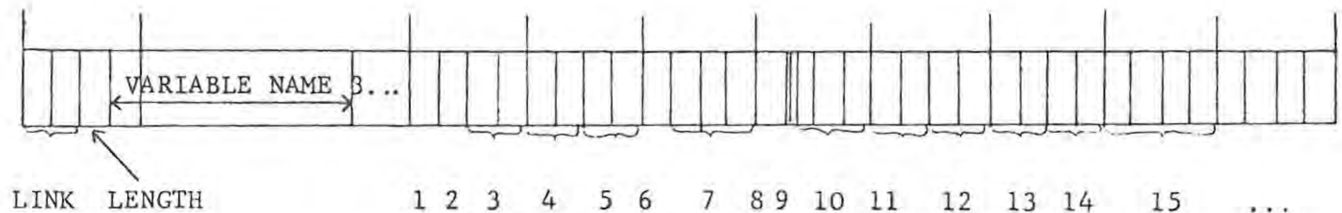
2/£ 1/9 1/, 3/9 1/. 2/9

The character string X(120) has only 1 subfield, viz. 120/X

6.6.3 UNIQUE VARIABLES

Type 3 variables are those which have unique occurrences within the DATA DIVISION. Non-unique variables, i.e. type 6 entries, have an identical symbol table entry format. Variables may be defined in the FILE SECTION or the WORKING-STORAGE SECTION.

The variable symbol table entry has the format below.



The numbered fields contain the following attributes.

1) The field type as defined by the picture clause. See section 6.6.2.

2) The amount of right shift, if the item is an elementary data item.

3) COBOL arrays may have up to 3 dimensions. This field gives the maximum value which may be assumed by the first subscript.

4) The maximum value of the second subscript.

5) The maximum value of the third subscript. If a field is not within the scope of an occurs clause or does not contain an occurs clause itself then the attributes 3, 4 and 5 will all be zero.

6) The number of subfields appearing within the picture clause character

string. In the case of group fields which do not contain picture clauses this attribute has the value zero and the field is assumed to be alphanumeric with the length as specified by attribute 7.

7) The length of the data item in characters.

8) A justified indicator indicating the presence or absence of a justified clause in the data item description.

9) A two bit usage clause indicator.

10) The absolute character start position of the field within the final object program.

11) If a data item has in its description an occurs clause or is within the scope of one or more occurs clauses then when a particular item within the array is accessed it will be required to calculate the address of the item. The address will be calculated from the start address of the field and the values of the subscripts. To assist in this calculation this field and the following two fields have been included and are used to contain the interval or number of characters between repetitions of occurrences of the field for each of the three subscripts. This field gives the interval of repetition associated with the first subscript.

For example,

```
01  A.
    02  B  OCCURS  3.
        03  C  OCCURS  5.
            04  D  PIC  X(3).
            04  E  PIC  X(5).
            04  F  PIC  X(5)  OCCURS  3.
```

will cause the following field values to appear in the symbol table entry for the elementary data item F.

Field 3 - 3	first subscript maximum value.
Field 4 - 5	second subscript maximum value.
Field 5 - 3	third subscript maximum value.
Field 11 - 115	interval of repetition for subscript 1.
Field 12 - 23	interval of repetition for subscript 2.
Field 13 - 5	interval of repetition for subscript 3.

12) The interval of repetition associated with the second subscript.

13) The interval of repetition associated with the third subscript.

14) A link to the subordinate data item symbol table entry.

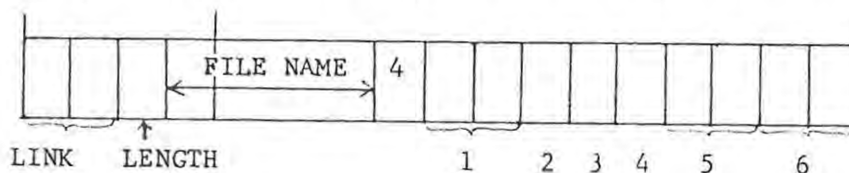
15) 16, etc. The subfield attributes for data items whose data descriptions contain picture clauses.

Group items, i.e. items which do not have a picture clause of their own, are considered to be alphanumeric fields with a length equal to the sum of the lengths of all the data items following the data item under consideration until the next data item with a level number less than or equal to that of the particular group item concerned is encountered. These data items will not have any subfield attributes appended to the end of the group item symbol table entry.

6.6.4 FILES

A symbol table entry for a file name is made on encountering a file name when it is defined within the FILE-CONTROL paragraph. File names must be unique within any one COBOL program.

The symbol table file entry format is as given below.



The numbered fields contain the following attributes.

1) A link to the file description information list which is inserted into the symbol table after the analysis of the FD information. See section 6.6.8.

2) The hardware code for the device to which the file has been assigned. The possible devices and hardware codes are as follows.

DEVICE	CODE
PRINTER	11
CARD-READER	12
EDS	61

3) The channel number associated with the device for the file.

4) The absolute location of the file status word in the final object program. This file status word is initially set to zero indicating that the file is closed at the beginning of the execution of the object program. A negative value indicates that the file has been opened for output and a positive value that the file has been opened for input.

5) This attribute and the next one are only used in the case of EDS files and both contain address table element numbers associated with areas of storage allocated on completion of the analysis of the DATA DIVISION. The address table element number contained within this field is a pointer to a three word parameter block which will be used in EDS read and write operations. The entries in the parameter block contain the following values.

    1) The amount of free space or unread area remaining within the disc buffer.

    2) A pointer to the current position within the buffer.

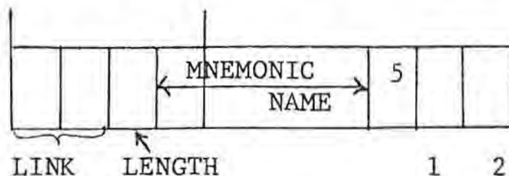
    3) The number of the current bucket in the disc buffer.

6) The address table element number associated with the start address of a disc file bucket buffer allocated at the same time as the parameter block for the file. The size of the allocated buffer is taken from the BLOCK CONTAINS clause. It is necessary to allocate a buffer in this manner and to use it to contain the data for disc file read and write operations since, in general, it may be possible to write more than one record to any particular bucket and thus to optimize on disc file usage the records are packed into the buckets. However, it should be noted that records are never split across buckets. Likewise in the case of reading from a disc file it is possible to read more than one record from a bucket.

6.6.5 MNEMONICS

A symbol table entry for a mnemonic name is made when a mnemonic is defined in the SPECIAL-NAMES paragraph. Mnemonic names must be unique within a program.

The symbol table entry has the format below.



The two numbered attributes contain the values described below.

1) The device code associated with the mnemonic name. Possible devices and their codes are as follows.

DEVICE	CODE	
PRINTER	11	
CARD-READER	12	
*DATE	0	
CHANNEL-n	n	(where n is an integer in the range 1 to 7 and refers to the corresponding tape channel)

2) In the case of card-readers and printers the device name is followed by an integral channel number which is stored in this field.

6.6.6 NON-UNIQUE VARIABLES

Symbol table entry type 6 is associated with a variable which is non-unique within the program. With the exception of the entry type field the entry is identical to that for a unique data name described in section 6.6.3.

6.6.7 PARAGRAPHS

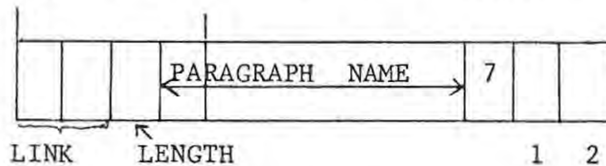
Entries for paragraph names are made in the symbol table when a paragraph name is defined in the PROCEDURE DIVISION and when paragraph names are referenced in PERFORM or GO TO statements. When a paragraph name is defined the symbol table is searched for the paragraph name. If it is found in the symbol table and is flagged as defined then an error is reported as the paragraph name has been defined twice. Similarly, if the entry is not of type 'paragraph' an

error is reported as paragraph names are required to be unique within a program.

If a symbol table entry for the paragraph name is flagged as undefined then the flag is changed to defined. If the paragraph name is not found in the symbol table then an entry is made with the flag set to defined. In both cases the appropriate pseudo-code is output.

When a paragraph name is referenced in a PERFORM or GO TO statement the symbol table is searched for the paragraph name. If the paragraph name is found it is checked to see that it is of type 'paragraph', flagged as either defined or undefined, and if not an error is reported. If the paragraph name is not found in the symbol table then an entry is made with the paragraph name flagged as undefined.

The symbol table entry for a paragraph name has the format below.



The numbered attributes contain the values described below.

1) The defined/undefined flag. (defined = 0, undefined = 1).

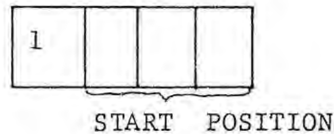
2) An index to the paragraph table within the code generator. See section 7.2.

6.6.8 THE FILE DESCRIPTION INFORMATION LIST

For every file description in the FILE SECTION a file description information list is built up from the clauses describing the file. At the end of the last record description for the particular file concerned the information in the list is inserted into the symbol table and a pointer to the file description information list is inserted into the symbol table entry for the file. See section 6.6.4.

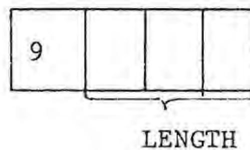
Each item of information in the file description is identified by a type code as described below.

The first entry in the file description information list is as shown below.



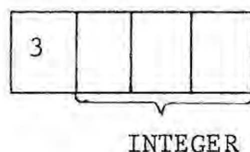
The entry gives the absolute character start address of a buffer which has a size equal to that of the largest record associated with the file.

The final entry in the file description information list gives the length in characters of the longest record associated with the file.



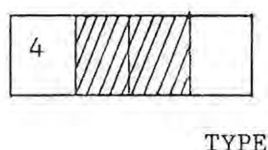
The above two entries will appear in every file description information list, however, the following are optional and depend upon the presence of the file description clauses.

The BLOCK [CONTAINS] <integer> [CHARACTERS] clause causes the entry below to be made in the list. The integer value in the entry, as in the COBOL clause, gives the block size of the associated file.



The LABEL { RECORDS [ARE] | RECORD [IS] } { STANDARD [WITH GENERATION-NO] | OMITTED }

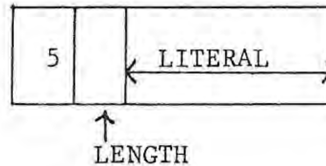
clause produces the entry below:



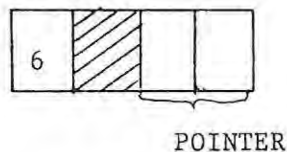
where TYPE may be 0 for OMITTED, 1 for STANDARD or 2 for STANDARD WITH GENERATION-NO.

The VALUE OF $\left\{\begin{array}{l}\text{ID}\\ \text{IDENTIFICATION}\end{array}\right\}$ [IS] $\left\{\begin{array}{l}\text{<data-name>}\\ \text{<literal>}\end{array}\right\}$

clause can cause one of the two following entries to be made in the information list.

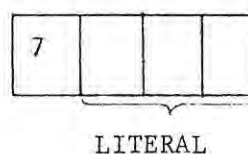


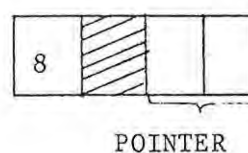
This entry is made when a literal identification appears in the above clause. The length field gives the number of characters appearing within the literal. The entry following this one is aligned on the next word boundary.



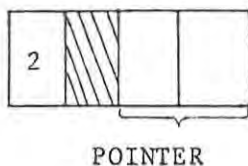
The second possibility, viz. a data name appearing as the identification, causes the above entry to be made. The pointer is to the entry for the data name within the symbol table. This pointer is filled in at the end of the analysis of the DATA DIVISION from the information contained within the data names list. See section 6.7.1.

One of the following two symbol table entries may be made on encountering an ACTIVE TIME clause. The first is made in the case of a literal retention period specification while the latter is made in the case of a data name being used to contain the retention period. Once again the pointer is to the data name entry in the symbol table and is filled in at the end of the DATA DIVISION analysis when the data names list is processed.





The GENERATION-NO IS <data name> clause causes the entry below to be inserted into the information list. The pointer is to the symbol table entry for the data name.



6.7 OTHER DATA DIVISION DATA STRUCTURES

These data structures are used only by that part of the syntax analyzer concerned with the analysis of the DATA DIVISION and are required for the checking of the syntax of the program.

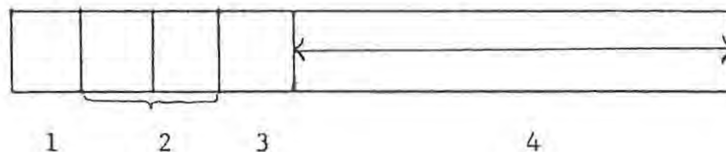
6.7.1 THE DATA NAMES LIST

The list is used to hold the data names which are referenced within the file descriptions in the FILE SECTION. Data names may be referenced within the clauses mentioned below and have the indicated identification values within the entry type fields in the list entries.

CLAUSE		IDENTIFICATION
GENERATION-NO IS	<data name>	2
VALUE OF ID IS	<data name>	6
ACTIVE-TIME IS	<data name>	8

On completion of the syntax analysis of the DATA DIVISION the list is processed to check that the referenced data items have been declared and are of the correct type. The appropriate pointers are then inserted into the corresponding fields in the file description information lists.

The list entries begin on word boundaries and have the format described below.



1) Identification code for the entry type.

2) A pointer to the file name entry in the symbol table in whose file description the data name was referenced.

3) The length of the data name.

4) The data name.

6.7.2 THE FIELD LIST

The purpose of this list is to enable the syntax analyzer to check that no data name is used more than once within a record description. The entries in the list are one word in length and contain pointers to the symbol table entries for the data names declared within a record.

6.7.3 THE GROUP ITEMS LIST

The function of this list is to enable the syntax analyzer to fill in the lengths of group fields when the end of a record is encountered or when a field with an equal or lower level number is encountered in the analysis of a record.

The entries in the list each occupy 1 word and contain the following values.

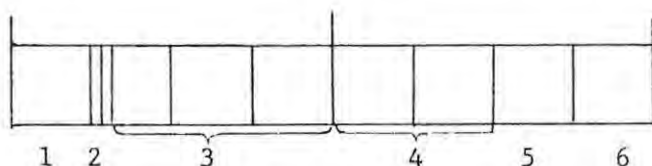
bits 0 - 11	:	a pointer to the attribute component of the group field entry in the symbol table.
bit 12	:	a flag which when unset indicates that there is an enclosing OCCURS clause and that the group field length must not be filled in when the item is deleted from this list. The length is filled in when the group item is deleted from the occurs clause list.
bits 13 - 23	:	the level number of the group item for which the entry has been made.

6.7.4 THE OCCURS CLAUSE LIST

The occurs clause list is used for the handling of data items which contain an occurs clause in their description or are within the scope of some enclosing occurs clause(s). In the case of elementary fields falling into this category it is necessary to retain the fields in some sort of list in order to be able to insert the interval(s) of repetition into the symbol table entries. In the case of group fields it is also required that the field lengths be calculated and inserted into the symbol table entries.

An additional problem is that of handling synchronized fields which occur within the scope of an occurs clause. A solution to this problem is suggested by Williams and Bulmer[26].

Each entry within the list occupies two words. The scope of each occurs clause within the list is preceded by an entry of type 'list'. The entry format is given below along with the functions of the fields.



1) The list entry type

   0 - list

   1 - elementary data item

   2 - group field.

2) If the entry is for a list then this 2 bit field indicates if there are any synchronized fields contained within the scope of the list. If the entry is for an elementary data item it indicates whether the elementary field is synchronized or not. In the case of group fields this field contains the value zero as group fields may not be synchronized.

3) If the entry is for a list then this field gives the number of repetitions of the occurs clause which caused the list entry, otherwise it gives the absolute character start position of the field within the final object program.

4) For a list entry this field contains a pointer to the back of the list in the occurs clause list, otherwise it is a pointer to the data component of the data item in the symbol table. In the case of FILLER fields this field contains the value zero.

5) For a list entry the field contains the level number of the data item which contained the occurs clause causing the list entry. For elementary fields it contains the length of the data item.

6) In the case of elementary or group fields this component contains the level number of the field.

7. THE COBOL COMPILER - PART II

THE PROCEDURE DIVISION AND CODE GENERATOR

7.1 INTRODUCTION

The complete syntax of a number of statements which may be encountered in the PROCEDURE DIVISION has been specified in the extended BNF notation described in section 6.3. The formal specification of these statements can be found in appendix 7.

It will be noticed that the specification includes no mention of data names which have multiple occurrences. For the sake of clarity it was decided to omit these from the specification. Quite obviously the whole concept of multiple data names can be handled by a single routine within the syntax analyzer to locate the correct occurrence of the data name entry when a multiple data name is referenced within any statement.

As can be seen from the formal specification the syntax checking required within the PROCEDURE DIVISION is of a fairly straightforward mechanical nature and thus the first part of this chapter will be devoted to the problems associated with the generation of pseudo-codes for input to the code generator. In this respect the problems associated with the handling of paragraph names and system labels, GO TO, IF and PERFORM statements, imperative statements encountered within ADD, SUBTRACT, MULTIPLY, DIVIDE and READ statements, as well as some general system consideration are described.

The remainder of the chapter is devoted to code generation aspects of the COBOL compiler.

7.2 PARAGRAPHS AND SYSTEM LABELS

As mentioned in section 6.6.7 an entry is made in the symbol table for a paragraph name when it is encountered for the first time. At the time when this symbol table entry is made the paragraph name is allocated a unique index to the paragraph table used by the code generator.

When a paragraph declaration is encountered an appropriate pseudo-code together with the paragraph table index is output to the pseudo-code stream. The corresponding pseudo-code action consists of a call to a defined local routine within the code generator with the index being the parameter to the routine.

The paragraph table in the code generator has been defined in the input parameters to the code generator generator and the local routine to handle paragraph definitions regards this table as containing two pieces of information about every label.

1) A pointer to the most recent occurrence of a reference to the paragraph name. All forward references to a paragraph name are chained together with the address field in the instruction containing a pointer to the previous paragraph reference.
2) The address of the paragraph once it has been declared.

Once the latter field has been filled in, i.e. once the paragraph name has been defined, this value is inserted into the address fields of any instructions which contain references to the paragraph concerned.

On encountering a statement which causes a branch to a paragraph an appropriate pseudo-code together with the correct paragraph table index is output to the pseudo-code stream. Within the corresponding pseudo-code action there is a call to a local routine which appears as the address field of some branch instruction. This local routine takes the code generator paragraph table index as its parameter and returns a value to be inserted into the address field of the branch instruction.

The value returned will be the actual address if the paragraph name has already been defined or, if not, it will be the current value of the field containing the pointer to the most recent occurrence of a reference to that paragraph name. In this latter case the absolute location of the instruction containing the call to the routine will be inserted into the field containing the pointer to the most recent occurrence.

As an example consider the following piece of COBOL source code.

COBOL CODE	ABSOLUTE LOCATION	GENERATED CODE	
GO TO PARAGRAPH-A.	100	BRN	0
.			
.			
.			
GO TO PARAGRAPH-A.	160	BRN	100
.			
.			
.			
PERFORM PARAGRAPH-A.	245	BRN	160
.			
.			
.			
PARAGRAPH-A.	320	-	
.			
.			
.			
GO TO PARAGRAPH-A.	410	BRN	320
.			
.			
.			

At this point the entry in the paragraph table corresponding to PARAGRAPH-A will appear as shown below.

BRANCH FORWARD CHAIN HEAD	ABSOLUTE LOCATION
245	320

From this information it is possible for the loader to fill in the absolute location for all instructions in the branch ahead chain.

System labels are used by the syntax analyzer in the generation of pseudo-codes for imperative and IF statements and their use will be described together with the relevant statement where necessary.

A system label table has been allocated within the code generator in the same way as the paragraph table and is manipulated in an identical fashion. Besides being allocated system labels are not manipulated at all within the syntax analyzer and thus no system label table is required within the syntax analyzer. System labels are allocated as indices to the system label table in the same way as paragraph names and it is these indices which appear within the generated pseudo-codes as system label definitions and branches to system labels.

7.3 IMPERATIVE STATEMENT HANDLING

Of the implemented statements it is only the READ and IF statements which do not fall into the category of imperative statements. An imperative statement, which consists of a sequence of one or more imperative statements, may occur in an ON SIZE ERROR clause, within an ADD, SUBTRACT, MULTIPLY or DIVIDE statement, and must appear in a READ statement as the action to be taken on encountering the end of the data, i.e. within the AT END clause.

Imperative statements may be nested to any depth and are terminated by a full stop. Thus even if there is more than one imperative statement being recognized a full stop marks the end of all of them.

A flag is used to indicate that an imperative statement is currently being recognized. This flag is set when the first imperative statement of a nested or non-nested sequence is recognized and simultaneously a system label is allocated. An appropriate pseudo-code with an associated pseudo-code action specifying that the size error flag must be tested and if found to be unset a branch to the system label just allocated must take place is output.

On encountering a full stop within the source program the imperative statement flag is tested and if found to be set the associated imperative statement system label is defined in the pseudo-code output stream.

As a READ statement may not occur within an imperative statement the same flag is used for the analysis of the imperative statement which must appear in the AT END clause.

7.4 THE IF STATEMENT IMPLEMENTATION

The complete syntax of the IF statement can be found in appendix 7. As COBOL allows nested IF statements which may have another IF statement contained within the code following a THEN this can lead to ambiguities in the statement structure. To avoid this the following rule is imposed.

An ELSE together with its associated statements is considered to belong to the nearest preceding IF statement which does not already have an ELSE associated with it.

For example, the statement :

```
IF A=B THEN IF X=Y THEN ADD P TO Q ELSE ADD R TO S
  ELSE ADD T TO U
```

causes P to be added to Q if A=B and X=Y

R to be added to S if A=B and X≠Y

T to be added to U if A≠B

The conditions which may appear within an IF statement have been restricted to relational and sign conditions. Relational conditions are restricted to simple relational conditions, i.e. AND and OR are not allowed.

The following rules have been imposed upon relational conditions.

1. If one of the operands is a literal then it is considered to have been moved before comparison to a field of the same class, usage and size as that of the field with which it is compared. The literal is zero or space filled if necessary.
2. If an alphanumeric (or alphabetic) field is compared with an alphanumeric (or alphabetic) field then the contents of the two fields are compared character by character starting at the leftmost. If the two fields are not of the same length then the shorter is considered to be extended to the length of the longer and the extra character positions are considered to contain spaces.
3. If an edited numeric field is being compared it is considered to be an alphanumeric field.

4. Non-elementary fields, excluding records, may be compared and are considered to be alphanumeric.

IF statements are analyzed by the syntax analyzer with the aid of a variable used to contain the current level of nesting of IF statements and a stack which may be regarded as having two components per entry and one entry for each IF statement currently being analyzed. The components of the stack contain a flag indicating whether the code following a THEN or an ELSE is being recognized and a system label number which will be branched to and inserted into the object program at the correct points.

In addition to the above a system label is allocated to be used as the end marker for the IF statement and its definition is inserted into the pseudo-code list on encountering the full stop marking the end of the IF statement or, in the case of nested IF statements, the end of the outermost of the nested IF statements.

The maximum level of nesting of IF statements has been limited to 8 which should be more than adequate for the type of user for which the compiler has been designed.

7.5 THE PERFORM STATEMENT IMPLEMENTATION

A reasonably standard set of rules has been imposed upon the PERFORM statement and its usage. The rules are as follows.

1. If the code within the range of a PERFORM statement contains another PERFORM statement then the range of the nested PERFORM statement must be either
   (a) contained entirely within the range of the enclosing PERFORM statement, or
   (b) totally removed from the range of the enclosing PERFORM statement.

2. The range of one PERFORM statement may overlap with the range of another PERFORM statement provided that the one PERFORM statement is not contained within the range of the other.

For example, the following code sequence is illegal.

```
PARAGRAPH-A.
          .
          .
          .
     PERFORM  PARAGRAPH-B THRU PARAGRAPH-D.
          .
          .
PARAGRAPH-B.
          .
          .
          .
PARAGRAPH-C.
          .
          .
          .
PARAGRAPH-D.
          .
          .
          .
PARAGRAPH-E.
          .
          .
          .
     PERFORM  PARAGRAPH-A  THRU  PARAGRAPH-C.
```

3. No two PERFORM statements may terminate on the same paragraph.

Since the syntax analyzer makes only one pass of the source program while simultaneously generating pseudo-codes the situation regarding PERFORM statements must perforce be somewhat artificial for two main reasons.

1) If one wishes to have a PERFORM statement of the form PERFORM <paragraph-name-1> THRU <paragraph-name-2> , where the paragraph names have not yet been declared, it is not possible to check the legality of the PERFORM statement until both paragraph names have been declared. This problem can be solved reasonably easily within the syntax analyzer by making use of two lists. These lists are:

   a) A paragraph names list containing the paragraph names in the order in which they occur within the PROCEDURE DIVISION.

b) A list containing the PERFORM statement range, i.e. the start and end paragraph names.

These two lists are used to check that the paragraph names are declared and that <paragraph-name-2> occurs after <paragraph-name-1> .

2) A far more serious problem is that of the insertion of pseudo-codes into the pseudo-code stream which will result in a return to the correct point on completion of the execution of a section of code under the control of a PERFORM statement.

Initially it was thought that this latter problem could be solved by having a stack for return addresses. On encountering a PERFORM statement within the object program it would be necessary to add the return address to this stack. A list of all the paragraph names following the last paragraph in the range of each PERFORM statement would have to be passed to the code generator and set up in an appropriate table. Whenever a paragraph definition occurred it would be necessary to search the perform paragraph list and if the paragraph concerned appeared within the list the code generator would be required to insert code of the form :

IF return address stack index > 1 THEN subtract 1 from the return address stack index and go to the address in the stack entry for index+1.

However, it will be noticed that the PERFORM statement rules 1a and 2 can be violated with the above arrangement and a branch to the wrong address can take place.

There are two solutions to this problem, viz. to make rules 1a and 2 illegal cases, which would not be a serious limitation in a COBOL compiler intended for teaching purposes. The second solution is to allow rules 1a and 2 to stand and to implement the PERFORM statement in the rather more complicated way described below and formally in appendix 7.

This solution requires the use of the lists mentioned in the solution to the first problem. Each PERFORM statement is allocated a number which corresponds

to the index to the perform statement range list entry for the particular PERFORM statement.

From this list and from the list of paragraph names, in the order in which they occur within the PROCEDURE DIVISION, a list of all the paragraph names which follow the last paragraph in the range of each PERFORM statement is constructed. In the case of a PERFORM statement having a range which extends to the last paragraph in the program it will be necessary to insert the pseudo-code for the definition of a dummy paragraph name at the end of the program.

Thus for each PERFORM statement there now exists a paragraph name before which code must be inserted to check whether the preceding code was executed by virtue of a PERFORM statement or not. If the code was executed under control of a PERFORM statement then a branch to the statement following the PERFORM statement must take place. In the case of a PERFORM statement which has the TIMES option included the branch must take place to an appropriate point before the statement following the PERFORM where it must be checked to see if the PERFORM statement range has been executed the correct number of times.

Within the code generator there will have to be an area of storage set aside for the list of paragraph table indices before which code must be inserted to check for the end of a PERFORM statement. For each paragraph name in the corresponding list in the syntax analyzer the appropriate index will be passed to the code generator by means of a pseudo-code with the index as a parameter. The corresponding pseudo-code action will initiate a call to a routine within the code generator which will insert the parameter into the defined storage area. This will have to be carried out before any of the PROCEDURE DIVISION pseudo-codes are interpreted. Because of the structure of the syntax analyzer it is possible to insert these pseudo-codes before those generated from the PROCEDURE DIVISION statements. See section 7.8.

Within the generated object program there will have to be an area of storage which conceptually contains two entries for each PERFORM statement within the program. These entries will be as follows.

1. The return address to be branched to on completion of the code executed under control of the PERFORM statement.
2. A flag indicating whether the preceding code has been executed under control of the PERFORM statement.

Once again values will be inserted into this table in such a way as to reflect the PERFORM statement numbering. In practice the above two entries will be combined into one with the return address being non-zero to indicate that the PERFORM statement is operational and zero to indicate that the code was not executed under control of a PERFORM statement.

At the appropriate point within the syntax analysis of a PERFORM statement a pseudo-code will be generated to cause the following code to be inserted into the object program:

```
LDN     1     return address
LDN     2     perform statement table index
STO     1     perform statement return address table (2)
BRN           first statement in perform range
```

where return address would be the location of the statement following the unconditional branch statement. The code may not be exactly the same as above since literals are limited to 12 bits (i.e. 4095) on the 1900 system. The return address would be calculated by means of a call to the appropriate routine within the code generator which would utilize values from within the instruction buffer parameter block.

On encountering the pseudo-code used for the definition of a paragraph name it would be necessary, during the code generation, to check if the paragraph name occurred within the perform paragraph names list. If the paragraph name did occur it would be necessary to generate code such as the following.

```
LDN     1     perform statement index
LDX     2     perform statement return address table (1)
BZE     2     *+3
STOZ          perform statement return address table (1)
EXIT    2     0
```

An example of this is as follows.

Assume that the start address of the return address list in the generated object program is 200.

COBOL CODE	LOCATION	GENERATED CODE		
PARAGRAPH-A.	400	-		
.				
.				
.				
PERFORM PARAGRAPH-B.	450	LDN	1	454
	451	LDN	2	1
	452	STO	1	200(1)
	453	BRN		600
PARAGRAPH-B.	600	-		
	675	LDN	1	1
	676	LDX	2	200(1)
	677	BZE	2	680
	678	STOZ		200(1)
	679	EXIT	2	0
PARAGRAPH-C.	680	-		
PERFORM PARAGRAPH-E.	740	LDN	1	744
	741	LDN	2	2
	742	STO	1	200(2)
	743	BRN		850
	785	LDN	1	3
	786	LDX	2	200(1)
	787	BZE	2	790
	788	STOZ		200(1)
	789	EXIT	2	0
PARAGRAPH-D.	790	-		

PERFORM PARAGRAPH-A THRU PARAGRAPH-C	800	LDN	1	804
	801	LDN	2	3
	802	STO	1	200(2)
	803	BRN		400
	845	LDN	1	4
	846	LDX	2	200(1)
	847	BZE	2	850
	848	STOZ		200(1)
	849	EXIT	2	0
PARAGRAPH-E.	850	-		
	885	LDN	1	2
	886	LDX	2	200(1)
	887	BZE	2	890
	888	STOZ		200(1)
	889	EXIT	2	0
PARAGRAPH-E.	890	-		
PERFORM PARAGRAPH-D.	940	LDN	1	944
	941	LDN	2	4
	942	STO	1	200(2)
	943	BRN		790

The state of the PERFORM lists as used by the syntax analyzer are depicted below.

PERFORM RANGE LIST

START PARAGRAPH	END PARAGRAPH
PARAGRAPH-B	PARAGRAPH-B
PARAGRAPH-E	PARAGRAPH-E
PARAGRAPH-A	PARAGRAPH-C
PARAGRAPH-D	PARAGRAPH-D

FOLLOWING PARAGRAPH LIST
PARAGRAPH-C
PARAGRAPH-F
PARAGRAPH-D
PARAGRAPH-E

Assuming that PARAGRAPH-E is being executed under control of the fourth PERFORM statement (i.e. under control of the second PERFORM under control of the third PERFORM under control of the fourth PERFORM) then the return address list in the object program will be in the state given below.

RETURN ADDRESS LIST
0
744
804
944

7.6 OTHER IMPLEMENTATION FEATURES

In addition to the already mentioned implementation features there are several others worth mentioning.

1) The representation of signed decimal numeric fields.

Fields which are of type signed decimal numeric have the sign stored in the most significant bit of the most significant character position of the field. Thus if the field contains a negative value then the numbers 0 to 9 in the leftmost character position of the field are represented as @ to I in the ICL internal code format.

2) Literals occurring within the PROCEDURE DIVISION.

When a literal is encountered within the PROCEDURE DIVISION in a statement such as

```
ADD -100.25 TO A
PERFORM PARAGRAPH-1 THRU PARAGRAPH-3 10 TIMES
```

the literal is inserted into the data area and appears in exactly the same form as an identifier of the appropriate type. In addition to the literal appearing within the data area a character index word containing the length and the absolute start position of the literal within the data area is also inserted and it is the address table element number associated with this word which is passed across to the appropriate pseudo-code actions as a parameter.

In this way literals occurring within the COBOL source program are not distinguished from identifiers and can be handled in exactly the same way by the same pseudo-code actions concerned with the manipulation of identifiers.

3) The conversion of decimal numbers to binary.

Because the compiler has been designed for teaching purposes it was decided to limit the range of numbers which may be used in arithmetic operations. Numbers which appear in arithmetic operations are limited to a maximum of 13 digits. This is the maximum number of digits which may be contained within 2 words (i.e. 46 bits) and should be sufficient for student usage.

The code skeletons to convert signed or unsigned decimal values containing fewer than 14 digits to double length binary numbers can be found in appendix 8. The parameters for the code skeletons are as follows.

1. The first of 2 adjacent accumulators into which the binary number is to be inserted.
2. The address table element number associated with the location containing the length and start address of the field whose contents are to be converted to binary.

4) The conversion of binary numbers to decimal.

This problem is a far more difficult one than the previous one because of the enormous number of possibilities which may arise in the conversion process.

Assume that the binary field produced as the result of some arithmetic operation contains a number which conceptually has i places to the left of the decimal point and j places to the right of the decimal point, i.e. the field contains the binary equivalent of the number as described by the COBOL picture clause 9(i)V9(j). The field into which this value is to be inserted may be assumed to have a picture clause description of the form 9(k)V9(1).

The code required to convert a binary number to a decimal number has been broken down into 9 separate code skeletons. One of these code skeletons is always inserted into the binary program. This is the one to load the contents of the location containing the length and start address of the result field. Depending upon the relative values of i, j, k and 1 various of the remaining 8 code skeletons are inserted together with their appropriate parameters. The binary result is expected to be in accumulators 6 and 7.

The code skeletons together with the conversion algorithm can be found in appendix 9.

7.7 THE IMPLEMENTED SYNTAX ANALYZER

The implemented part of the syntax analyzer consists of that part concerned with the IDENTIFICATION, ENVIRONMENT and DATA divisions and a small part of the PROCEDURE DIVISION analyzer. However, the formal specification of several of the PROCEDURE DIVISION statements has been completed and it should not be too large a task to complete this section of the syntax analyzer. Furthermore, the code skeletons corresponding to the pseudo-codes to be inserted into the output stream for the not yet implemented statements have been specified and it is only required that these code skeletons be added to the pseudo-code action table used by the code generator when the specified statements are implemented. Thus there are no changes to be made to the code generator in order to include any further implementation of the syntax analyzer.

The only possible changes to the code generator are in the area of the local routines. It is a simple operation to generate a new code generator with

an enhanced set of local routines.

Listings of the implemented parts of the syntax analyzer can be found in appendix 10.20.

7.8 THE SYNTAX ANALYZER - CODE GENERATOR INTERFACE

The code generator has been generated in such a way as to expect its input from a particular disc file. This disc file contains the pseudo-code stream which was output from the syntax analyzer. The syntax analyzer has been written in such a way that all storage areas, constants, etc. are defined in terms of pseudo-codes on the output file before any code generating pseudo-codes appear. As it may be necessary to define data areas and constants on recognition of certain syntactic structures in the PROCEDURE DIVISION after other code generating pseudo-codes have been output the following has been done.

Two files are used to contain the pseudo-codes output from the syntax analyzer. The first one is used to contain all the data area and constant definitions while the second one is used to contain the pseudo-codes as generated from the COBOL statements in the PROCEDURE DIVISION.

On completion of the syntax analysis it is necessary to concatenate these two files after emptying the appropriate buffers. This is done by simply reading the instruction pseudo-code file and appending the information contained within this file to the file containing the data area definitions.

As certain standard routines may be required within the binary programs generated from some COBOL source programs the above scheme contains an additional feature. The standard routines would be routines to perform functions such as : the data manipulation required in moving a numeric field to an edited numeric field, sorting routines, etc. Quite obviously it would be ridiculous to include the code for features such as the above at each point where it is required in the binary program.

The proposed system is to include all the required routines within the pseudo-code action table and thus each one would be associated with some pseudo-

code. Within the syntax analyzer, while the analysis of the PROCEDURE DIVISION is being carried out, it is possible to detect which of these routines will be required within the binary program to be generated. A table of these routines would be kept and those required appropriately flagged during the syntax analysis.

On completion of the syntax analysis this table would be scanned and all the routines flagged as required could then be included in the generated binary program by means of the inclusion of the appropriate pseudo-codes in the output stream.

The situation is not quite as simple as this in that each pseudo-code generated in the manner described above would have to be preceded by a definition of the current address together with a predefined address table element. It would be by means of this address table element that the routine would be called from the code which generated the call(s).

The table of routines within the syntax analyzer would require three components.

1) A used/unused flag.
2) A pseudo-code action number.
3) An address table element number to contain the start address of the routine.

The above information and that required to be set up by the code generator for the PERFORM statement and paragraph handling must be inserted into the output stream before the start of the rest of the pseudo-codes generated from the COBOL source program PROCEDURE DIVISION.

With the system set up as described this is very easily accomplished by adding these actions and pseudo-codes to the file containing the data area definitions before appending the rest of the pseudo-codes.

The complete operation of the compiler can be described as follows.

The compiler consists of a controlling segment which is given control on entry. The controlling segment first calls an initialization routine and

then individually calls routines for the analysis of the four divisions of the COBOL source program. On completion of the analysis of the DATA DIVISION a storage definition for all the storage defined is output to the file containing the data area definitions.

At this point the data names list (see section 6.7.1) is checked to ensure that all the data names used in the various file descriptions have been declared and the appropriate pointers are inserted into the file description information lists (see section 6.6.8) within the symbol table. All declared files which are assigned as EDS files are allocated a disc buffer of the size specified in the file description list entry for the BLOCK CONTAINS clause. The address table element number associated with the disc file buffer is inserted into the symbol table entry for the disc file.

The controlling segment now calls that part of the syntax analyzer concerned with the analysis of the PROCEDURE DIVISION which generates the appropriate pseudo-codes and data area definitions.

On completion of the PROCEDURE DIVISION analysis it is checked to see whether there were any PERFORM statements used within the program and if so the legality of these statements is checked and the operations described in section 7.5 are performed.

An additional check which is carried out is that all the paragraph names occurring within the symbol table have been flagged as defined.

At this point the syntax of the COBOL source program has been checked and if any errors have been detected the compiler will halt with an appropriate message indicating that the program is syntactically invalid. If no errors have been detected control is passed to a routine to scan the list of COBOL run-time routines to check which of these routines, if any, are to be included within the binary program to be generated. If there are any required run-time routines the appropriate start address table element definitions and pseudo-codes are added at the correct point to the list of pseudo-codes to be processed by the code generator.

The pseudo-code sequence in the file containing the code generating pseudo-codes is then appended to that in the file containing the data area definitions. The COBOL code generator has been inserted in a loadable form into a particular disc file and is now loaded into store. As the syntax analyzer and system tables currently in store are no longer required they are overwritten by the code generator which is loaded at the bottom of store.

Control is now passed to the code generator which uses the pseudo-code stream and the optimized pseudo-code action table and index as input and produces a binary program in loadable form in the output file defined when the code generator was created. Also output to this disc file will be the paragraph names and system labels forward branch tables.

The binary program can then be loaded into store by means of an appropriate loader which will be required to resolve forward branches to paragraph names and system labels.

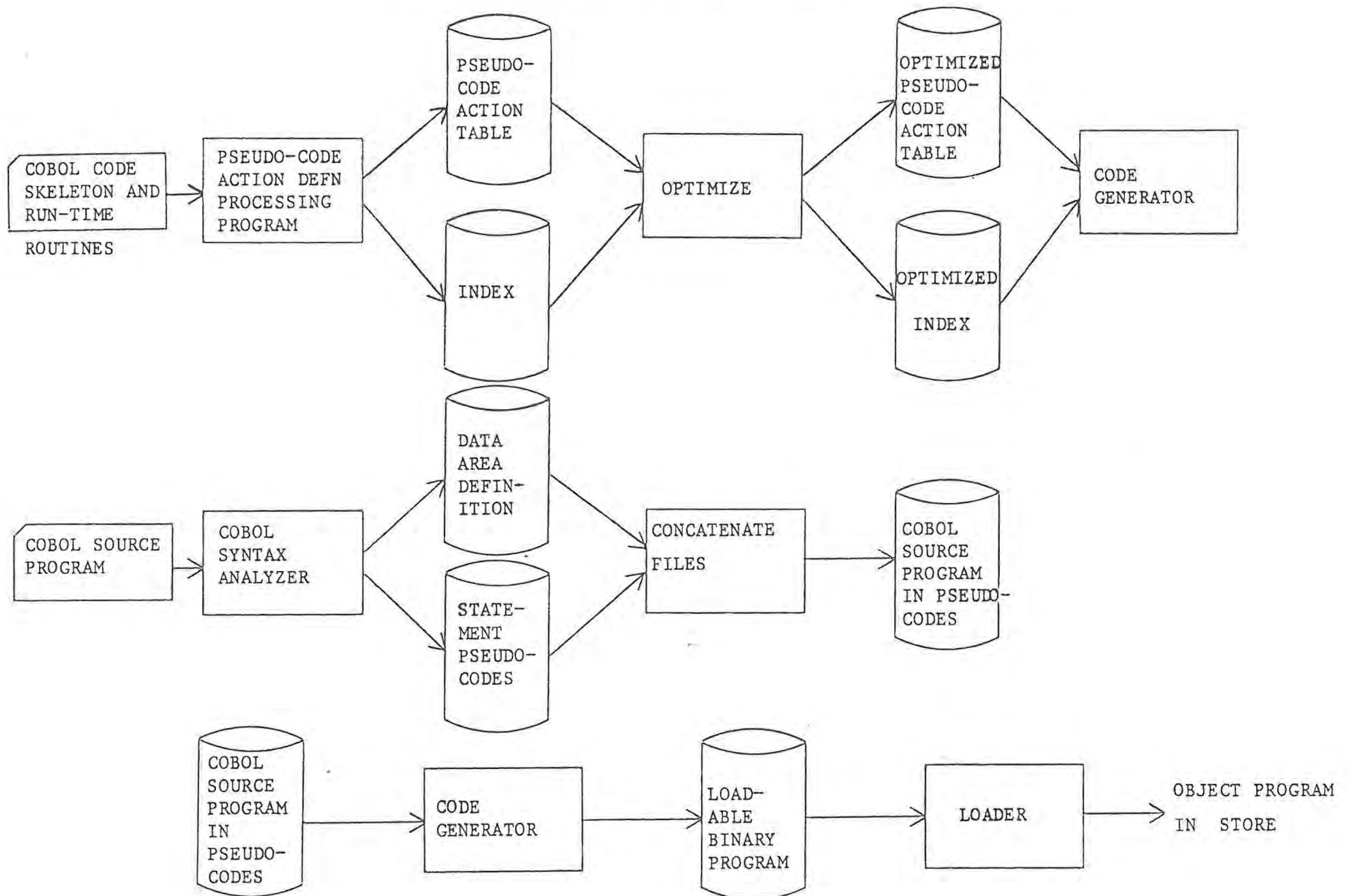
The structure of the COBOL compiler is given in the diagram overleaf. It will be noticed that the code generator makes use of an optimized pseudo-code action table (See section 4.4). The code skeletons used by the code generator for the implemented parts of the syntax analyzer can be found in appendix 10.23.

7.9 THE COBOL CODE GENERATOR

Two different COBOL code generators have been generated both of which produce identical output from the same input. The first COBOL code generator produced by the code generator generator was defined in terms of the locally optimized code skeletons which were used in the generation of the code generator generator by the bootstrap program. The data used for the generation of this code generator can be found in appendix 10.1, together with the decompiled version of the code generator produced in appendix 10.22. The code skeletons used appear in appendix 10.10.

In the case of the COBOL code generator a somewhat modified define address constant routine has been employed. The reason for this is that the 1900 system, being word oriented, allows for the individual addressing of

THE STRUCTURE OF THE COBOL COMPILER



characters within a word. In addition a count field may also be included within the character index word. To allow for this the definition of an address constant has been defined as follows. The list is to consist of character representations of the character index words in the form below.

COUNT/ADDRESS TABLE ELEMENT NUMBER.CHARACTER NUMBER

The length of the list gives the number of words in the list which will not be the same as the number of defined address constants.

The action is to insert the character number into bits 0 and 1, the count into bits 2 to 9 and the address from the specified address table element into bits 10 to 23. If there is more than one address constant definition in the list it is assumed to immediately follow the preceding definition.

Only those facilities necessary to interpret the pseudo-code actions have been included. Thus the COBOL code generator does not include the register allocation system, the forward and backward branching systems and the relocation system. A few of the local routines have been included within the code generator and as more of the syntax analyzer is implemented several more local routines will have to be included. This sort of change is the only one which is envisaged as being required by any future effort in the further implementation or extension of the COBOL compiler with respect to the code generator.

The second COBOL code generator produced had the action analyzing routines defined in terms of the general pseudo-codes. The production of this code generator was performed merely as an exercise to show that it is possible to define a code generator in terms of the general pseudo-codes.

The code generator produces binary programs identical to those produced by the previously described COBOL code generator. A listing of the code generator set up data can be found in appendix 10.12, while appendix 10.13 contains a listing of the code skeletons. Appendix 10.21 contains a decompiled version of this code generator.

7.10 EXTENDING THE COBOL COMPILER

There are four extension aspects of the system which must be considered.

1) Implementing the existing formal specification.

This should be a fairly easy process because of the modular nature of the syntax analyzer. All that is required is that the formal specification be converted to one or more PLAN routines and incorporated into the syntax analyzer with calls to the routines inserted into the control segment of the PROCEDURE DIVISION syntax analyzer at the appropriate points. As the code skeletons have already been specified it is merely required that these be added to the pseudo-code action table.

2) Implementing non-specified statements.

To implement the syntax analysis and code generation for statements for which a formal specification does not yet exist it will be found that this can most easily be done by first writing the formal specification in the notation described in section 6.3 and then proceding as in part 1 above.

In addition the implementor will be required to write the code skeletons corresponding to the pseudo-codes which will be generated on recognition of the various syntactic structures.

3) Extending the pseudo-code action table.

To extend the table requires that the additional code skeletons be added to the list of the already existing ones. If there are any instructions which have been used in these code skeletons which are not already specified in the opcode to internal code mapping then they must be added to the specification. This data is then run through the pseudo-code action definition processing program to produce a new pseudo-code action table. This is followed by a run of the program to optimize the new pseudo-code action table.

4) Adding local routines to the code generator.

The procedure to include another local routine in the code generator is as follows. Write the routine using those facilities which are permitted within a local routine and run this together with a suitable opcode to machine

code mapping through the pseudo-code action definition processing program. Obtain the output by means of the program to print the output files produced. This code is then added to the existing input to the code generator generator in a local routine definition. The most convenient point at which to add this would probably be after the current local routine definitions. The address table element containing the start address of the local routine must then be inserted into the address definition list associated with address table element 30 at the appropriate point. The appropriate point is that position within the list which equals the number by means of which the local routine is referred to within the pseudo-code actions.

This data must then be input to the code generator generator in order to produce a new code generator incorporating the additional local routine(s).

8. CONCLUSION

While this project has not attempted to solve the problem which Aho and Sethi[1] describe under the heading of "Automatic Code generation" and conclude is beyond present capabilities it is felt that the system of code generator generation described in this thesis is a reasonable and practical alternative to the problem of automatic code generation.

The storage and mill time requirements of the various generated systems were as follows. The original bootstrapped code generator generator on the ICL 1900 system occupied 3414 words of storage. To generate the COBOL code generator defined in terms of the locally optimized pseudo-codes required 2 seconds of processor time, while the code generator generator defined in terms of the general pseudo-codes for the ICL system required 7 seconds.

The latter code generator generator took 16 seconds to generate itself and 4 seconds to generate the COBOL code generator. The storage requirement of the COBOL code generator was 2972 words and 4177 words for the general pseudo-code code generator generator.

As the storage requirements for these systems were relatively small it can be seen that the system may be readily transported to most mini-computers currently available without any problems of storage size.

As the COBOL code generator was using an optimized pseudo-code action table the size, in buckets, of the disc file required to hold the table was approximately equal to the number of different pseudo-codes which could be generated from the syntax analyzer.

The code skeletons for the general pseudo-codes for the INTERDATA system were written in two hours. The required local routines have not yet been written but these should take no longer than two to three hours. Thus the effort required to transport the system should be of the order of four or five hours. This does not include the writing of a loader for the new system or the physical process of performing the transportation.

A further advantage of this system is that it is very easy to see what code is being produced and what actions are being taken for each pseudo-code. If it is found that the code generator is not functioning as expected it is a reasonably straightforward process to trace the error, make the appropriate changes to the specification and to create a corrected version.

With respect to the use of extended BNF as a means for formally specifying the syntax of the COBOL compiler the following advantages of this system were recognized.

1) The use of a formal specification of the static semantics enables one to separate the language problems from the programming problems and hence simplifies both tasks.
2) Once a formal specification has been produced the implementation of a syntax analyzer in some language can be achieved in a relatively short space of time.
3) The resulting syntax analyzer has a better chance of being correct if based on a correct formal specification for both syntax and static semantics than if only based on a formal specification of the syntax.
4) If one wishes to implement a similar compiler for a different machine the same formal specification can be used thus considerably reducing the implementation time.
5) If alterations to the design are required these can be made with a minimum of effort by first making the alterations to the formal specification.

If a less systematic method had been used in the implementation of the COBOL syntax analyzer it is likely that the data structures and programs would have required alterations at several points in the development process.

A number of test examples (appendix 10.24) were compiled by the produced COBOL compiler and the ICL COBOL compiler XEKB to test the relative efficiency

of the produced COBOL compiler. The amount of time, in seconds, taken in each case is given below.

	R.U. COBOL compiler	ICL COBOL compiler
Program 1	1	7
Program 2	1	7
Program 3	4	20

From the above it can be seen that the produced COBOL compiler runs considerably faster than XEKB. As more statements are implemented the efficiency of this compiler should not decrease significantly and it is felt that the efficiency of the compiler can be increased by the rewriting of some of the modules.

REFERENCES

1) Aho A.V. and Sethi R. "How Hard is Compiler Code Generation?" in Automata, Languages and Programming edited by A. Salomaa and M. Steinby. Springer-Verlag, 1 - 15 (1977).

2) Ammann U. "On Code Generation in a PASCAL Compiler". Software-Practice and Experience, Vol. 7, 391 - 423 (1977).

3) Bell C.G. and Newell A. "Computer Structures : Readings and Examples". McGraw-Hill (1971).

4) Brown P.J. (editor) "Software Portability". Cambridge University Press (1977).

5) "Central Processors". ICL Technical Publication 4412 (1976).

6) Chai W.A. and Chai H.W. "Programming Standard COBOL". Academic Press (1976).

7) "COBOL". ICL Technical Publication 4427 (1976).

8) "COBOL compilers". ICL Technical Publication 4426 (1976).

9) Donovan J.J. "Systems Programming". McGraw-Hill (1972).

10) Earley J. and Sturgis H. "A Formalism for Translator Interactions". Comm. ACM, Vol. 13, 607 - 617 (1970).

11) Elson M. and Rake S.T. "Code-generation technique for large-language compilers". IBM Systems Journal, Vol. 9, 166 - 188 (1970).

12) Fraser C.W. "A Knowledge-based Code Generator Generator". Sigplan Notices, Vol. 12, 126 - 129 (1977).

13) Giles P. "Mini-COBOL". The Computer Journal, Vol. 12, 208 - 214 (1969)

14) Griffiths M. "Introduction to Compiler-compilers" in Compiler Construction An Advanced Course edited by G. Goos and J. Hartmanis. Springer-Verlag, 356 - 364 (1974).

15) Koster C.H.A. "Two-level Grammars" in Compiler Construction An Advanced Course edited by G. Goos and J. Hartmanis. Springer-Verlag, 146 - 156 (1974)

16) Lecarme O. and Bochmann G.V. "A (Truly) Usable and Portable Compiler Writing System". Information Processing, 218 - 221 (1974)

17) Lecarme O. and Peyrolle-Thomas M.-C. "Self-compiling Compilers : An Appraisal of their Implementation and Portability". Software-Practice

and Experience, Vol. 8, 149 - 170 (1978).

18) Maginnis J.B. "Fundamental ANSI COBOL Programming". Prentice Hall Inc. (1975).

19) "Model 8/32 Processor User's Manual". Interdata Inc., Publication Number 29-428R01 (1975).

20) "PLAN Reference Manual". ICL Technical Publication 4322 (1972).

21) Poole P.C. and Waite W.M. "Portability and Adaptability". in Advanced Course in Software Engineering edited by F.L. Bauer. Springer-Verlag, 183 - 277 (1973).

22) Powell A.J. "BCPL (1900) System Manual". Queen Mary College.

23) Richards M. "The BCPL Programming Manual". University of Cambridge (1973).

24) Waite W.M. "Code Generation" in Compiler Construction An Advanced Course edited by G. Goos and J. Hartmanis. Springer-Verlag, 302 - 332 (1974).

25) Williams M.H. "A Formal Notation for Specifying Syntax Interpretation Rules" to appear in Computer Languages.

26) Williams M.H. and Bulmer A.R. "Array Handling in COBOL Compilers". Software-Practice and Experience, Vol. 7, 469 - 474 (1977).

27) Wirth N. "The Design of a PASCAL Compiler". Software-Practice and Experience, Vol. 1, 309 - 333 (1971).

APPENDIX 1

THE ADDRESS TABLE AND DATA AREA REQUIREMENTS OF THE CODE GENERATOR GENERATOR

ADDRESS TABLE NUMBER	FUNCTION
1	An area of storage large enough to hold the index to the pseudo-code action table or the index defined as a list of constants
2	The pseudo-code action buffer large enough to hold the longest pseudo-code action
3	The pseudo-code action parameter storage area
4	A parameter list to hold the parameters for local routines.
5	The instruction buffer data block
6	The instruction buffer
7	The relocatable buffer data block
8	The relocatable buffer
9	The standard location used in instruction assembly
10	The condition type register
11	A condition operand flag
12	Condition operand one
13	Condition operand two
14	The address table
17	The data area one data block
18	The data area one buffer
19	The data area two data block
20	.
.	.
.	.

31	The start address of the fetch next symbol local routine
32	Fetch pseudo-code action routine
33	Write a buffer to disc routine
34	A temporary work area (7 basic units in length)
35	Address constants for address table elements 5,6,7,8,17,18,19, ... (depending upon the number of defined data areas)
36	A pseudo-code action pointer
51	The source symbol flag indicating the source of a definition action i.e. indicating whether the definition action was encountered in the pseudo-code list or within a pseudo-code action
53 - 57	To hold return addresses for nested routine calls
58	The result of a call to the routine to fetch the next symbol
59	A temporary store (1 basic unit long)
96	To contain the buffer number relative to address table element 35 for use by the routine to write a buffer to a disc file
97	The list of free index registers
98	The free register list
99	The index register use stack
100	The register use stack
101	The branch backward stack
102	The forward branch stack
104	The pseudo-code action number to be

	fetched in a call to the fetch pseudo-code action routine
113	The length of the pseudo-code action as returned by the fetch pseudo-code action routine

ACTION START ADDRESS TABLE ELEMENTS

60	Define store
61	Define constant
62	Align
63	Transfer address
64	Define address constant
65	Define local routine
66	Tabulate address or current address
67	Initialization action
68	Final action
69	Stack the current address
70	Resolve forward branch
71	Local routine parameter
72	Pseudo-code action parameter
73	Add or subtract a constant
74	Extract bits
75	Immediate bits
76	Immediate characters
77	Relocatable instruction
78	Call a local routine
79	Condition handling
80	The end of a condition operand definition
81	Then
82	The end of then code
83	Fetch a register
84	Free a register

85	Literal
86	Address table access
87	Multiply by a constant
88	Relative branch
89	Use a register
90	The end of else code
91	Unstack an address from the branch backward stack
92	Branch forward
93	Use index register

ADDITIONAL DATA AREAS USED BY THE 1900 SYSTEM

40	The control area for the output file
41	The bucket number to be written to
42	An output disc buffer
43	The start address of the disc buffer (i.e. address table element 42)
44	The control area for the input file containing the pseudo-code action table
45	The start address of the pseudo-code action buffer (i.e. address table element 2)
46	The pseudo-code input buffer
47	The amount of non-interpreted space in the pseudo-code buffer
48	The current position within the pseudo-code buffer
49	The control area for the input file containing the pseudo-codes
50	The current bucket number in the pseudo-code buffer

52	The allowable operators in a local routine definition (i.e. the constants 14,15,16,17,26,27,28,29)
94	The open file control areas for the pseudo-code file
103	The address of the pseudo-code input buffer
105	Additional buffer for fetching a non-optimize pseudo-code action
106	The address of the non-optimized pseudo-code action buffer
107	The program name
108	The start address of the controlling segment
109	The open file control area for the pseudo-code action table file
110	The open file control area for the output file
112	The close file control area
114	The open file control area for the pseudo-code action table index
115	The start address of the pseudo-code action table index

APPENDIX 2

THE FORMAL DEFINITION OF THE ACTION ANALYZING ROUTINE

```
DEFINE STORE.
-------------

    FETCHNEXT(DATAAREA)
    FETCHNEXT(ADDRESSTABNO)
    FETCHNEXT(SIZE)
    DATAAREA = (DATAAREA+1)*2
    AREASTART = [135+DATAAREA]
    114+ADDRESSTABNO = [AREASTART+3]+[AREASTART]
    LOOP: IF SIZE > [AREASTART+2]
             THEN
                BEGIN
                SIZE = SIZE-[AREASTART+2]
                WRITEBUFFER(DATAAREA)
                AREASTART = 0
                AREASTART+2 = [AREASTART+1]
                GO TO LOOP
                END
    AREASTART = [AREASTART]+SIZE
    AREASTART+2 = [AREASTART+2]-SIZE

DEFINE CONSTANT
---------------

    FETCHNEXT(DATAAREA)
    FETCHNEXT(ADDRESSTABNO)
    FETCHNEXT(NOOFCONSTANTS)
    DATAAREA = (DATAAREA+1)*2
    AREASTART = [135+DATAAREA]
    AREABUFFER = [135+DATAAREA+1]
    114+ADDRESSTABNO = [AREASTART+3]+[AREASTART]
    FOR I = 1 STEP 1 UNTIL NOOFCONSTANTS DO
       BEGIN
       IF [AREASTART+2] = 0
          THEN
             BEGIN
             WRITEBUFFER(DATAAREA)
             AREASTART = 0
             AREASTART+2 = [AREASTART+1]
             END
       FETCHNEXT(CONSTANT)
       AREABUFFER+[AREASTART] = CONSTANT
       AREASTART = [AREASTART]+1
       AREASTART+2 = [AREASTART+2]-1
       END


ALIGN
-----

    FETCHNEXT(DATAAREA)
    FETCHNEXT(BOUNDARY)
    IF DATAAREA = 0
       THEN
          AREASTART = [135]
       ELSE
```

```
        BEGIN
        DATAAREA = (DATAAREA+1)*2
        AREASTART = [135+DATAAREA]
        END
    REM = [AREASTART] - ([AREASTART]'/'BOUNDARY)*BOUNDARY
    IF REM # 0
      THEN
        BEGIN
        BOUNDARY = BOUNDARY-REM
        IF BOUNDARY > [AREASTART+2]
          THEN
            BEGIN
            BOUNDARY = BOUNDARY-[AREASTART+2]
            WRITEBUFFER(DATAAREA)
            IF DATAAREA = 0
              THEN
                BEGIN
                AREASTART = 1
                AREASTART+2 = [AREASTART+1]*WORDSIZE
                END
              ELSE
                BEGIN
                ARAEASTART = 0
                AREASTART+2 = [AREASTART+1]
                END
            END
        AREASTART = [AREASTART]+BOUNDARY
        AREASTART+2 = [AREASTART+2]-BOUNDARY
        END
```

TRANSFER ADDRESS

```
    FETCHNEXT(DATAAREA1)
    IF DATAAREA1 = 0
      THEN
        AREASTART1 = [135]
      ELSE
        BEGIN
        DATAAREA1 = (DATAAREA1+1)*2
        AREASTART1 = [135+DATAAREA1]
        END
    FETCHNEXT(DATAAREA2)
    IF DATAAREA2 = 0
      THEN
        AREASTART2 = [135]
      ELSE
        BEGIN
        DATAAREA2 = (DATAAREA2+1)*2
        AREASTART2 = [135 +DATAAREA2]
        END
    IF DATAAREA1 = 0
      THEN
        VALUE = ([AREASTART1]+(WORDSIZE-2))'/'WORDSIZE
      ELSE
        VALUE = [AREASTART1]
    AREASTART2+3 = VALUE+[AREASTART1+3]
    IF DATAAREA2 = 0
      THEN
        BEGIN
```

```
          AREASTART2 = 1
          AREASTART2+2 = [AREASTART2+1]*WORDSIZE
          END
       ELSE
          BEGIN
          AREASTART2 = 0
          AREASTART2+2 = [AREASTART2+1]
          END
    WRITEBUFFER(DATAAREA1)
```

DEFINE ADDRESS CONSTANT

```
    FETCHNEXT(DATAAREA)
    FETCHNEXT(ADDRESSTABNO)
    FETCHNEXT(NOOFCONSTANTS)
    DATAAREA =(DATAAREA+1)*2
    AREASTART = [I35+DATAAREA]
    AREABUFFER = [I35+DATAAREA+1]
    I14+ADDRESSTABNO = [AREASTART]+[AREASTART+3]
    FOR I = 1 STEP 1 UNTIL NOOFCONSTANTS DO
       BEGIN
       IF [AREASTART+2] = 0
          THEN
             BEGIN
             WRITEBUFFER(DATAAREA)
             AREASTART = 0
             AREASTART+2 = [AREASTART+1]
             END
       FETCHNEXT(ADDRESS)
       AREABUFFER+[AREASTART] = [I14+ADDRESS]
       AREASTART = [AREASTART]+1
       AREASTART+2 = [AREASTART+2]-1
       END
```

DEFINE LOCAL ROUTINE

```
    FETCHNEXT(ADDRESSTABNO)
    I14+ADDRESSTABNO = [I5+3]+[I5]'/'WORDSIZE
    FETCHNEXT(LENGTH)
    I113 = LENGTH
    I36 = 0
    LENGTH = (LENGTH+1)'/'CPBU
    COMMENT**CPBU = CHARACTERS PER BASIC STORAGE UNIT**
    FOR I = 0 STEP 1 UNTIL LENGTH-1 DO
       BEGIN
       FETCHNEXT(WORD)
       I2+I = WORD
       END
    FOR K = [I2+[I36]] WHILE [I36] <= [I113] DO
       IF K = 14 THEN CALL(I73)
        ELSE IF K = 15 THEN CALL(I74)
         ELSE IF K = 16 THEN CALL(I75)
          ELSE IF K = 17 THEN CALL(I76)
           ELSE IF K = 26 THEN CALL(I85)
            ELSE IF K = 27 THEN CALL(I86)
             ELSE IF K = 28 THEN CALL(I87)
              ELSE IF K = 29 THEN CALL(I88)
```

```
            ELSE ERROR(10)

DEFINE ADDRESS TABLE ELEMENT
----------------------------

    FETCHNEXT(ADDRESSTABNO)
    FETCHNEXT(VALUE)
    IF VALUE = 0
        THEN
            !14+ADDRESSTABNO = [!5+3]+[!5]'/'WORDSIZE
        ELSE
            !14+ADDRESSTABNO = VALUE

STACK THE CURRENT ADDRESS
-------------------------

    COMMENT**ADDRESS TABLE ELEMENT 101 CONTAINS THE START ADDRESS
                OF THE BACKWARD BRANCH STACK, ELEMENT 0 CONTAINS THE
                STACK POINTER.**
    !101 = [!101]+1
    !101+[!101] = [!5+3]+[!5]'/'WORDSIZE


UNSTACK ADDRESS
---------------

    IF [!101] < 1
        THEN
            ERROR(11)
        ELSE
            BEGIN
            !9 = [!101+[!101]]
            !101 = [!101]-1
            END


RESOLVE FORWARD BRANCH
----------------------

    COMMENT**ADDRESS TABLE ELEMENT 102 CONTAINS THE START ADDRESS
                OF THE FORWARD BRANCH STACK, ELEMENT 0 CONTAINS THE
                STACK POINTER, A NEGATIVE VALUE REPRESENTS "THEN" AND
                A POSITIVE VALUE "ELSE".
    VALUE = [!102+[!102]]
    IF VALUE < 0
        THEN
            VALUE = -VALUE
    ADDRESS = [!5+3]+[!5]'/'WORDSIZE
    ORS(!6+ADDRESS,VALUE)
    COMMENT**ORS IS A PROCEDURE TO LOGICALLY OR PARAMETER 2 INTO
                THE ADDRESS CONTAINED IN PARAMETER 1.**
    !102 = [!102]-1
    IF [!102] = 0
        THEN
            BEGIN
            WHILE [!5] > INSTRUCTIONBUFFERSIZE DO
                BEGIN
                WRITEBUFFER(0)
                COLLAPSE BUFFER
```

```
                !5 = [!5]-INSTRUCTIONBUFFERSIZE
                END
            !5+2 = INSTRUCTIONBUFFERSIZE-[!5]
            END


BRANCH FORWARD
--------------

    !9 = 0
    ADDRESS = -[!5]'/'WORDSIZE
    IF [!102] = 0
        THEN
            BEGIN
            !5+2 = [!5+2]+EXPANDEDBUFFERSIZE
            !102 = 1
            !102+[!102] = ADDRESS
            END
        ELSE
            BEGIN
            IF [!102+[!102]] > 0
                THEN
                    BEGIN
                    !102 = [!102]+1
                    !102+[!102] = ADDRESS
                    END
                ELSE
                    BEGIN
                    ADDRESS = -[!102+[!102]]
                    LOCATION = [!5]'/'WORDSIZE+[!5+3]+1
                    ORS(!6+ADDRESS,LOCATION)
                    !102+[!102] = [!5]'/'WORDSIZE
                    END
            END


PSEUDO-CODE ACTION PARAMETER
----------------------------

    I = [!2+[!36]]-1
    !9 = [!3+I]
    !36 = [!36]+1


ADDRESS TABLE ACCESS
--------------------

    !9 = [!14+[!9]]


CALL LOCAL ROUTINE
------------------

    ROUTINENO = [!2+[!36]]-1
    !36 = [!36]+1
    ROUTINEADDRESS = [!30+ROUTINENO]
    CALL(ROUTINEADDRESS)


LOCAL ROUTINE PARAMETER
-----------------------
```

```
    I = [!2+[!36]]-1
    !4+1 = [!9]
    !36 = [!36]+1
```

ADD OR SUBTRACT A CONSTANT

```
    SIGN = [!2+[!36]]
    VALUE = [!2+[!36]+1]*64 + [!2+[!36]+2]
    IF SIGN = 0
       THEN
          !9 = [!9]+VALUE
       ELSE
          !9 = [!9]-VALUE
    !36 = [!36]+3
```

MULTIPLY BY A CONSTANT

```
    VALUE = [!2+[!36]]*64 + [!2+[!36]+1]
    !36 = [!36]+2
    !9 = [!9]*VALUE
```

RELATIVE BRANCH

```
    STARTPOS = [!2+[!36]]
    NOOFBITS = [!2+[!36]+1]
    SIGN = [!2+[!36]+2]
    AMOUNT = [!2+[!36]+3]*64 + [!2+[!36]+4]
    !36 = [!36]+5
    IF SIGN = 0
       THEN
          !9 = [!5+3]+[!5]'/'WORDSIZE+AMOUNT
       ELSE
          !9 = [!5+3]+[!5]'/'WORDSIZE-AMOUNT
    IF NOOFBITS > [!5+2]
       THEN
          BEGIN
          WRITEBUFFER(0)
          !5 = 1
          !5+2 = [!5+1]*WORDSIZE
          END
    SLL(!9,WORDSIZE-NOOFBITS)
    SRL(!9,STARTPOS-1)
    COMMENT**SLL AND SRL ARE PROCEDURES TO PERFORM LOGICAL LEFT
             AND RIGHT SHIFTS RESPECTIVELY ON THE CONTENTS OF THE
             LOCATION WHOSE ADDRESS IS PARAMETER 1, THE NUMBER OF
             PLACES BY WHICH THE VALUE MUST BE SHIFTED IS INDICATED
             AS PARAMETER 1.
    CURRENTPOS = [!5]'/'WORDSIZE
    ORS(!6+CURRENTPOS,[!9])
    !5 = [!5]+NOOFBITS
    !5+2 = [!5+2]-NOOFBITS
```

```
RELOCATABLE INSTRUCTION
-----------------------

   DATAAREA = [!2+[!36]]
   !36 = [!36]+1
   IF DATAAREA = 0
      THEN
         LOCATION = [!5]'/'WORDSIZE + [!5+3]
      ELSE
         BEGIN
         AREASTART = [!35+(DATAAREA+1)*2)]
         LOCATION = [AREASTART]+[AREASTART+3]
         END
   IF [!7+2] < 2
      THEN
         BEGIN
         WRITEBUFFER(2)
         !7 = 0
         !7+2 = [!7+1]
         END
   !8+[!7] = LOCATION
   !8+[!7]+1 = DATAAREA
   !7 = [!7]+2
   !7+2 = [!7+2]-2


CONDITION
---------

   !10 = [!2+[!36]]
   !36 = [!36]+1
   !11 = 0


END OF CONDITION OPERAND
------------------------

   IF [!11] = 0
      THEN
         BEGIN
         !12 = [!9]
         !11 = 1
         END
      ELSE
         !13 = [!9]


THEN
----

   I = [!36]+2
   J = [!36] + [!2+[!36]]*64 + [!2+[!36]+1]
   IF [!10] = 1
      THEN
         IF [!12] = [!13]
            THEN
               !36 = I
            ELSE
               !36 = J
   IF [!10] = 2
      THEN
```

```
        IF [|12] # [|13]
           THEN
              |36 = I
           ELSE
              |36 = J
  IF [|10] = 3
     THEN
        IF [|12] > [|13]
           THEN
              |36 = I
           ELSE
              |36 = J
  IF [|10] = 4
     THEN
        IF [|12] < [|13]
           THEN
              |36 = I
           ELSE
              |36 = J
  IF [|10] = 5
     THEN
        IF [|12] >= [|13]
           THEN
              |36 = I
           ELSE
              |36 = J
  IF [|10] = 6
     THEN
        IF [|12] <= [|13]
           THEN
              |36 = I
           ELSE
              |36 = J


END OF TRUE CODE
----------------

  |36 = [|36] + [|2+[|36]]*64 + [|2+[|36]+1]


FETCH A REGISTER
----------------

  COMMENT**ADDRESS TABLE ELEMENT  97 : INDEX REGISTER FREE STACK
           ADDRESS TABLE ELEMENT  98 : REGISTER FREE STACK
           ADDRESS TABLE ELEMENT  99 : INDEX REGISTER USE STACK
           ADDRESS TABLE ELEMENT 100 : REGISTER USE STACK
          IN EACH CASE ELEMENT 0 CONTAINS THE STACK POINTER**
  TYPE = [|2+[|36]]
  |36 = [|36]+1
  IF TYPE = 0
     THEN
        BEGIN
           IF [|97] = 0
           THEN
              ERROR(16)
           ELSE
              BEGIN
              REGNO = [|97+[|97]]
              |97 = [|97]-1
```

```
                  !99 = [!99]+1
                  !99+[!99] = REGNO
                  END
            END
         ELSE
            BEGIN
            IF TYPE = 1
               THEN
                  BEGIN
                  IF [!98] = 0
                     THEN
                        ERROR(16)
                     ELSE
                        BEGIN
                        REGNO = [!98+[!98]]
                        !98 = [!98]-1
                        !100 = [!100]+1
                        !100+[!100] = REGNO
                        END
                  END
               ELSE
                  BEGIN
                  IF [!98] < 2
                     THEN
                        ERROR(16)
                     ELSE
                        BEGIN
                        REGNO1 = [!98+[!98]]
                        REGNO2 = [!98+[!98]-1]
                        !98 = [!98]-2
                        !100+[!100]+1 = REGNO1
                        !100+[!100]+2 = REGNO2
                        !100 = [!100]+2
                        END
                  END
            END


FREE A REGISTER
-------------

   TYPE = [!2+[!36]]
   !36 = [!36]+1
   IF TYPE = 0
      THEN
         BEGIN
         IF [!99] = 0
            THEN
               ERROR(15)
            ELSE
               BEGIN
               !99 = [!99]-1
               !97 = [!97]+1
               END
         END
      ELSE
         BEGIN
         IF [!100] = 0
            THEN
               ERROR(15)
            ELSE
```

```
                 BEGIN
                 !100 = [!100]-1
                 !98 = [!98]+1
                 END
           END
```

USE A REGISTER

```
    REGNO = [!2+[!36]]
    !36 = [!36]+1
    !9 = [!100+[!100]-REGNO]
```

USE AN INDEX REGISTER

```
    REGNO = [!2+[!36]]
    !36 = [!36]+1
    !9 = [!99+[!99]-REGNO]
```

EXTRACT BITS

```
    STARTPOS = [!2+[!36]]
    NOOFBITS = [!2+[!36]+1]
    !36 = [!36]+2
    IF NOOFBITS > [!5+2]
       THEN
          BEGIN
          WRITEBUFFER(0)
          !5 = 1
          !5+2 = [!5+1]*WORDSIZE
          END
    SLL(!9,WORDSIZE-NOOFBITS)
    SRL(!9,STARTPOS-1)
    CURRENTPOS = [!5]'/'WORDSIZE
    ORS(!6+CURRENTPOS,[!9])
    !5 = [!5]+NOOFBITS
    !5+2 = [!5+2]-NOOFBITS
```

IMMEDIATE BITS

```
    NOOFBITS = [!2+[!36]]
    !9 = [!2+[!36]+1]
    !36 = [!36]+2
    IF NOOFBITS > [!5+2]
       THEN
          BEGIN
          WRITEBUFFER(0)
          !5 = 1
          !5+2 = [!5+1]*WORDSIZE
          END
    SLL(!9,WORDSIZE-BYTESIZE)
    STARTPOS = [!5] - (([!5]'/'WORDSIZE)*WORDSIZE)
    SRL(!9,STARTPOS-1)
    CURRENTPOS = [!5]'/'WORDSIZE
```

```
    ORS(I6+CURRENTPOS,[I9])
    I5 = [I5]+NOOFBITS
    I5+2 = [I5+2]-NOOFBITS
```

IMMEDIATE CHARACTERS

```
    NOOFCHARS = [I2+[I36]]
    I36 = [I36]+1
    FOR I = 1 STEP 1 UNTIL NOOFCHARS DO
        BEGIN
        I9 = [I2+[I36]]
        I36 = [I36]+1
        IF [I5+2] < BYTESIZE
           THEN
              BEGIN
              WRITEBUFFER(0)
              I5 = 1
              I5+2 = [I5+1]*WORDSIZE
              END
        SLL(I9,WORDSIZE-BYTESIZE)
        STARTPOS = [I5] - (([I5]'/'WORDSIZE)*WORDSIZE)
        SRL(I9,STARTPOS-1)
        CURRENTPOS = [I5]'/'WORDSIZE
        ORS(I6+CURRENTPOS,[I9])
        I5 = [I5]+BYTESIZE
        I5+2 = [I5+2]-BYTESIZE
        END
```

LITERAL

```
    I9 = 0
    FOR I = 1 STEP 1 UNTIL CPBU DO
        BEGIN
        SLL(I9,BYTESIZE)
        I9 = [I9]+[I2+[I36]]
        I36 = [I36]+1
        END
```

THE CONTROL SEGMENT

```
    COMMENT**I51 = SOURCE FLAG,I36 = PSEUDO-CODE ACTION POINTER**
    SWITCH DEFINITION = DEF1,DEF2,DEF3,DEF4,DEF5,DEF6,DEF7,ERR,DEF9,
                        DEF10,DEF11
    SWITCH ACTION = ACT1,ACT2,ACT3,ERR,ACT5,ERR,ACT7,ERR,ERR,ACT10,
                    ACT11,ACT12,ACT13,ACT14,ACT15,ACT16,ACT17,ACT18,
                    ACT19,ACT20,ACT21,ACT22,ACT23,ACT24,ACT25,ACT26,
                    ACT27,ACT28,ACT29,ACT30,ACT31,ACT32,ACT33,ACT34
    CALL(I67)
    CONTROLLOOP:FETCHNEXT(PSEUDOCODE)
    I51 = 0
    IF PSEUDOCODE < 12 THEN GO TO DEFINITION(PSEUDOCODE)
    I51 = 1
    FETCHACTION(PSEUDOCODE)
    PSEUDOCODE = PSEUDOCODE-12
    PARAMETERS = [I1+PSEUDOCODE*3+2]
```

```
IF PARAMETERS > 0
   THEN
      FOR I = 1 STEP 1 UNTIL PARAMETERS DO
         BEGIN
         FETCHNEXT(PARAMETER)
         |3+I-1 = PARAMETER
         END
|36 = 0
ACTIONLOOP:IF [|36] >= [|113]
              THEN
                 GO TO CONTROLLOOP
              ELSE
                 BEGIN
                 I = [|2+[|36]]
                 |36 = [|36]+1
                 GO TO ACTION(I)
                 END
ACT1:CALL(|60)
GO TO ACTIONLOOP
ACT2:CALL(|61)
GO TO ACTIONLOOP
ACT3:CALL(|62)
GO TO ACTIONLOOP
ACT5:CALL(|64)
GO TO ACTIONLOOP
ACT7:CALL(|66)
GO TO ACTIONLOOP
ACT10:CALL(|69)
GO TO ACTIONLOOP
ACT11:CALL(|70)
GO TO ACTIONLOOP
ACT12:CALL(|71)
GO TO ACTIONLOOP
ACT13:CALL(|72)
GO TO ACTIONLOOP
ACT14:CALL(|73)
GO TO ACTIONLOOP
ACT15:CALL(|74)
GO TO ACTIONLOOP
ACT16:CALL(|75)
GO TO ACTIONLOOP
ACT17:CALL(|76)
GO TO ACTIONLOOP
ACT18:CALL(|77)
GO TO ACTIONLOOP
ACT19:CALL(|78)
GO TO ACTIONLOOP
ACT20:CALL(|79)
GO TO ACTIONLOOP
ACT21:CALL(|80)
GO TO ACTIONLOOP
ACT22:CALL(|81)
GO TO ACTIONLOOP
ACT23:CALL(|82)
GO TO ACTIONLOOP
ACT24:CALL(|83)
GO TO ACTIONLOOP
ACT25:CALL(|84)
GO TO ACTIONLOOP
ACT26:CALL(|85)
GO TO ACTIONLOOP
```

```
ACT27:CALL(186)
GO TO ACTIONLOOP
ACT28:CALL(187)
GO TO ACTIONLOOP
ACT29:CALL(188)
GO TO ACTIONLOOP
ACT30:CALL(189)
GO TO ACTIONLOOP
ACT31:CALL(190)
GO TO ACTIONLOOP
ACT32:CALL(191)
GO TO ACTIONLOOP
ACT33:CALL(192)
GO TO ACTIONLOOP
ACT34:CALL(193)
GO TO ACTIONLOOP
DEF1:CALL(160)
GO TO CONTROLLOOP
DEF2:CALL(161)
GO TO CONTROLLOOP
DEF3:CALL(162)
GO TO CONTROLLOOP
DEF4:CALL(163)
GO TO CONTROLLOOP
DEF5:CALL(164)
GO TO CONTROLLOOP
DEF6:CALL(165)
GO TO CONTROLLOOP
DEF7:CALL(166)
GO TO CONTROLLOOP
DEF9:CALL(168)
GO TO CONTROLLOOP
DEF10:CALL(169)
GO TO CONTROLLOOP
DEF11:CALL(170)
GO TO CONTROLLOOP
ERR:ERROR(10)
```

APPENDIX 3

THE CODE SKELETON ACTION IDENTIFIERS

IDENTIFIER	FUNCTION
#1	Define store
#2	Define constant
#3	Align
#5	Define address constant
#7	Define address table element
#10	Stack the current address for a backward branch
#11	Resolve a forward branch
%i	The use of pseudo-code action parameter i
!i	The use of address table element i
+i,-i,*i	Add, subtract or multiply by i
$i	Marks an instruction as being relocatable with respect to data area i
↑i,↑i(parameters)	Call local routine i, with the parameters indicated in the parameter list
£i	Fetch a register where i indicates the register type For the 1900 implementation 0 - index register, 1 - general purpose register, 2 - 2 adjacent general purpose registers
?i	Free a register of type i
@i	Use the register i positions from the top of the register use stack
&i	Use the index register i positions from the top of the index register use stack
-	Unstack the address at the top of the backward branch stack
+	Add the address of the current instruction to the forward branch stack i.e. branch forward
IF,THEN	The keywords to be used in a conditional code
ELSE,FI	insertion statement
=,#,>	The relational operators which may appear in the
<,>=,<=	condition part of a conditional code insertion statement

APPENDIX 4

THE PSEUDO-CODE OR PSEUDO-CODE ACTION TABLE IDENTIFIERS

For the actions which may appear within the pseudo-code action table all parameters for action 1,2,3,5 and 7 are four characters in length. In all other cases they are assumed to be one character in length unless otherwise stated.

ACTION NUMBER	FUNCTION	PARAMETERS
1	Define store	The data area in which the storage is to be defined
		The address table element to be associated with the first word of the defined storage area
		The number of words (or bytes etc.) to be defined
2	Define constant	The data area number
		The associated address table element
		The number of constants in the list
		The constants to be defined
3	Align	The data area number
		The boundary multiple
4	Transfer address	Sending data area number
		Receiving data area number
5	Define address constant	The data area number
		The associated address table element
		The number of address constants in the list
		The address table element numbers
6	Define local routine	Address table element to be associated with the start address of the local routine
		The number of characters of routine code - say n

ACTION NUMBER	FUNCTION	PARAMETERS
		The n characters of routine code
7	Define address table element	The address table element to be defined The value to be assigned
9	Final action	-
10	Add the current address to the branch backwards stack	-
11	Resolve forward branch	-
12	Local routine parameter	The parameter number
13	Pseudo-code parameter	The parameter number
14	Addition or subtraction of a constant	Plus/minus indicator (0 - plus, 1 - minus) The number to be added or subtracted (2 characters)
15	Extract bits	The starting position within the instruction The number of bits
16	Immediate bits	The number of bits The bits left aligned
17	Immediate characters	The number of immediate characters The immediate characters
18	Relocatable instruction	The data area number with respect to which the instruction is relocatable
19	Call local routine	The local routine number
20	Condition	Condition type (1-=,2-#,3->,4-<, 5-<=,6->=)

ACTION NUMBER	FUNCTION	PARAMETERS
21	The end of a condition operand	-
22	The start of the THEN code	The number of characters to be skipped if the condition has evaluated to false i.e. to get to the code to be executed on the false condition (2 characters)
23	The start of the ELSE code (or end of THEN)	The number of instructions to be skipped to get to the code following the FI(2 characters
24	Fetch a register	Register type (0 - index, 1 - register, 2-2 registers)
25	Free a register	Register type
26	Literal	The binary literal (4 characters)
27	Address table access	-
28	Multiply	The multiplication factor (2 characters)
29	Relative branch	The starting position of the field within the instruction The field length in bits Backward/forward flag(0 - forward,1 - backward) The amount to be jumped (2 characters)
30	Use register	The register location relative to the start of the register use stack
31	The end of ELSE code	-

ACTION NUMBER	FUNCTION	PARAMETER
32	Unstack address	-
33	Branch forward	-
34	Use an index register	The index register location within the index register use stack relative to the top of the stack

APPENDIX 5

THE SYNTAX OF THE COBOL SUBSET

```
IDENTIFICATION DIVISION .

PROGRAM-ID . program-name .

ENVIRONMENT DIVISION .

CONFIGURATION SECTION .

SOURCE-COMPUTER . ICL-190n .

OBJECT-COMPUTER . ICL-190n

      MEMORY SIZE integer WORDS .

[SPECIAL-NAMES .

      [hardware-name integer [TYPE type-number] IS  mnemonic-name-1] ...  . ]

[INPUT-OUTPUT SECTION .

FILE-CONTROL .

SELECT file-name-1 ASSIGN TO hardware-name integer  [TYPE type-number] .

[SELECT ... ]  .... ]

DATA DIVISION .

[FILE SECTION .

file description

record description [record description]  ....

[file description ... ]  ... ]

[WORKING-STORAGE SECTION .

record description [record description] ... ]


file description

FD file-name

[FILE CONTAINS ABOUT integer RECORDS]

[BLOCK CONTAINS integer [CHARACTERS] ]

[LABEL  { RECORDS ARE }   { STANDARD  [ WITH  GENERATION-NO] }
        { RECORD IS   }   { OMITTED.                       } ]
```

```
[VALUE OF {ID            } IS {data-name} ]
          {IDENTIFICATION}    {literal  }

[ACTIVE-TIME IS {data-name}]
                {literal  }

[GENERATION-NO IS data-name]

[DATA {RECORD IS  } data-name [data-name] ... ] .
      {RECORDS ARE}

record description :

      level-number {data-name-1}
                   {FILLER     }

[ {PICTURE} IS character-string]
  {PIC    }

[USAGE IS {COMPUTATIONAL} ]
          {COMP         }

[OCCURS integer TIMES]

[ {SYNCHRONIZED} {LEFT } ]
  {SYNC        } {RIGHT}

[VALUE IS literal-1]

[JUSTIFIED [RIGHT]] .

PROCEDURE DIVISION .

     paragraph-name-1 .

     statement-1 [statement-2] ...  .

     [statement-3[statement] ... .] ...

     [paragraph-name-2 . ...] ...

statements

ACCEPT indentifier FROM mnemonic-name
```

```
ADD  { identifier-1 }  [, { identifier-2 } ] ... TO
     { literal-1    }     { literal-2    }

     identifier-m  [ROUNDED]  [ON SIZE ERROR]

     imperative statement ]

ADD  { identifier-1 } , { identifier-2 } [ { identifier-3 } ] .....
     { literal-1    }   { literal-2    }   { literal-3    }

GIVING identifier-n  [ROUNDED]  [ON SIZE ERROR  imperative statement]

CLOSE  file-name-1 [file-name-2] ...

DISPLAY  { identifier-1 } [ { identifier-2 } ]        ... [UPON mnemonic-name]
         { literal-1    }   { literal-2    }

DIVIDE   { identifier-1 }   INTO   { identifier-2 }     GIVING identifier-3
         { literal-1    }          { literal-2    }

     [ROUNDED] [ON SIZE ERROR imperative statement]

EXAMINE identifier REPLACING ALL literal-2 BY literal-3

EXIT

 GO TO procedure-name

IF condition THEN  { statement-1    }  { OTHERWISE }  { statement-2    }
                   { NEXT SENTENCE  }  { ELSE      }  { NEXT SENTENCE  }

MOVE    { identifier-1 }    TO  identifier-2  [identifier-3] ...
        { literal-1    }

MULTIPLY   { identifier-1 }   BY   { identifier-2 }    GIVING identifier-3
           { literal-1    }        { literal-2    }

     [ROUNDED] [ON SIZE ERROR imperative statement]

NOTE  character-string .

         {     INPUT file-name-1 [file-name-2] ...         }
OPEN     {  [OUTPUT file-name-3 [file-name-4] ... ]        }
         {    OUTPUT file-name-1 [file-name-2] ...         }
         {  [ INPUT file-name-3 [file-name-4] ... ]        }

PERFORM procedure-name-1 [THRU procedure-name-2 ] { integer    }  TIMES
                                                  { identifier }

READ file-name RECORD [INTO identifier] AT END imperative statement
```

```
STOP { literal
     { RUN

SUBTRACT { literal-1      } [ { literal-2      } ] ...
         { identifier-1   }   { identifier-2   }

    FROM identifier-m [ROUNDED] [ON SIZE ERROR imperative statement]

SUBTRACT { identifier-1 } [ { identifier-2 } ] .. FROM { literal-m    }
         { literal-1    }   { literal-2    }           { identifier-m }

    GIVING identifier-n [ROUNDED] [ON SIZE ERROR imperative statement]

WRITE record-name [FROM identifier-1] [ { BEFORE } ADVANCING
                                        { AFTER  }

    { identifier-2 LINES }
    { integer LINES      } ]
    { mnemonic-name      }
```

The following changes or additions are relevant to this implementation :

(1) The specifications of the program name, source and object computer are treated as comments.

(2) The TYPE option under SPECIAL-NAMES and for SELECT-ASSIGN TO is treated as a comment.

(3) The FD option FILE CONTAINS ABOUT integer RECORDS is treated as a comment.

(4) The format of data on disc will not necessarily be compatible with other COBOL compilers, though an attempt will be made to produce compatibility.

(5) The FD option DATA RECORDS ARE/RECORD IS ... is treated as a comment.

(6) The PICTURE clause character string can contain the following characters. A, X, 9, 1, S, Z, *, £, 0, -, +, ,, ., B, V or P following the ICL 1900 COBOL rules for combining them, with the following additional restrictions :

- (a) £ may only be followed by £, 9, V, P etc., but not by Z or *.
- (b) Z,*,£ may not occur after a V or P.
- (c) +,- can only occur at the beginning of an edited numeric field, and may not be followed by Z, * or £.
- (d) An alphanumeric field may not start with 9 (i.e. it must start with A or X, or with B, 0, . or , followed by A or X).
- (e) A binary field (1s) may not contain a 1 followed by a V.

(7) The condition as appearing in the IF statement has the following form :

$$\left\{\begin{array}{l}\text{identifier-1}\\ \text{literal-1}\end{array}\right\} \quad [\underline{\text{IS}}]\ [\underline{\text{NOT}}] \left\{\begin{array}{l}\underline{\text{GREATER THAN}}\\ \underline{\text{LESS THAN}}\\ \underline{\text{EQUAL TO}}\end{array}\right\} \quad \left\{\begin{array}{l}\text{identifier-2}\\ \text{literal-2}\end{array}\right\}$$

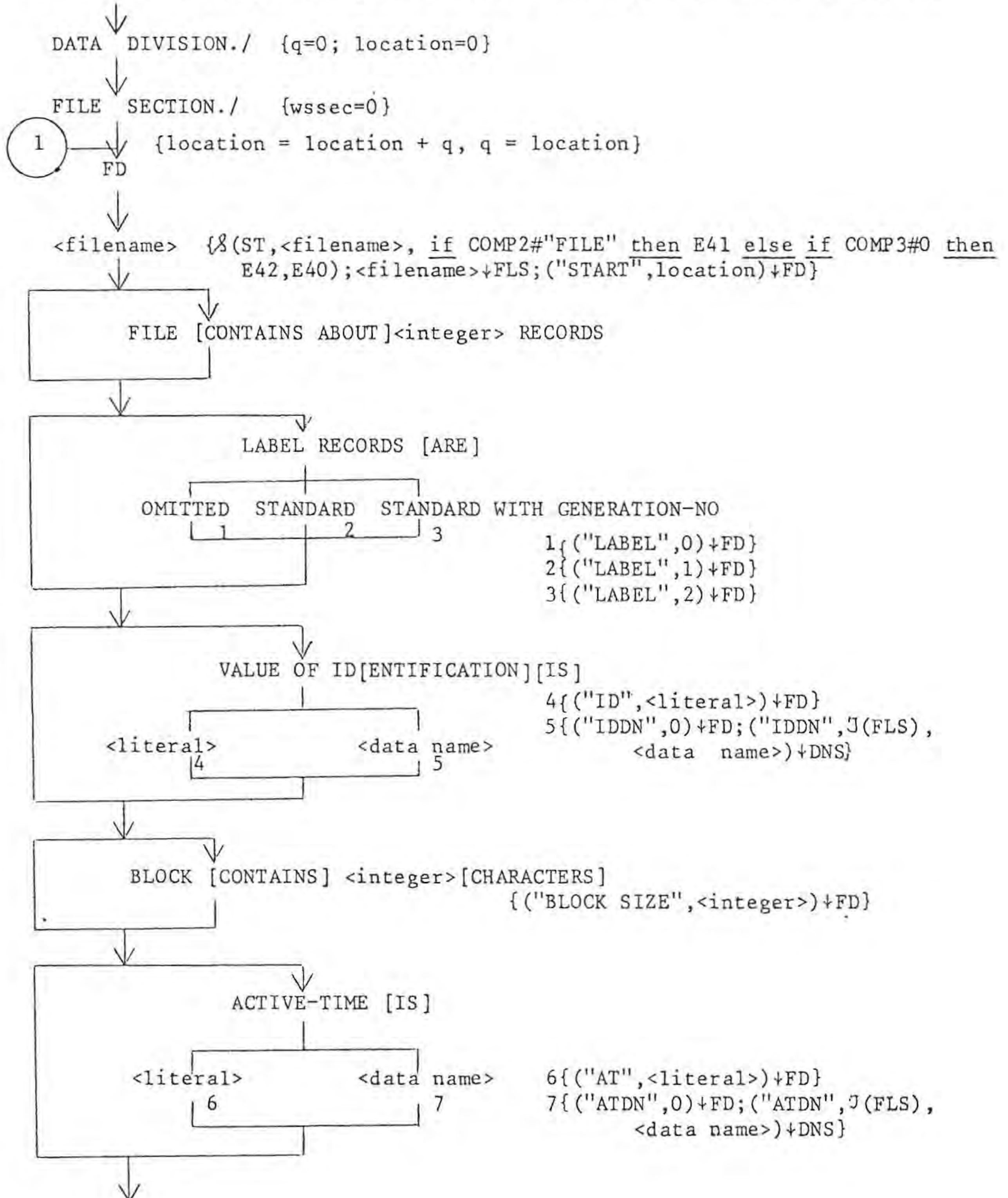
or

$$\text{identifier-1} \quad [\underline{\text{IS}}] \quad [\underline{\text{NOT}}] \left\{\begin{array}{l}\underline{\text{POSITIVE}}\\ \underline{\text{NEGATIVE}}\\ \underline{\text{ZERO}}\end{array}\right\}$$

A P P E N D I X 6

THE FORMAL SPECIFICATION OF THE DATA DIVISION

/ means that the preceding COBOL source must appear on a line of its own



GENERATION-NO IS <data name>
{("GENNO",0)↓FD;("GENNO", Ɔ(FLS),<dataname>)↓DNS}

DATA RECORDS ARE <data-names-list>

{p = location}

<level-number=1>

<data name>

. 1

<data-name-options>

3

2

<level-number>1>

4

FILLER

<data name>

<filler-name-options>.

.

<data-name-options>

5

6

7

8

9

{q = q - location; ("SIZE",q)↓FD;FD⇓ST;λ ⇑ FD;
λ ⇑ FS;λ⇑FSS}

FD

1

WORKING-STORAGE SECTION./
{ wssec=1;action similar
to above}

PROCEDURE DIVISION./

1. {𝒮(ST,<data name>,E47,(<data name>,"RECORD",ℑ(FLS), 0,0,p) ↓ ST }
2. {𝒮(ST,<data name>,E47,(<data name>,"ELEMENTARY RECORD",ℑ(FLS), type, justified, subfields, length, shift, comp, p, subfield list) ↓ ST; p = p + length; <u>if</u> p > q <u>then</u> q=p)}
3. {end of record action (e.g. insert the record length into the "RECORD" entry in the symbol table, for all data items currently in the group items list (section 6.7.3) fill in the appropriate lengths, for all data items in the occurs clause list (section 6.7.4) insert start positions, number of repetitions, intervals of repetition etc.)}
4. {<u>for</u> level-number<= COMP3(ℑ(GS)) <u>do</u> reduce group stack ; <u>for</u> level-number<= COMP6(ℑ(OCS)) <u>do</u> reduce occurs stack}
5. {<u>if</u> level > 0 <u>then</u> ("ELEMENTARY ITEM", sync, p , 0, length,<level-number>)↓ OCS }
6. {<u>if</u> level > 0 <u>then</u> ("GROUP ITEM", 0, p, 0, 0, <level-number>) ↓ OCS}
7. {𝒮 (FS,<data name>,E52 , 𝒮 (ST,<data name>, <u>if</u> COMP2#"UVAR" <u>or</u> "NUVAR" <u>then</u> E53 <u>else</u> COMP2 = "NUVAR"; comp2="NUVAR", comp2 = "UVAR"); (<data name>, comp2,6,0,0,0,0,0,0,0,0,p,0,0,0,0,0) ↓ ST; (<data name>)↓FS; (<data name>, <u>if</u> level > 0 <u>then</u> 1 <u>else</u> 0, <level-number>)↓GS; <u>if</u> level > 0 <u>then</u> ("GROUP FIELD", 0,p,<data name>,0,<level-number>) ↓ OCS)}
8. { 𝒮 (FS,<data name>, E52 , 𝒮 (ST,<data name>, <u>if</u> COMP2# "UVAR" <u>or</u> "NUVAR" <u>then</u> E53 <u>else</u> COMP2 = "NUVAR"; comp2 = "NUVAR", comp2 = "UVAR"); (<data name>, comp2, type, shift, 0,0,0, subfields, length, justified, comp, p, 0,0,0,0, subfield list)↓ST;(<data name>)↓FS; <u>if</u> level > 0 <u>then</u> ("ELEMENTARY ITEM", sync, p,< data name> , length,<level-number>) ↓OCS; p = p + length)}
9. {type = length = shift = subfields = subfield list = repetitions = sync = comp = justified = 0}

The system variables and stack names have the following usage

FD file description information list

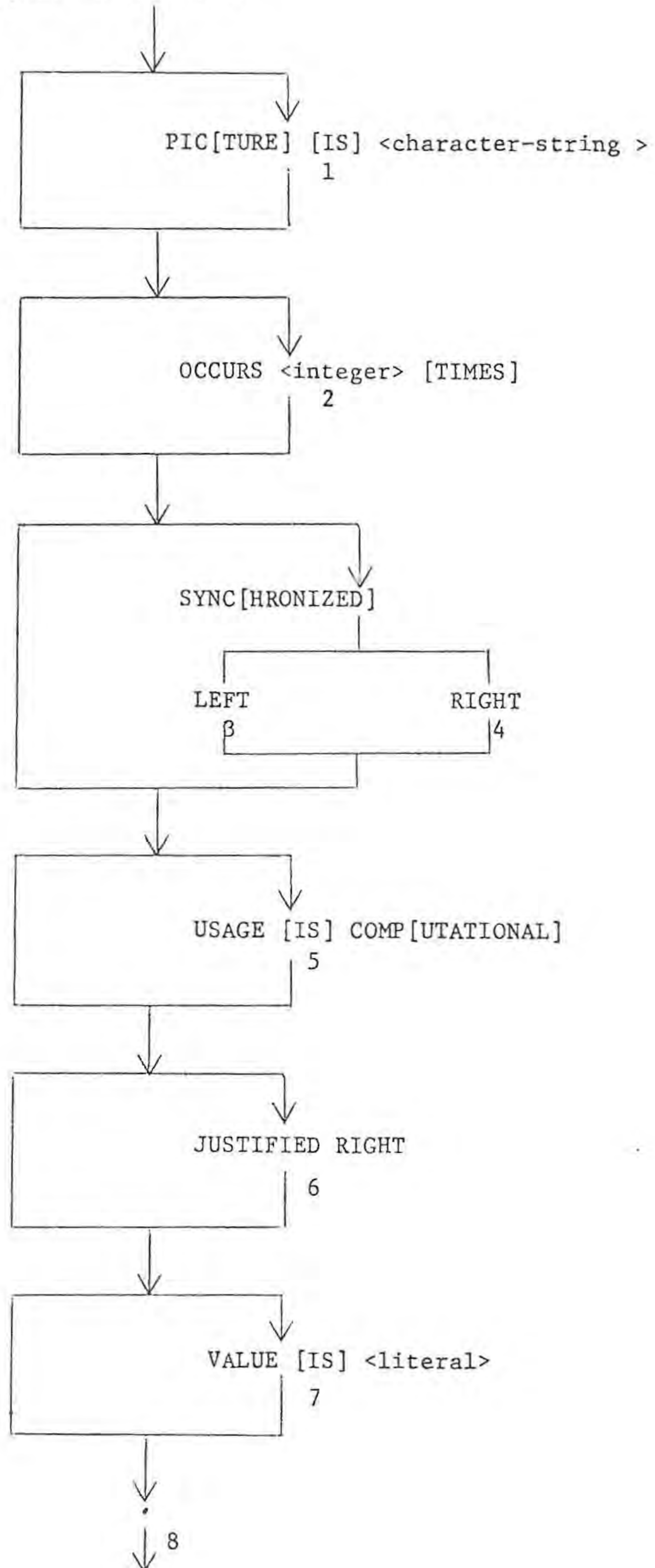
FLS file stack

DNS data names stack

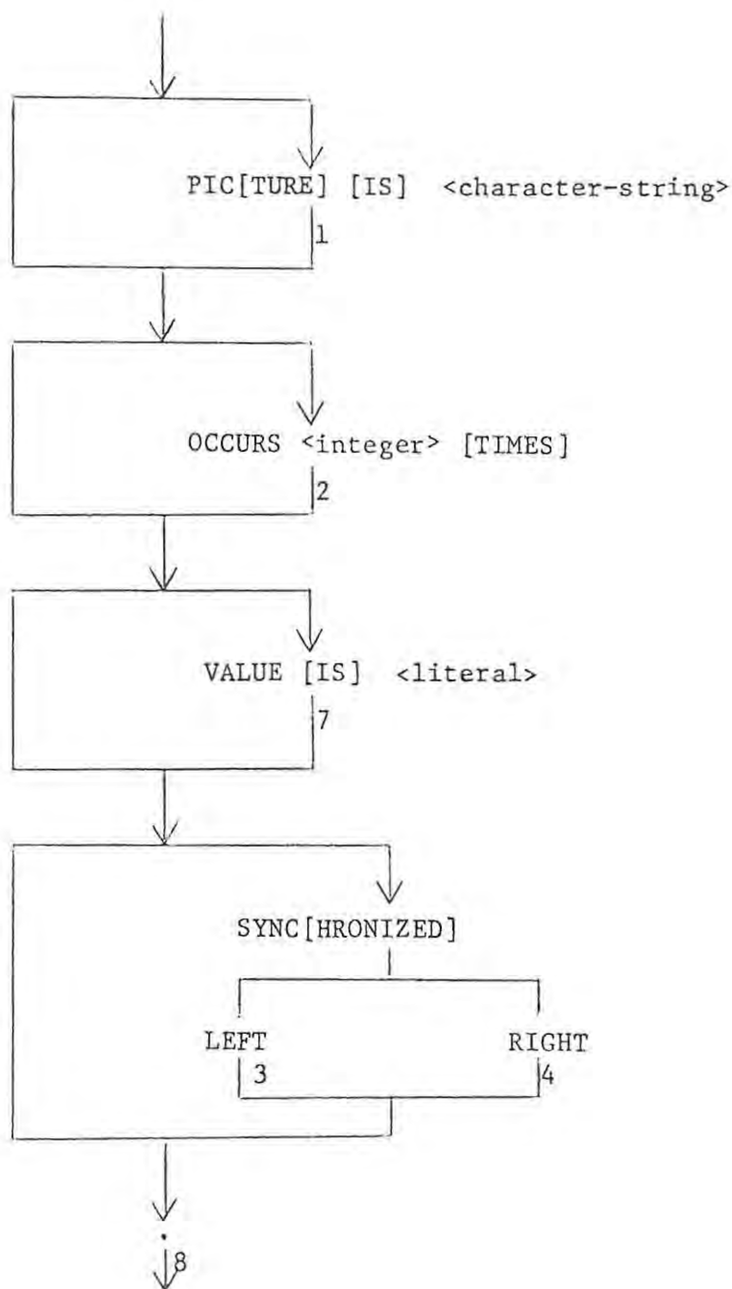
ST	symbol stack
GS	group items stack
OCS	occurs clause handling stack
FS	field names stack
type	the type of the data item(returned by the picture clause analyzer)
p	the current position
q	the size of the largest record within a file
location	the start address of a file buffer
length	the length of a data item (from the pca)
subfields	the number of subfields in the picture clause (from the pca)
subfield list	the subfields (from the pca)
justified	a justified clause flag
comp	a computational clause flag
wssec	a flag indicating the FILE or WORKING-STORAGE SECTION analysis
level	the current level of occurs clause nesting

Data-names-options

The clauses below may appear in any order



Filler-name-options



1 {Picture clause analyzer (type, length, shift, subfields, subfield list)}

2 {repetitions = <integer>}

3 {sync = "LEFT"}

4 {sync = "RIGHT"}

5 {comp = 1}

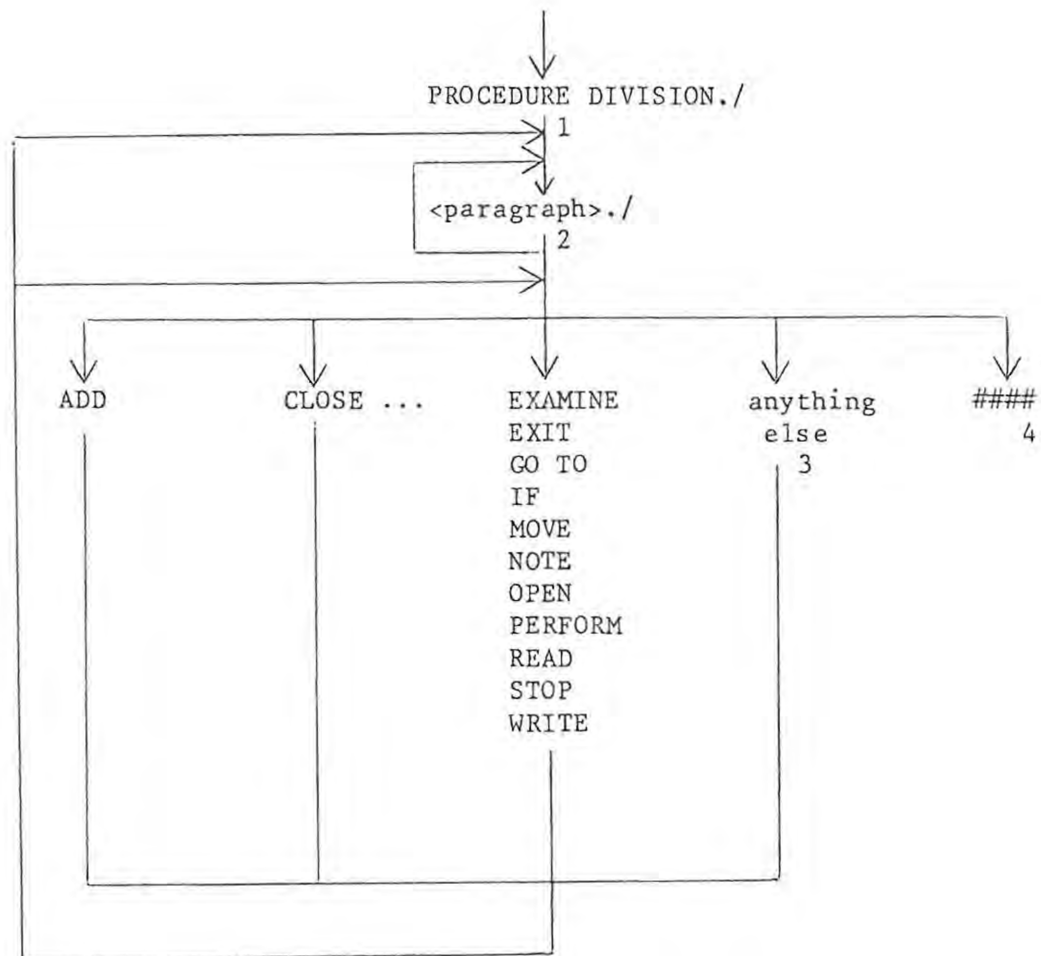
6 {justified = 1}

7 {if wssec = 0 then E62 else as in move statement}

8 {if justified = 1 and type#5 then E58; if repetitions > 1 then (if level = 3 then E72 else level = level+1;("LIST", sync, repetitions, 0,<level-number>) ↓OCS)}

APPENDIX 7

THE FORMAL SPECIFICATION OF THE PROCEDURE DIVISION

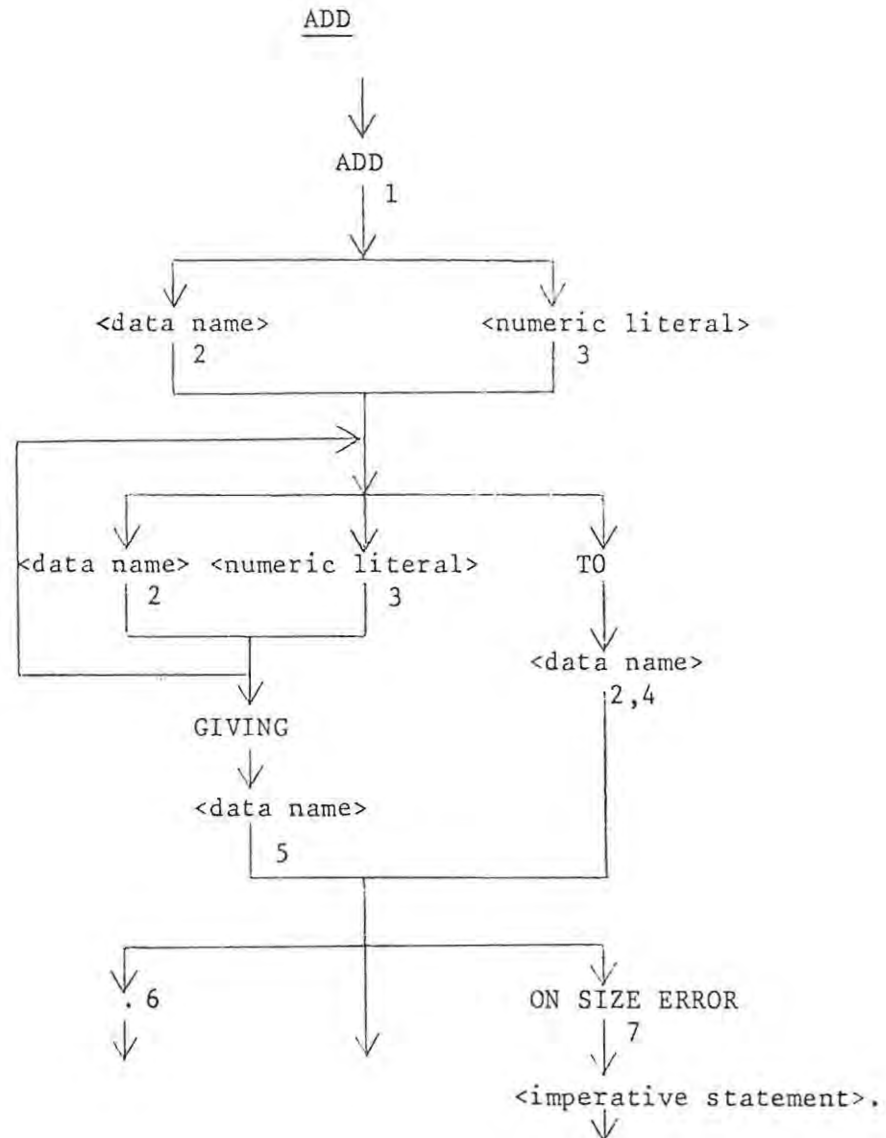


1 {∀DNS : (fill in pointers); pti=0; pi=0; ifcntr=0; impstflag='false'}

2 {𝒮(ST,<paragraph>, <u>if</u> COMP2# "PARAGRAPH" <u>then</u> E164 <u>else</u> <u>if</u> COMP3=0 <u>then</u> E103 <u>else</u> COMP3=0;('PARAGRAPH',COMP4)↓IS, (<paragraph>, "PARAGRAPH",0,pti)↓ST; ('PARAGRAPH',pti)↓IS; pti = pti+1);(<paragraph name>)↓PNS}

3 {E104}

4 {check no undefined paragraphs; check performs; <u>if</u> errorfree <u>then</u> generate code}



1. {('ZEROISE',5)↓IS; maxsh=0; maxlms=0; noop=0}

2. {S(ST,<data name>, if COMP2 = "UVAR" then sign = COMP3; if sign # 0 or 1 then E119 else shift = COMP4; length = COMP9; if COMP5 # 0 then sar(<data name>,i) else sav(<data name>, i) else if COMP2 = "ELEMENTARY RECORD" then sign = COMP4; if sign # 0 or 1 then E119 else shift = COMP8; length = COMP7; saer(<data name>,i) else E119, E120); if shift > maxsh then maxsh = shift; if length - shift>maxlms then maxlms = length - shift; if sign=0 then ('ADD UNSIGNED',i,shift)↓TIS; else ('ADD SIGNED',i,shift)↓TIS; noop = noop+1; if noop > 10 then E 159}

3 {inlt(<numeric literal>, shift, length, sign, i), if shift>maxsh then maxsh=shift; if length - shift>maxlms then maxlms = length - shift; if sign = 0 then ('ADD UNSIGNED',i,shift)↓TIS else ('ADD SIGNED',i,shift)↓TIS; noop = noop+1; if noop>10 then E159}

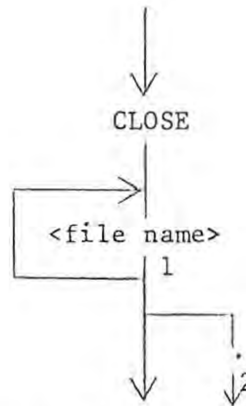
4 {∀TIS:(x↑TIS; COMP3(x)=maxsh - COMP3(x); if COMP3(x) = 0 then if COMP1(x) = 'ADD UNSIGNED' then COMP1(x) = 'ADD UNSIGNED NO X' else COMP1 (x) = 'ADD SIGNED NO X'; x↓IS);λ⇑TIS; lxan (idcode); if sign = 0 then(('CHECK SIZE,SET POSITIVE')↓IS; if idcode = "ON" or "SIZE" then A7; cbd(i,maxlms, maxsh, length - shift, shift)) else (j = ati;('DEFINE STORE',1,ati,1)↓IS; ati = ati+1;('CHECK SIZE,SET POSITIVE,FLAG',j)↓IS; if IDCODE = "ON" or "SIZE" then A7; cbd(i,maxlms,maxsh,length - shift, shift);('CHECK SIZE, INSERT SIGN',i,j)↓IS}

5 {δ(ST,<data name>, if COMP2 = "UVAR" then type = COMP3 else if COMP2 = "ELEMENTARY RECORD" then type = COMP4 else E119; if type = 0 or 1 then A4 else if type = 4 then cen(<data name>, i) else E121, E120)}

6 {if imptflag then ('SYSTEM LABEL', impstlab)↓IS; impstflag = 'false'; if ifcntr > 0 then A5 (as in the IF statement)}

7 {if not impstflag then impstflag = 'true'; impstlab = slab; slab = slab+1; ('TEST SIZE ERROR',impstlab)↓IS}

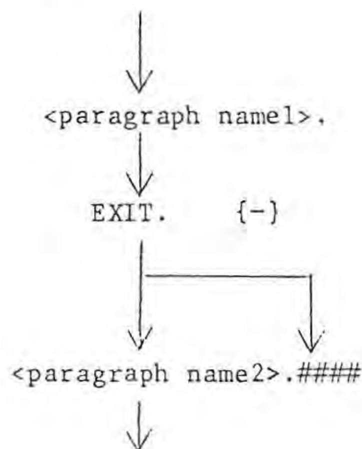
CLOSE



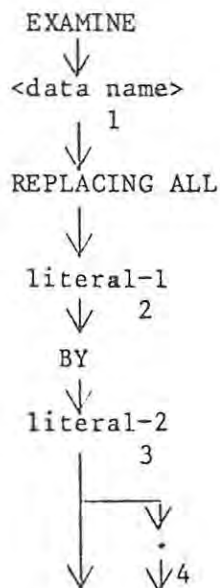
1 {γ(ST,<file name>, if COMP2 # "FILE" then E110 else if COMP4 = "CARD-READER" or "PRINTER" then ('CLOSE SLOW PERIPHERAL', COMP6, COMP5, COMP4)↓IS else i = ati; ('DEFINE CONSTANT',1, ati,3, 262/#1000,0,0)↓IS; ati = ati+1; j = ati; ∀ FDSTACK:(y↑FDSTACK; if COMP1(y)=3 then block = COMP2(y));('DEFINE CONSTANT', 1, ati, 3, 262/0,0, block)↓IS; ati= ati+1; ('DEFINE ADDRESS CONSTANT',1, ati,1,COMP8)↓IS; ati = ati+1; ('DEFINE STORE',1,ati,1)↓IS; ati = ati+1; ('CLOSE DISC FILE',COMP5,COMP6,COMP7,COMP8,i,j)↓IS,E111)}

2 {A6(as in the ADD statement)}

EXIT



EXAMINE



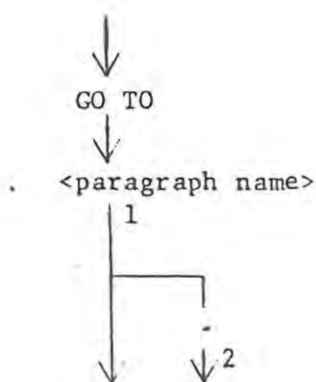
1 {δ(ST,<data name>, if COMP2 = "UVAR" then (if COMP5#0 then sar (<data name>,i) else sav(<data name>,i)) else if COMP2 = "ELEMENTARY RECORD" then saer(<data name>, i) else E116,E117)}

2 {if length(<literal-1>)>1 then E118}

3 {if length (<literal-2>)>1 then E118 else ('EXAMINE',i,<literal-1>,<literal-2>)↓IS}

4 {A6 (as in the ADD statement)}

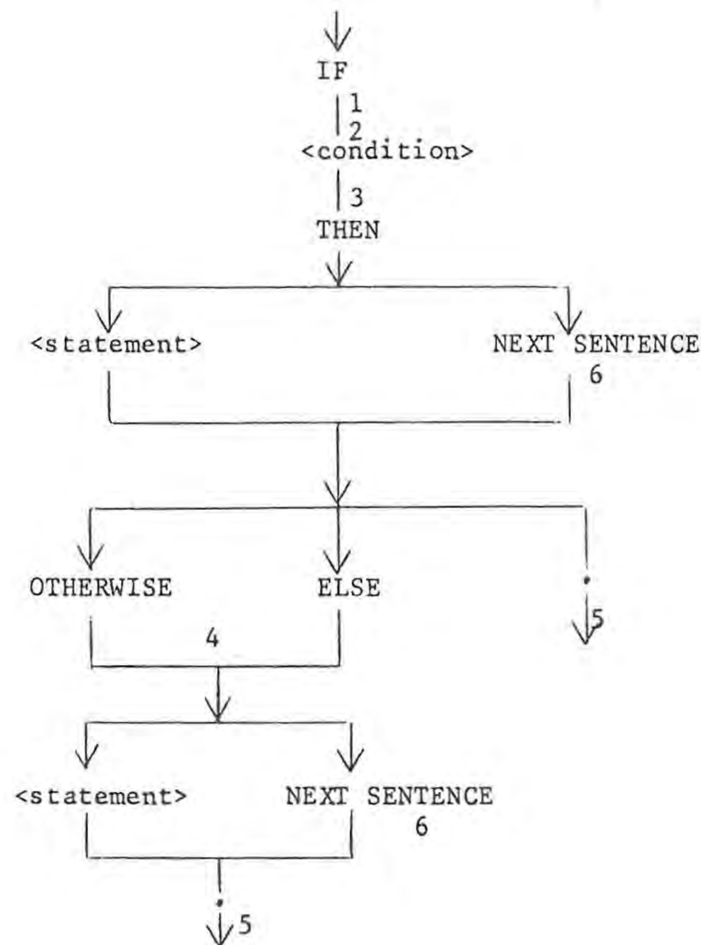
GO TO



1 {δ (ST,<paragraph name>, if COMP2 # "PARAGRAPH" then E164 else ('GO TO',COMP4)↓IS,(<paragraph name>, "PARAGRAPH",1, pti) ↓ ST;('GO TO', pti)↓IS; pti = pti+1}

2 {A6 (as in the ADD statement)}

IF



1. {<u>if</u> impstflag <u>then</u> E144; <u>if</u> ifcntr ⩽ 7 <u>then</u> ifcntr = ifcntr+1 <u>else</u> E140}

2 {condition analyzing routine}

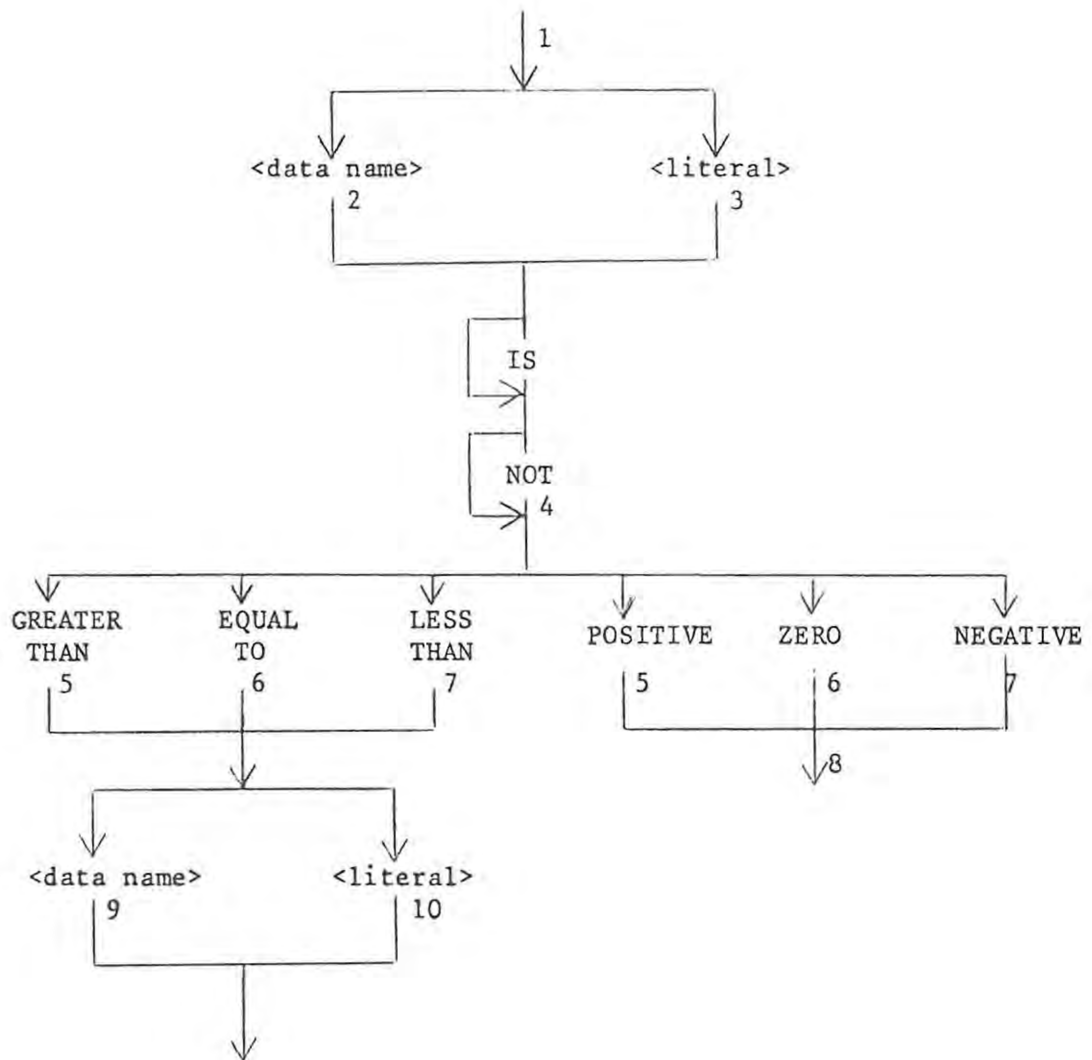
3 {('IF CONDITION FALSE',slab)↓IS; IFSTACK(ifcntr)=(1,slab); slab = slab+1; <u>if</u> ifcntr=1 <u>then</u> endif = slab; slab = slab+1}

4 {<u>if</u> COMP1(IFSTACK(ifcntr))=1 <u>then</u> ('GO TO SLAB', endif)↓IS; COMP1(IFSTACK (ifcntr))=0; ('SYSTEM LABEL',COMP2(IFSTACK(ifcntr)))↓IS <u>else</u> <u>if</u> ifcntr > 1 <u>then</u> ('GO TO SLAB', endif)↓IS; ifcntr = ifcntr-1; A4 <u>else</u> E141}

5 {∀IFSTACK:(x↑IFSTACK; <u>if</u> COMP1(x)=1 <u>then</u> ('SYSTEM LABEL',COMP2(x))↓IS); ('SYSTEM LABEL',endif)↓IS; ifcntr = 0 }

6 {('GO TO SLAB', endif)↓IS}

CONDITIONS



1 {literal=0; not=0}

2 {δ(ST,<data name>, if COMP2 = "UVAR" then (shift = COMP4; if COMP3 = 0 or 2 then type = 1 else if COMP3 = 1 or 3 then type = 0 else type = -1; if COMP5#0 then sar(<data name>,i) else sav(<data name>, i)) else if COMP2 = "ELEMENTARY RECORD" then (shift = COMP8; saer (<data name>,i); if COMP4 = 0 or 2 then type = 1 else if COMP4 = 1 or 3 then type = 0 else type = -1) else E139, E139)}

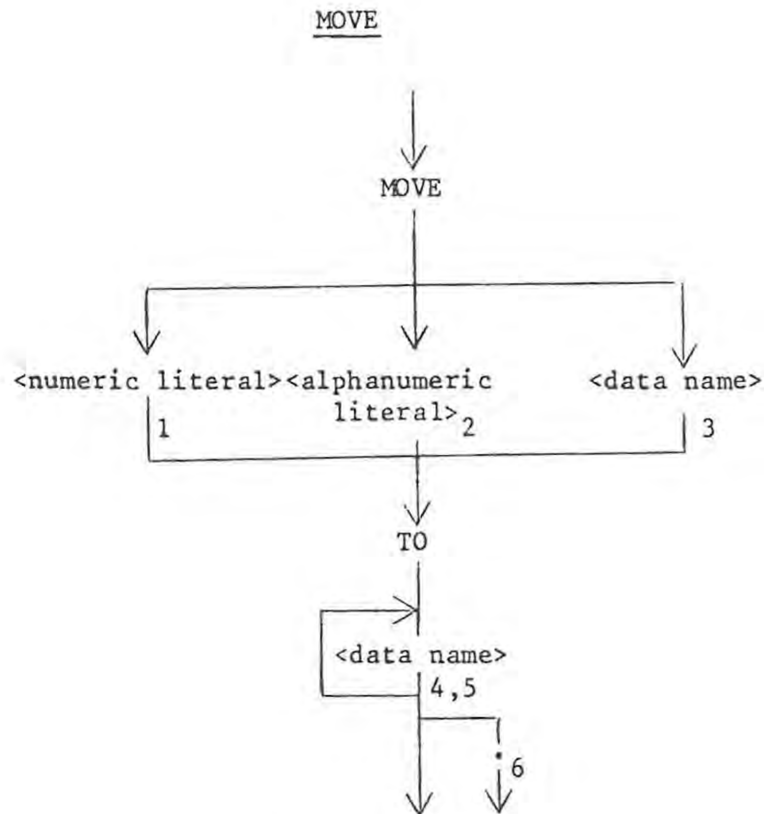
3 {literal = 1; if <literal>= 'NUMERIC' then inlt(<literal>, shift, length, sign, j); if sign = 0 then type = 1 else type = 0 else ianl(<literal>, j); type = -1 }

4 {not = 1}

5 {condition = 1}

```
6   {condition = 0}

7   {condition = -1}

8   {if literal#0 then E148 else gsc(condition,not,i)}

9   {δ(ST,<data name>, if COMP2 = "UVAR" then (if COMP3<4 and type = 0
    or 1 then (if COMP3 = 0 or 2 then sign = 0 else sign = 1; if COMP5#0
    then sar(<data name>,j) else sav(<data name>,j);nrc (condition,not,type,i,
    shift,sign,j, COMP4)) else if COMP3>3 and type = -1 then (if COMP5#0 then
    sar(<data name>,j) else sav(<data name>,j); arc(condition,not,i,j)) else
    E149) else if COMP2 = "ELEMENTARY RECORD" then (if COMP4<0 and type = 0
    or 1 then (saer(<data name>,j); if COMP4 = 0 or 2 then sign = 0 else
    sign = 1; nrc(condition,not,type,i,shift,sign,j,COMP8)) else if COMP4>
    3 and type = -1 then arc (condition, not,i,j) else E149) else E139,E139)}

10  {if literal#0 then E148 else if type = 0 or 1 and <literal> = 'NUMERIC'
    then inlt(<literal>, lshift,length,sign,j); nrc(condition, not,type,i,
    shift,sign,j,lshift) else if type = -1 and <literal> = 'ALPHANUMERIC'
    then ianl(<literal>,j);arc(condition,not,i,j)else E149}
```

MOVE



1 {inlt(<numeric literal>,shift1,length1,sign,i); if sign = 0 then type1 = 0 else type1 = 1}

2 {ialt(<alphanumeric literal>,i ;type1 = 5}

3 {δ(ST,<data name>, if COMP2 = "UVAR" then type1 = COMP3; shift1 = COMP4; length1 = COMP9 ; if COMP5#0 then sar(<data name>,i) else sav(<data name>,i) else if COMP2 = "ELEMENTARY RECORD" then type1 = COMP4; shift1 = COMP8; length1 = COMP7; saer(<data name>,i) else if COMP2 = "RECORD" then type1 = 8; shift1 = 0; length1 = COMP5; sarc(<data name>,i) else E134, E135)}

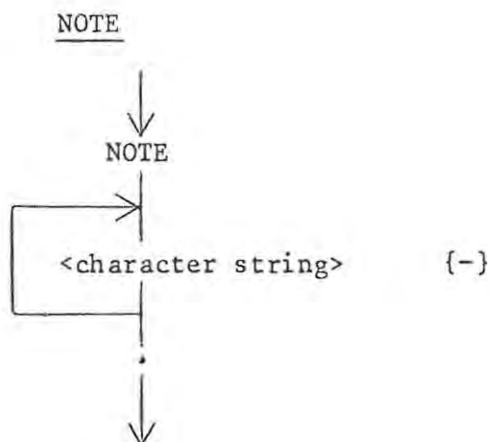
4 {δ(ST,<data name>, if COMP2 = "UVAR" then type2 = COMP3; shift2 = COMP4; length2 = COMP9; if COMP5#0 then sar(<data name>,j) else sav (<data name>,j) else if COMP2 = "ELEMENTARY RECORD" then type2 = COMP4; shift2 = COMP8; length2 = COMP7; saer(<data name>,j) else if COMP2 = "RECORD" then type2 = 8; shift2 = 0; length2 = COMP5, sarc(<data name>,j) else E134,E135)}

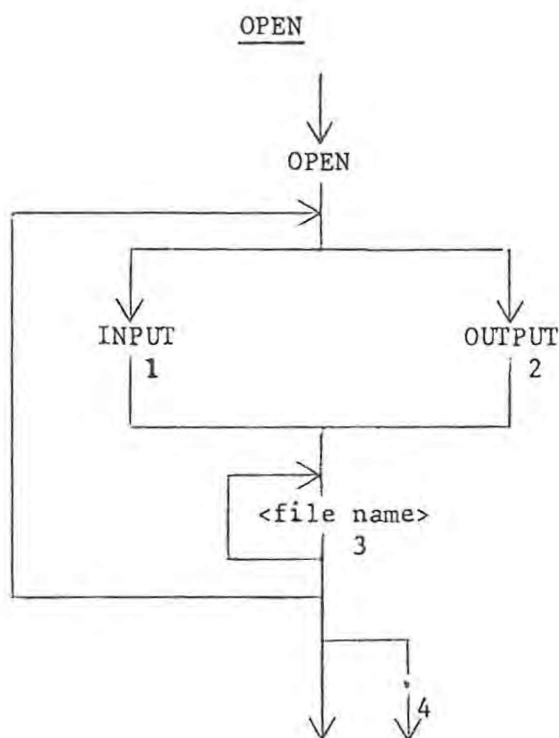
5 {if (type1 = 0 or 1) and (type2 = 0 or 1) then mnn(type1, length1,shift1, i,type2,length2,shift2,j) else if type1 = 5 and (type2 = 0 or 1) then man(length1,i,length2,j) else if (type1 = 4 or 5 or 6 or 7 or 8) and (type2 = 4 or 5 or 6 or 7 or 8) then maa(type1,length1,i,type2,length2,j,<data name>)

else if (type1 = 0 or 1) and type2 = 4 then mne (type1, length1, shift1,i,

length2, shift2,j,<data name>) else E158 }

6 {A6(as in the ADD statement)}





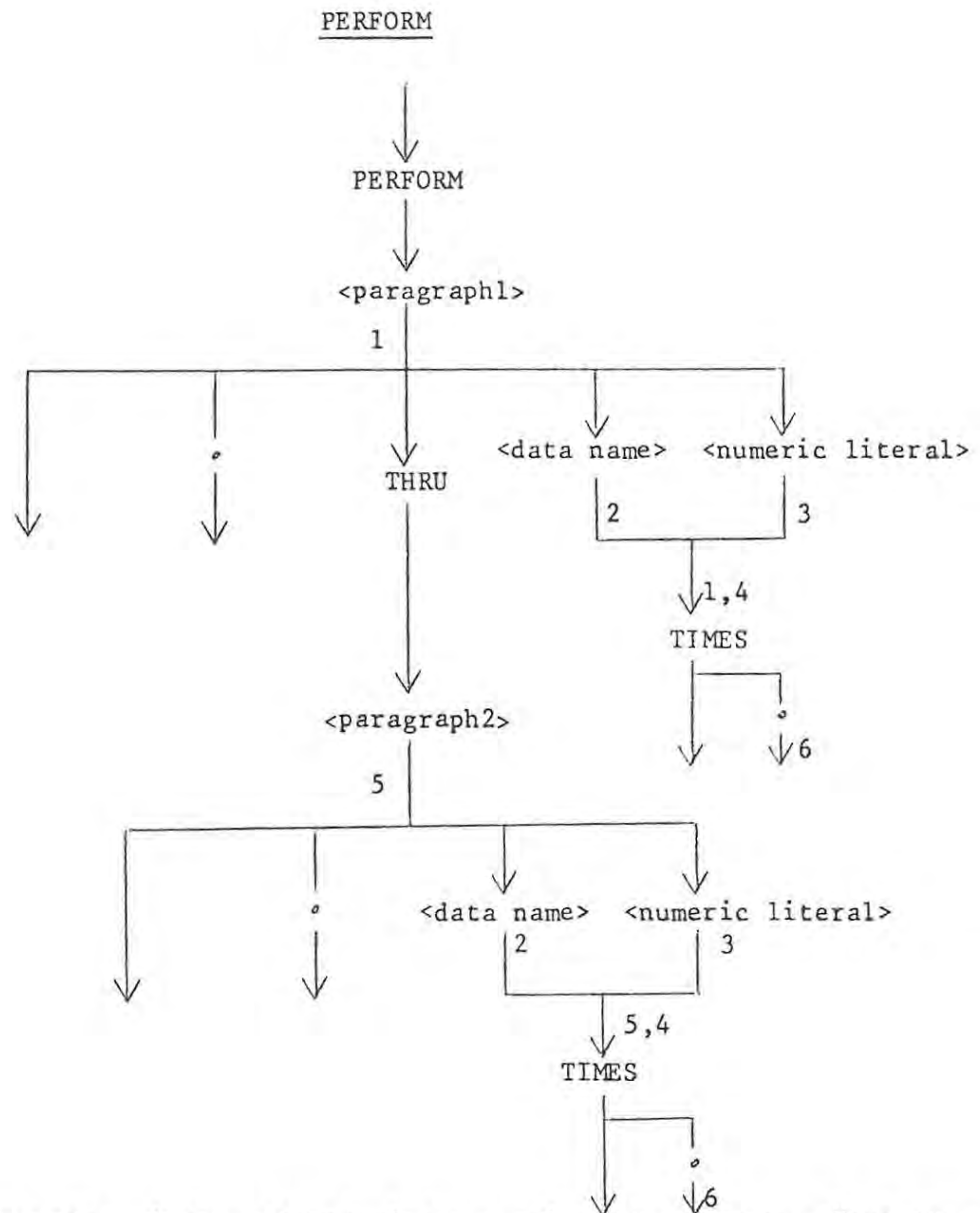
1 {x = "INPUT"}

2 {x = "OUTPUT"}

3 {λ(ST,<file name>, if COMP2≠"FILE" then E112 else if COMP4 = "CARD-READER" and x = "INPUT" then ('OPEN INPUT', COMP5,COMP6)↓IS else if COMP4 = "PRINTER" and x = "OUTPUT" then ('OPEN OUTPUT',COMP5,COMP6)↓IS else if COMP4 = "EDS" then (if x = "INPUT" then iica(<file name>,i); ('OPEN INPUT DISC',COMP5,COMP6,i,COMP7)↓IS else ioca (<file name>,i);('OPEN OUTPUT DISC', COMP5, COMP6,i,COMP7)↓IS) else E150, E113)}

4 {A6(as in the ADD statement)}

PERFORM

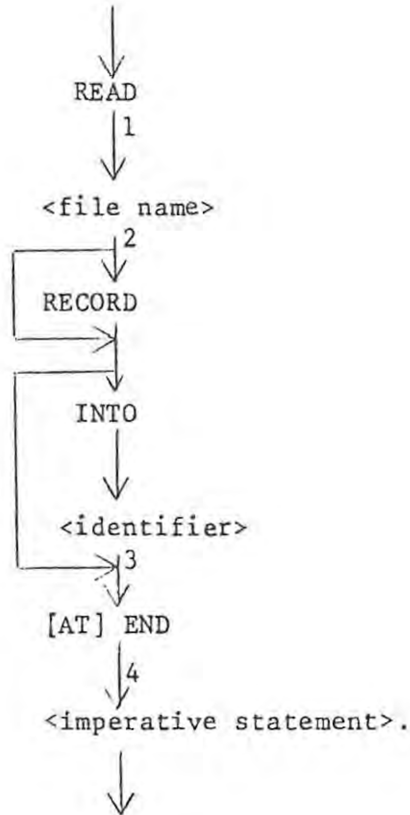



```
1   {𝒮(ST,<paragraph1>, index = COMP4, index = pti ; (<paragraph1>, "PARAGRAPH",1,
    pti)↓ST; pti = pti+1);('PERFORM',pi,index)↓IS;(<paragraph1>,<paragraph1>) ↓
    PRL; pi = pi+1 }

2   {𝒮(ST,<data name>, if COMP2 = "UVAR" and COMP3=0 and COMP4 = 0 then (if
    COMP5#0 then sar(<data name>,i) else sav (<data name>,i)) else if COMP2 =
    "ELEMENTARY RECORD" and COMP4 = 0 and COMP8 = 0 then saer (<data name>,i)
    else E143, E143); ('DEFINE STORE',1, ati,1)↓IS;j = ati;ati = ati+1;
    ('CREATE NO OF TIMES FOR PERFORM',i,j)↓IS }

3   { inlt(<numeric literal>,shift,length,sign,i);if sign#0 or shift#0 then
    E157 else ('DEFINE STORE',1,ati,1)↓IS;j = ati;ati = ati+1;('CREATE NO OF
    TIMES FOR PERFORM',i,j)↓IS }

4   {('PERFORM LOOP',j)↓IS}
```

5 {flag = 0; λ(ST,<paragraph1>, index = COMP4, index = pti ; (<paragraph1>, "PARAGRAPH",1,pti)↓ST; flag = 1; pti=pti+1); λ(ST,<paragraph2>, if flag = 1 then E168, if flag#1 then E169 else (<paragraph2>, "PARAGRAPH",1, pti)↓ST; pti = pti+1); ('PERFORM',pi,index)↓IS;(<paragraph1>,<paragraph2>)↓PRL; pi = pi+1}

6 {A6 (as in the ADD statement)}

READ

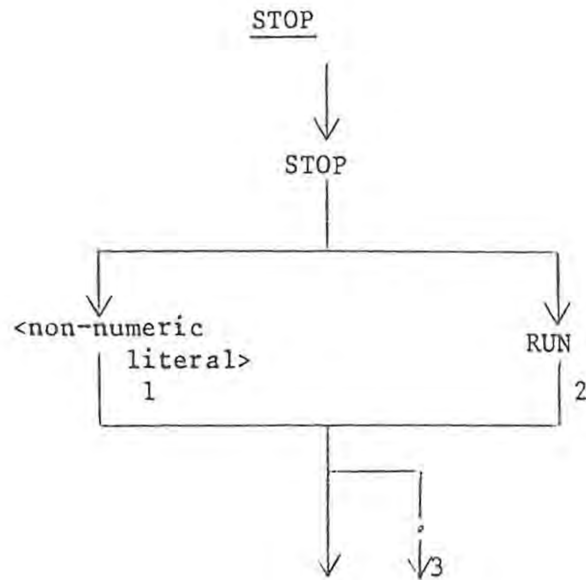


1 {if impstflag then E145}

2 {δ(ST,<file name>, if COMP2#"FILE" then E129 else ('CHECK IF FILE OPEN FOR INPUT',COMP6)↓IS; ∀ FDSTACK:(y↑FDSTACK; if COMP1(y) = 3 then count3 = COMP2(y) else if COMP1(y)=1 then stadd = COMP2(y) else if COMP1(y) = 9 then count = COMP2(y));i=ati;('DEFINE CONSTANT',1,ati,1,stadd)↓IS; ati=ati+1; if COMP4 = "CARD-READER" then j=ati; ('DEFINE CONSTANT',1,ati,4,259/0,0,count, stadd)↓IS; ati=ati+1; ('READ FROM CR',COMP5,j)↓IS else if COMP4 = "EDS" then j = ati; count3 = count3/4;('DEFINE CONSTANT',1,ati,3,262/0,0,count3) ↓IS; ati=ati+1; ('DEFINE ADDRESS CONSTANT',1,ati,1,COMP8)↓IS;ati=ati+1; ('DEFINE CONSTANT',1,ati,1,0)↓IS;ati=ati+1;('DISC FILE READ',j,i,count,COMP5, COMP7,COMP8)↓IS; else E154,E131)}

3 {type1 = 8; shift1 = 0; length1 = count; MOVE A4 and A5 }

4 {impstflag = 'true'; impstlab = slab; slab = slab+1;('TEST SIZE ERROR', impstlab)↓IS}

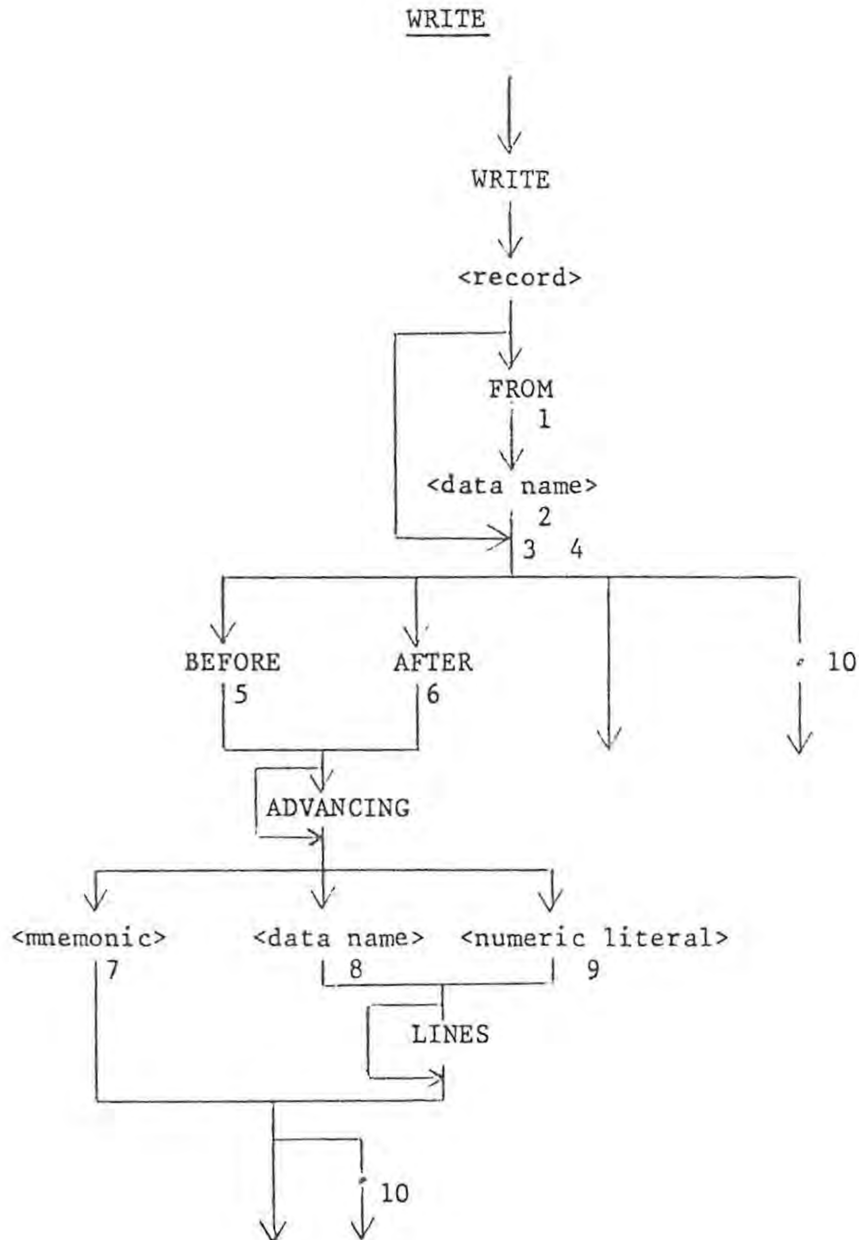


1 {if length(<non-numeric literal>)>40 then E146 else ianl(<non-numeric literal>,i);('SUSPEND',i)↓IS}

2 {('STOP')↓IS}

3 {A6(as in the ADD statement)}

WRITE



1 {MOVE A4}

2 {MOVE A3;A5}

3 {S(ST,<record>, if COMP2 = "RECORD" then (if COMP3 = 0 then E124 else x = COMP3; count = COMP5; stadd = COMP6-1) else if COMP2 = "ELEMENTARY RECORD" then (if COMP3 = 0 then E124 else x = COMP3; count = COMP7; stadd = COMP10-1)else E124,E126);('CHECK IF FILE OPEN FOR OUTPUT',COMP6(x))↓IS; staddl = ati;('DEFINE CONSTANT',l,ati,l,stadd+l)↓IS; ati = ati+l; if COMP4(x) = "PRINTER" then (ica = ati;('DEFINE CONSTANT',l,ati,4,258/0,0, count+l,stadd)↓IS; ati = ati+l; discflag = 0; stadd2 = ati; ('DEFINE CONSTANT', l,ati,l,stadd)↓IS; ati = ati+l) else if COMP4(x) = "EDS" then (∀FDSTACK :

(y↑FDSTACK; if COMP1(y) = 3 then buclength = COMP2(y)); ica = ati; ('DEFINE CONSTANT',1,ati,3,262/0,0,buclength)↓IS; ati = ati+1; ('DEFINE ADDRESS CONSTANT',1,ati,1,COMP8)↓IS; ati = ati+1; ('DEFINE CONSTANT' ,1,ati,1,0)↓IS; ati = ati+1; if count>buclength-12 then E160; discflag = 1) else E151}

4 {if discflag = 1 then ('DISC FILE WRITE',ica,staddl,count, COMP5(x), COMP7(x), COMP8(x))↓IS else ('INSERT CONTROL CHAR',#40, stadd2)↓IS; ('WRITE TO LP', COMP5(x),ica)↓IS}

5 {if discflag#0 then E152 else advanceflag = 'before'}

6 {if discflag#0 then E152 else advanceflag = 'after'}

7 {ℵ(ST,<mnemonic>, if COMP2#"MNEMONIC" then E132 else if COMP3 = 0 or> 7 then E132 else ('INSERT CONTROL CHAR',#50 + COMP3, stadd2)↓IS;('WRITE TO LP',COMP5(x),ica)↓IS, E133)}

8 {ℵ(ST,<data name>, if COMP2 = "UVAR" and COMP3 = 0 then (if COMP5#0 then sar(<data name>,i) else sav(<data name>,i) else if COMP2 = "ELEMENTARY RECORD" and COMP4 = 0 then saer (<data name>,i) else E132 ; if advance-flag = 'before' then ('INSERT CONTROL CHAR',#40, stadd2)↓ IS;('WRITE TO LP',COMP5(x),ica)↓IS;('CDB',4,i)↓IS;('INSERT CONTROL CHAR',1,stadd2)↓IS; ('INSERT CONSTANT',1,ica)↓IS;('WRITE BLANKS TO LP-IDENTIFIER', COMP5(x), ica)↓IS else('INSERT CONTROL CHAR',1,stadd2)↓IS;('INSERT CONSTANT',1,ica) ↓IS; ('CDB',4,i)↓IS;('WRITE BLANKS TO LP-IDENTIFIER',COMP5(x),ica)↓IS; ('INSERT CONSTANT',count,ica)↓IS;('INSERT CONTROL CHAR',#40,stadd2)↓IS; ('WRITE TO LP',COMP5(x),ica)↓IS, E133)}

9 {if literal = 0 then E153; if advanceflag = 'before' then ('INSERT CONTROL CHAR',#40, stadd2)↓IS; ('WRITE TO LP',COMP5(x),ica)↓IS;('INSERT CONSTANT', 1,ica)↓IS;('INSERT CONTROL CHAR',1, stadd2)↓IS;('WRITE BLANK LINES',COMP5(x), ica,literal)↓IS else literal = literal-1; if literal>0 then (('INSERT CONSTANT',1,ica)↓IS;('INSERT CONTROL CHAR',1,stadd2)↓IS;('WRITE BLANK LINES', COMP5(x), ica, literal)↓IS;('INSERT CONSTANT',count,ica)↓IS);('INSERT CONTROL CHAR',#41, stadd2)↓IS;('WRITE TO LP',COMP5(x),ica)↓IS}

10 {A6 (as in the ADD statement)}

The global system variables and routines used in the formal specification have the functions given below

ati	address table index
pti	paragraph table index
slab	system label table index
impstflag	flag indicating whether an imperative statement is operational
impstlab	the system label to be inserted into the output stream on encountering the end of an imperative statement
ifcntr	the level of IF statement nesting
pi	PERFORM statement index
IS	instruction stack (or pseudo-code output stream)
TIS	temporary instruction stack (see ADD statement)
PNS	paragraph names stack
PRL	paragraph range list
sarc, sav, saer	routines to create the length/start addresses of records, variables and elementary records respectively.
sar	routine to generate the pseudo-codes for the creation of the length and start address of a subscripted variable
ianl, inlt	routines to insert alphanumeric and numeric constants respectively into the data area together with the length and start addresses of the constants
lxan	the lexical analyzer

The routines listed below generate pseudo-codes to perform the indicated functions

cbd	convert a binary value to decimal
cen	convert a number to an edited numeric field
iica	initialize a disc file open for input control area
ioca	initialize a disc file open for output control area
gsc	generate a sign condition
nrc	generate a numeric relational condition
arc	generate an alphanumeric relational condition
mnn	move a numeric field to a numeric field
man	move an alphanumeric field to a numeric field
maa	move an alphanumeric field to an alphanumeric field
mne	move a numeric field to an edited numeric field

APPENDIX 8

THE DECIMAL TO BINARY CONVERSION CODE SKELETONS

1) Code skeleton to convert an unsigned decimal number to a binary number.

Parameter 1 : the first of 2 adjacent accumulators into which the binary number is to be inserted

Parameter 2 : the address table element number containing the address of the location containing the length and start address of the decimal number.

```
LDX     1     !%2,0
STOZ          %1,0
STOZ          %1+1,0
CBD     %1    0,1
BCHX    1     *-1
```

2) Code skeleton to convert a signed decimal number to a binary number. The parameters are the same as those described above.

```
STOZ          %1,0
LDX     2     %2,0
LDN     1      1,0
LDCH  %1+1     0,2
TXL   %1+1    45,0
BCS           *+3
STOZ           1,0
SBN   %1+1    32,0
BCHX    2     *+1
CBD    %1      0,2
BCHX    2     *-1
BNZ     1     *+3
NGXC  %1+1    %1+1,0
NGX    %1     %1,0
```

APPENDIX 9

THE ALGORITHM AND CODE SKELETONS FOR THE CONVERSION OF BINARY NUMBERS TO DECIMAL NUMBERS

Assume that :

i) the number in accumulators 6 and 7 contains the binary equivalent of the decimal number represented in COBOL picture clause notation as 9(I)V9(J)

ii) the decimal field in which the answer is to appear has K places to the left and L places to the right of the decimal point.

The algorithm is as follows.

```
X = MIN(I,K)+MIN(J,L)       (I.E. X CONTAINS THE LENGTH IN DIGITS OF THE
                             REQUIRED PART OF THE BINARY NUMBER)
SKELETON(I(ADDRESS TABLE ELEMENT NUMBER CONTAINING THE
          LENGTH/START ADDRESS OF THE RESULT FIELD)
IF X <= 6
   THEN
      BEGIN
      IF K <= I
         THEN
            BEGIN
            IF L <= J
               THEN
                  BEGIN
                  IF K+J > J
                     THEN
                        BEGIN
                        IF J-L > 6
                           THEN
                              BEGIN
                              N[1] = K+J-6
                              N[5] = 6
                              END
                           ELSE
                              BEGIN
                              N[1] = K+L
                              N[5] = J-L
                              END
                        SKELETON(G(N[5]))
                        SKELETON(A(N[1]))
                        SKELETON(D)
                        END
                     ELSE
                        BEGIN
                        N[1] = K+J
                        SKELETON(A(N[1]))
                        SKELETON(D)
```

```
                            END
                        END
                    ELSE
                        BEGIN
                        N[1] = K+J
                        N[2] = K+J
                        SKELETON(A(N[1]))
                        SKELETON(C(N[2]))
                        END
                END
            ELSE
                BEGIN
                IF L <= J
                    THEN
                        BEGIN
                        IF I+J > 6
                            THEN
                                BEGIN
                                N[3] = K-I
                                IF J-L > 6
                                    THEN
                                        BEGIN
                                        N[1] = I+J-6
                                        N[5] = 6
                                        END
                                    ELSE
                                        BEGIN
                                        N[1] = I+L
                                        N[5] = J-L
                                        END
                                SKELETON(G(N[5]))
                                SKELETON(A(N[1]))
                                SKELETON(B(N[3]))
                                SKELETON(D)
                                END
                            ELSE
                                BEGIN
                                N[1] = I+J
                                N[3] = K-I
                                SKELETON(A(N[1]))
                                SKELETON(B(N[3]))
                                SKELETON(D)
                                END
                        END
                    ELSE
                        BEGIN
                        N[1] = I+J
                        N[2] = I+J
                        N[3] = K-I
                        SKELETON(A(N[1]))
                        SKELETON(B(N[3]))
                        SKELETON(C(N[2]))
                        END
                END
    END
ELSE(I.E. 6< X <=13)
    BEGIN
    N[4] = K+J-6
    IF K <= I
        THEN
            BEGIN
```

```
          SKELETON(E(N[4]))
          SKELETON(F)
          SKELETON(D)
          END
      ELSE
          BEGIN
          IF L <= J
              THEN
                  BEGIN
                  IF N[4] > 6
                      THEN
                          BEGIN
                          N[6] = N[4]-6
                          N[4] = 6
                          SKELETON(H(N[6]))
                          SKELETON(E(N[4]))
                          SKELETON(F)
                          SKELETON(D)
                          END
                      ELSE
                          BEGIN
                          SKELETON(E(N[4]))
                          SKELETON(F)
                          SKELETON(D)
                          END
                  END
              ELSE
                  BEGIN
                  SKELETON(E(N[4]))
                  SKELETON(F)
                  SKELETON(D)
                  END
          END
  END
```

The code skeletons used by the algorithm are the following

```
A       LDX   5    7,0
        LDX   6    %1+45,0
        DVD   5    6

B       LDCT  4    %1,0
        LDN   5    0,0
        DCH   5    0,3
        BCHX  3    *+1
        BUX   4    *-2
```

```
C       LDCT  4   %1,0
        CBD   6   0,3
        BCHX  3   *+1
        BUX   4   *-2

D       CBD   6   0,3
        BCHX  3   *-1

E       DVD   6   %1+45,0
        LDX   0   7,0
        LDX   7   %1+45,0
        DVD   6   7,0

F       LDX   6   0,0
        DVD   6   50,0
        LDX   0   7,0
        LDX   7   50,0
        DVD   6   7,0
        LDX   6   0,0

G       DVD   6   %1+45,0

H       DVS   5   %1+45,0
        SRC   56  24,0
        DVD   6   %1+45,0
        LDX   6   5,0

I       LDX   3   !%1,0
```