A SUBSET OF ALGOL-60, IMPLEMENTATION

DISSERTATION SUBMITTED TO JAWAHARLAL NEHRU UNIVERSITY
IN PARTIAL FULFILMENT FOR THE DEGREE OF
MASTER OF PHILOSOPHY

XIII + 124 P.

LACHHMAN DASS

SCHOOL OF COMPUTER & SYSTEMS SCIENCES
JAWAHARLAL NEHRU UNIVERSITY
NEW DELHI-110067

1981

C E R T I F I C A T E

SCHOOL OF COMPUTER AND SYSTEMS SCIENCES
JAWAHARLAL NEHRU UNIVERSITY
NEW DELHI - 110 067

The research work embodied in this dissertation has been carried out in the School of Computer and Systems Sciences and is original and has not been submitted so far in part or full for any other degree or diploma of any Univ.

Lachhman Dass

LACHHMAN DASS
Student

N. P. Mukherjee

PROF. N.P.MUKHERJEE
N. P. MUKHERJEE
Dean,
School of Computer and Systems Sciences
Jawaharlal Nehru University
New Delhi-110067

P. C. Saxena

DR. P.C. SAXENA
Supervisor

A C K N O W L E D G E M E N T S

It is indeed a pleasure to acknowledge the guidance of Dr. P.C. Saxena. His scholarly guidance, enthusiastic encouragement and constructive criticism enabled me to learn much out of this work.

For making computer and other facilities available, I am grateful to Prof. N.P. Mukherjee, the Incharge, School of Computer and Systems Sciences.

I am grateful to all my friends for their continuing held and useful comments and especially to Mr. A.K. Bansal for his time to time discussions.

Finally, I am also indebted to C.S.I.R. for its financial assistance.

Mr. K. Chand deserves special mention for his prompt and efficient typing

New Delhi
31.12.1981

Lachhman Dass
LACHHMAN DASS

_C_O_N_T_E_N_T_S_

ACKNOWLEDGEMENTS	ii
PREFACE	iv
1. COMPILER STRUCTURE	2
1.1 LEXICAL ANALYSIS	4
1.2 SYNTAX ANALYSIS	6
1.2.1 TOP-DOWN PARSING	7
1.2.2 BOTTOM-UP PARSING	7
1.2.2 OTHER GRAMMARS AND PARSING METHODS	14
1.3 SEMANTIC ANALYSIS	14
1.4 CODE GENERATION	17
1.5 OPTIMIZATION	19
2. AN ALGOL-60 SUBSET	23
2.1 STRUCTURE OF THE LANGUAGE	25
2.2 BASIC SYMBOLS, IDENTIFIERS, NUMBER AND BASIC CONCEPTS	26
2.2.1 LETTERS	26
2.2.2 DIGITS	27
2.2.3 DELIMITERS	27
2.2.4 IDENTIFIERS	28
2.2.5 NUMBERS	29
2.2.6 VALUES AND TYPES	30

2.3 EXPRESSIONS 31
2.3.1 VARIABLES 31
2.3.2 SYNTAX 31
2.3.3 ARITHMETIC EXPRESSIONS 32
2.3.4 BOOLEAN EXPRESSION 35
2.4 STATEMENTS 36
2.4.1 COMPOUND STATEMENTS AND BLOCKS 36
2.4.2 ASSIGNMENT STATEMENTS 39
2.4.3 GO TO STATEMENTS 41
2.4.4 DUMMY STATEMENTS 41
2.4.5 CONDITIONAL STATEMENTS 42
2.4.6 PROCEDURE STATEMENTS 45
2.5 DECLARATIONS 46
2.5.1 SYNTAX 47
2.5.2 TYPE DECLARATIONS 47
2.5.3 PROCEDURES DECLARATIONS 48

3. DATA STRUCTURE 51
3.1 SYMBOL TABLE 51
3.2 LITERAL TABLE 53
3.3 SPECIAL CHARACTER TABLE 54
3.4 KEYWORD TABLE 55
3.5 BLOCK LIST 55
3.6 TEMPORARY LABEL TABLE 56

3.7 ERROR MESSAGE TABLE 56

3.8 ERROR MESSAGE FILE 57

3.9 INTERMEDIATE CODE 58

4. LOGIC AND ALGORITHM OF THE COMPILER 61

4.1 LEXICAL PHASE 62

4.2 SYNTAX PHASE 64

4.3 SEMANTICS AND INTERMEDIATE CODE GENERATOR PHASE 80

5. PROGRAMMING DETAILS 89

5.1 MAIN PROGRAM 89

5.2 STATE 90

5.3 GETSYM 91

5.4 EXPR 92

5.5 IO 93

5.6 GENCOD 93

5.7 BLCKST 94

5.8 CLBLCK 94

5.9 SEARCH 95

5.10 SEARCHB 95

5.11 SRDMBL 95

5.12 INSERT 96

5.13 OPPROC 96

5.14 PROCID 96

5.15 OPBLCK 97

5.16 DCLREN 97

5.17 NEXTEN 97

REFERENCES 99

APPENDIX PCOMPILER LISTING 100

To my parents

PREFACE

The efficient use of Computer in various fields requires not only extensive knowledge of the problems to be solved but also very specialized computer know-how. A knowledge of any one high level programming language limits the problem that can be solved to fairly simple one from the computer specialists stand point.

Algol language designed for scientific calculation was one of the earliest and most influential high-level language. The most widely used version of the language is known as Algol-60. Algol was the first language which introduced the concept of block structured programmes. And it is due to its expressibility and readability it became very popular in the educational field for designing and teaching art of computer programming. In fact, design of number of languages like Pascal, Algol-68, PL/I etc. was heavily dependent on the design philosophy of Algol.

Among the higher level languages our interest is to study Algol-60. Brief discription of the Algol language is given herewith.

ALGOL SUMMARY

DATA

TYPES: integer, real, Boolean

STRUCTURES: simple variables, arrays

DATA OPERATIONS:

+,-,*,/,exponentiation ( ),

relational (, , =, , , ≠),

logical (=, , v, , -)

CONTROL OPERATIONS:

goto

if ⟨condition⟩ then ⟨statement⟩

if ⟨condition⟩ then ⟨statement⟩ else ⟨statement⟩

for ⟨var⟩ :=n1 step n2 until n3 do

for ⟨var⟩ :=n1 step n2 until while ⟨condition⟩do

procedure call by use of name

return

halt

PROGGRAM STRUCTURES:

simple statements, compound statements, procedures

SYSTEM ENVIRONMENT:

usually batch
I/O varies among implementation
library of standard functions
non-Algol procedures sometimes permitted
run-time packages support I/O and space allocation

Data types and expressions in Algol are integers, reals and Boolean. Arrays are of more general type and they can have negative index sets.

A significant feature of Algol is its block structure. Blocks may be nested. Variable names defined within a particular block have the scope of definition limited to the block in which it appears and any inner blocks. Information about a variable declared in a block is completely inaccessible outside the block. An example of block structure is as follows:

```
Begin    real a,b;integer c;
         a:=7.3;  b:=2.3;  c:=2;
         begin integer a,c;real b;
               a:=7;  b:=3.5;  c:=a+5;
         end  (a=7, b=3.5, c=12)
end      (a=7.3, b=2.3, c=2)
```

Figure 1.1. Algol Block Structure

Upon entering a block, all the variables declared have storage allocated to them, and upon exit from a block, the storage is deallocated.

Block structure provides a great deal of expressive power. In a conditional expression, for example, where

the general form is

if ⟨condition⟩ then ⟨statement⟩ else ⟨statement⟩

the statement to be executed conditionally can be a compound statement. Thus a statement which may be a compound statement can be executed conditionally by means of this single statement.

Algol permits more powerful use of conditional statements also such as:

A:=(if B then C else D)

This is an assignment statement, which assigns to A the value of C or D according as B is true or false.

Much flexibility is allowed in the use of procedure parameters. The names of routines can be passed, and data can be passed by value or by name. In passing parameters by value, the actual parameters used in the call to the procedure are evaluated and these values are used throughout the execution of the procedure. In passing by name, the names of values are passed and are accessed by the procedure body as needed. This means that the values may be changed by the procedure before they are used.

In the definition of Algol, no user-defined external procedures are allowed. Some implementations of Algol

loosen this restriction, even tothe point of permitting access to procedures initially written in languages other than Algol. Recursion is permitted in Algol. The following is an example of recursive procedure to calculate a factorial:

```
integer procedure factorial (n);
        integer n;
        factorial:=if n<3 then n
                          else n*factorial(n-1);
```

The one area in which Algol, as originally defined, is quite deficient is that of I/O. The language was primarily designed as a way of expressing algorithms, and I/O statements were not made a part of the official language. Each implementation of Algol must include statements that permits I/O, but these tend to vary from system to system.

In this dissertation a compiler for a subset of Algol-60 is described. It is written in PL/I language available on computer EC-1020B. The dissertation has been divided into five chapters.

In the first chapter a survey of compiler structure is given, and formal techniques are explained.

The second chapter lists the subset of Algol-60. This subset is taken for implementation.

The third chapter explains various data structures used in the compiler implementation.

The fourth chapter describes the logic and algorithm of the compiler.

In the fifth chapter programming details of various subroutines is explained.

CHAPTER I

COMPILER STRUCTURE

Compiler translate statements in a programming language into instructions, which, possibly after leading, can be directly executed by the hardware of some computer, not necessarily the one on which the translation has taken place.

A useful conceptualization of a compiler is that it is a program which takes as input a string of characters representing a program in a language and outputs a set of machine instructions with the same meaning as was intended by the statements in the input. A useful model of the structure of a compiler is shown in figure 1.1, which shows the important phases involved during compilation. It has many variations in practice. For instance, not shown are the important functions of generating output listing, and setting up tables and routines for run-time support. Some compilers generate code as the syntax analysis is done and thus do not have an explicit code-generation phase; many do no optimization;and some combine lexical and syntax analyses.

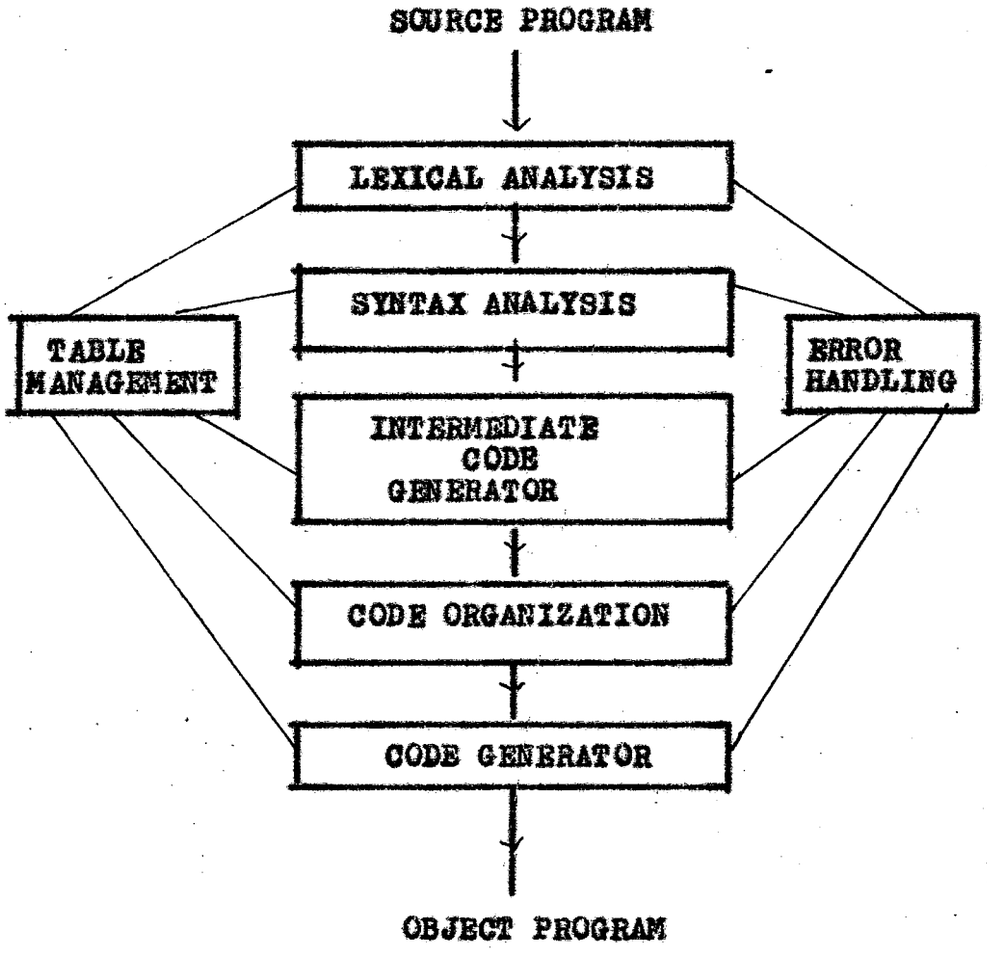


Figure 1.1. Phases of a Compiler

The Technology of Compilers:

Figure 1.1 makes it clear that, there are several major tasks that must be performed in translating statements in a high-level language into machine code that is executable by hardware. Our intention here is to explain the more common ones and the principles on which they are built.

1.1 Lexical Analysis

It separates characters of the source language program into groups that logically belong together; these groups are called tokens. The usual tokens are keywords, such as DO, IF, THEN, identifiers, such as XN or IX, operator symbols such as * or +, and punctuation symbols such as parentheses or commas. The output of the lexical analyzer is a stream of tokens which is passed to the next phase, the syntax analyzer, or parser. The tokens in this stream can be represented by codes which we may regard as integers.

Figure 1.3 illustrate the role of lexical analyzer in transformation of program text. It usually performs several "cleaning functions to remove unneeded blanks, comments, line numbers, end-of-record marks, carriage-return characters, and so forth, that have no meaning to the translator".

Words in computer languages usually belong to one of three classes: (1) identifiers (labels, variables), (2) terminals (operators, reserved words), and (3) literals (numerical constants, strings). The lexical analyzer returns to the parser a code for the token that it found. In the case that the token is an identifier or another token with a value, the value is also passed to the parser. The usual method of providing this information is for the lexical analyzer to call a book-keeping routine which installs the actual value in the symbol table if it is not already there. The lexical analyzer then passes the two components of a token to the parser. The first is a code for the token type (identifier), and the second is the value, a pointer to the place in the symbol table reserved for the specific value found.

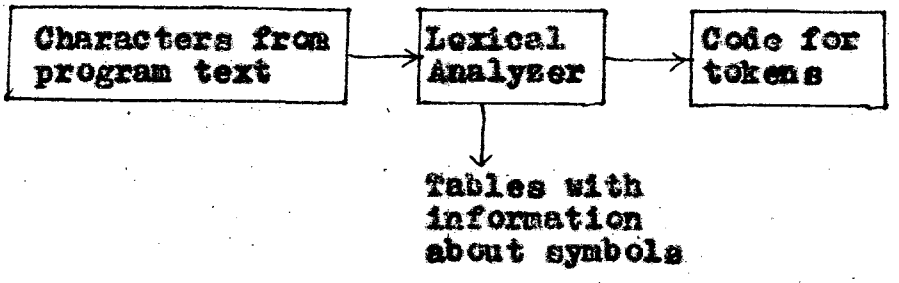


Figure 1.2 Model of lexical analysis

1.2 Syntax Analyser

The function of the syntax analyzer is to recognize the major constructs of the language and to call the appropriate action routines that will generate the intermediate form for these constructs. General model of a syntax analyzer is shown in figure 1.3.

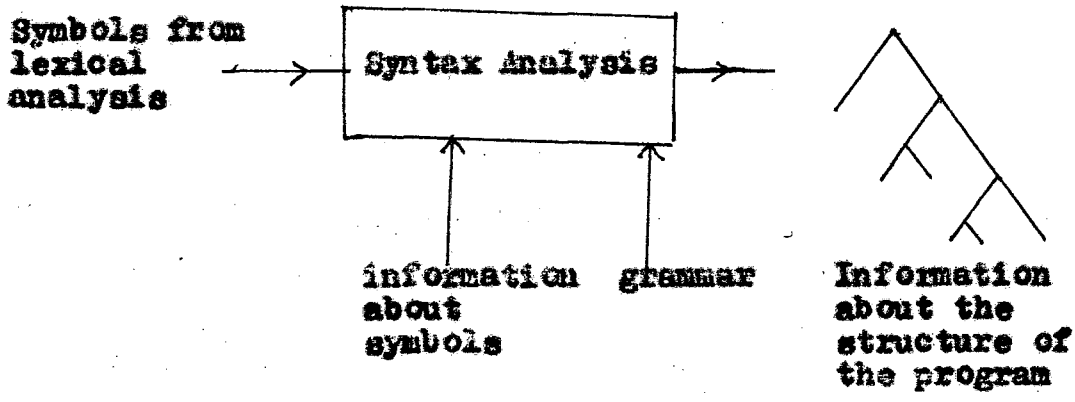


Figure 1.3. General Model of a Parser

Two approaches are common in the design of compiler. In one, the parser does a complete structural analysis of the input stream, and the output is an internal form of representation known as a tree. Control is then passed to the semantic analyzer. In the other, the parser is a master control routine which calls on lexical and semantic

analysis routines as it parses. In this case, the output may be either an internal representation such as a tree or machine language. A comprehensive study of the subject is given in Aho and Ullman (1973), David Gries (1977).

1.2.1 Top-Down Parsing

The top-down parsing algorithm builds the syntax tree, starting with the root node and working down to the sentence. It is described as analysis by synthesis that is, analysis by attempting to generate a program using the rules of a grammar as productions. It checks to see whether or not a group of one or more symbols in the input stream satisfies a particular production rule and predicts that the string satisfies the right-hand side of this rule. If it is wrong, it must back up to the beginning of the string and whether the string satisfies another production rule.

1.2.2 Bottom Up

We will discuss this technique in more detail since it will be used for the parsing in the compiler presented in this dissertation.

Bottom up parsing treats grammar rules as reductions.

Rather than trying to generate the entire program from the initial symbol, a bottom up parser attempts to reduce the entire program to the initial symbol of grammar. The parser checks to see whether or not part of the string satisfies the right-hand side of a production rule. If a reduction can be made, that part of the string is replaced by the left-hand side of the production rule. Parsing then continues with the modified string. If at any point in this process the string cannot satisfy a production rule, backup must occur. To avoid backup, the grammar is usually modified or restricted. Simple precedence grammars and operator precedence grammars are restricted grammars. A grammar is a simple precedence grammar when there is atmost one relationship between any two symbols, and no two production rules have identical right hand parts. Consider the following grammar

⟨expr⟩ ::= ⟨expr⟩ + ⟨term⟩ | ⟨term⟩

⟨term⟩ ::= ⟨term⟩ * ⟨factor⟩ | ⟨factor⟩

⟨factor⟩ ::= (⟨expr⟩) | ⟨number⟩

Figure 1.4. Grammar for bottom-up example

This grammar is not a simple precedence grammar because more than one relationship holds in some cases (for example between + and <term> , # and <expr> , (and <expr>).

This conflict can be removed by adding additional production rules to change the grammar of figure 1.4 to the form shown in figure 1.5. Figure 1.6 shows the precedence relationship for the new grammar.

<s> ::= # <e> #

<e> ::= <expr>

<expr> ::= <expr> + <t>

<t> ::= <term>

<term> ::= <term> * <factor>

<term> ::= <factor>

<factor> ::= (<e>)

<factor> ::= <number>

Figure 1.5. Revised grammar for bottom-up example

	⟨e⟩	⟨expr⟩	⟨t⟩	+	⟨term⟩	⟨factor⟩	*	⟨number⟩	(	)	#
⟨e⟩									≐	≐	
⟨expr⟩									⋗	⋗	
⟨t⟩									⋗	⋗	
+		≐		⋖	⋖		⋖	⋖			
⟨term⟩			⋗			≐			⋗	⋗	
⟨factor⟩			⋗			⋗			⋗	⋗	
*							≐	⋖	⋖		
⟨number⟩			⋗				⋗		⋗	⋗	
(	≐	⋖	⋖		⋖	⋖		⋖	⋖		
)			⋗			⋗			⋗	⋗	
#	≐	⋖	⋖		⋖	⋖		⋖	⋖		

Figure 1.6. Simple precedence relationships

The parsing algorithm uses a stack as follows. Each incoming symbol of the string being parsed is compared with the symbol at the top of the stack. The relationship can be found by checking a precedence table or precedence function based on such a table. If the relationship between the symbol on the top of the stack and the new symbol is $\lessdot$ or $\doteq$, the new symbol is placed on the top of the stack if there is no relationship, an error condition exists. If the relationship is $\gtrdot$, the stack is searched downward until a relationship $\lessdot$ is found. This can be thought of as a scan from right-to-left, in contrast to the left-to-right scan to find the relationship $\gtrdot$. The handle then consists of all symbols between the $\lessdot$ and $\gtrdot$ relationships, and they are replaced by the left part of the production rule whose right part is satisfied by these symbols. An associated semantic routine can be called at this time.

The simple precedence method may be used for languages with grammars having few production rules if the rules can be modified to simple precedence form. For languages with grammars having many production rules, the resulting table would be too large to be practical. For some grammars, it is possible to replace the table by precedence functions f and g satisfying the following

relations for all symbols i and j of the grammar.

$f(i) < g(i)$ if $i \lessdot j$

$f(i) = g(i)$ if $i \doteq j$

$f(i) > g(i)$ if $i \gtrdot j$

Operator Precedence

In contrast to a simple precedence grammar in which relationships among all symbols must be considered, a parser for an operator precedence grammar only considers relationships among terminal symbols. From the point of view of the parser, in the grammar of figure 1.4, these are +,*,),(,and <number>, plus the enclosing symbol #.

	+	*	(	)	<number>	#
+	$\gtrdot$	$\lessdot$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$
*	$\gtrdot$	$\gtrdot$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$
(	$\lessdot$	$\lessdot$	$\lessdot$	$\doteq$	$\lessdot$	
)	$\gtrdot$	$\gtrdot$		$\gtrdot$		$\gtrdot$
<number>	$\gtrdot$	$\gtrdot$		$\gtrdot$		$\gtrdot$
#	$\lessdot$	$\lessdot$		$\lessdot$	$\lessdot$	$\doteq$

Figure 1.7.Operator precedence relationships

Figure 1.7 shows the precedence relationships for these symbols. At most one precedence relationship can hold between any two terminal symbols, and nonterminal symbols cannot be adjacent.

The parser stacks all incoming symbols until the relationship .> holds between the top terminal symbol and the incoming symbol. The parser uses the operator precedence tables (which are smaller than simple precedence tables since they involve only terminal symbols) or operator precedence functions based on these tables. It may use separate stacks for terminals (operators) and non-terminals (operands). When the relationship .> is found, the appropriate reduction can be made and the stack modified. As the nonterminals are ignored in parsing, the operator precedence semantic routines must do more than the simple precedence semantic routines to ensure the correctness of the reduction.

The operator precedence method is applicable to more languages than the simple precedence method. When a particular terminal symbol is used in two different ways, however, there may be more than one precedence relationship between two terminal symbols. For languages permitting such usage, an operator precedence grammar is not adequate.

1.2.3 Other Grammars and Parsing Methods

Simple precedence parsers and operator precedence parsers are not always satisfactory for programming languages. It is often easier to determine the production rule applicable to the handle of a string if we can look at more symbols to the left and right. Parsers for (m,k) precedence grammars, for instance, are more general than simple precedence parsers in that they determine precedence on the basis of at most m symbols in the stack and k in the input string. Parsers for (m,k) bounded context grammars consider at most m symbols to the left of a handle and k to the right to determine the applicable production rule. Parsers for LR(k) grammars can determine which non-terminal should replace the handle by scanning the input string from left to right and going at most k symbols to the right of the handle. These parsers store information with each stacked item that can be used to determine whether stacking or reduction is to occure and if the latter, which production rule is applicable.

Further information on parsing methods can be found in (Gries, 1977),(Aho and Ullman, 1973).

1.3 Semantic Analysis

The source program is transformed to an internal representation by semantic analysis routines and directed

by the syntax analysis of the parser. The time at which semantic analysis occures is influenced by the method chosen for syntax analysis. To avoid having to undo semantic actions when back up is necessary, semantic processing may be deferred until the completion of syntax processing. In this case, semantic analysis may be done on a parse tree that is the output of the parser. Semantic analysis associated with optimizing also occurs after the completion of syntax processing.

A semantic analysis routine develops the internal form of the output on the basis of information in the stack and the symbol table. In addition to the information needed for determination of a handle in syntax analysis, the stack may also contain semantic information. For syntax analysis, it is necessary to know that the current symbol is an identifier; for semantic analysis, we must know which identifier. It is in the semantic analysis routines that type-checking occurs. In addition, much that is implicit in the source program is made explicit at this time.

Internal forms of representation in common use include postfix notation, quadruples, triples, and trees. Let us consider each of these forms.

Postfix notation, in which operators follow their

operands, is especially useful for producing code for stack machines. The expression

a + b * c

appears in postfix notation as

abc * +

A quadruple consists of an operator, operands, and the location of the result of the operation. The following expression can be represented by the quadruples

* b c temporary 1

+ a temporary 1 temporary 2

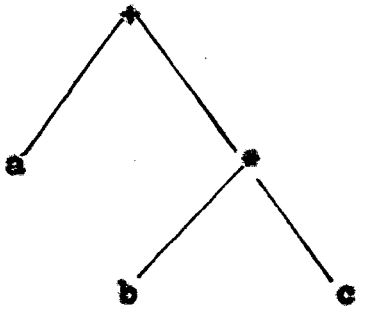
A triple consists of an operator and its operands. The result is not stated explicity, but the triples are numbered and can be referenced by subsequent triples. The expression above can be represented by the triples

(1) * b c

(2) + a (1)

Trees make explicit the information provided by triples. An entire parsed program can be represented by a tree. Each subtree of this tree represents an operator and its operands. Our example can be represented

in tree form as follows:



The task of semantic analysis is the transformation of the source program or its parse tree to an internal form from which code can be generated.

1.4 Code Generation

Code generation consists of the transformation of the internal form of a program to an assembly language or machine language form. The addressing mechanism and type of code depend on the specific target machine for which the code is generated. Addresses may be absolute or relocatable, depending on the software (generally, linkers and loaders) and hardware (for example, base registers or paging facilities) of the target system.

As noted above, postfix notation is appropriate for a stack machine and indeed, code may be emitted directly by the semantic analysis routines. Often, however, code is

produced by separate code-generating routines called generators.

Typically, a code-generating routine has a skeletal structure for its construct. If the internal form consists of quadruples or triples, specific information such as operand addresses can be passed as arguments to the routine, which can then insert it into the skeletal structure. Code generated for the expression a + b * c might be something like this:

```
load b
mpy c
store temporary 1
load a
add temporary 1
store temporary 2
```

Code for quadruples is generated in the order in which the quadruples appear. Information about the temporary results must be maintained throughout the generation process.

Code for triples is generated in a similar manner, but the internal representation is smaller and information about temporary results (in this case, the numbers of the triples) need be maintained only as long as the related triple can be referenced.

Code can be generated from trees using any of the several tree-walking algorithms. As the code is generated, the tree can be modified to permit optimization.

Code can be generated from postfix notation by scanning the postfix notation from left to right, placing operands on a stack, and having the code generator emit code for the current operator and the operands on the top of the stack.

A major problem of code generation has to do with the addressing of data and program. Issues to be considered include the time of binding variables to locations, the use of registers, and overlaying. With respect to the binding time of variables, both the binding times permitted by the language and those permitted by the system must be considered. The efficient use of registers is a form of optimization involving the fewest loads and stores of temporary results. The general problem of overlaying involves the sharing of the same space by disjoint local variables or temporary results as well as the problem of how to structure a program to make efficient use of segmentation or paging.

1.5 Optimization

We have mentioned optimization in connection with the efficient use of registers, but there are many places in

which optimization is appropriate. Compilers vary in the amount and kinds of optimization they do. There is a optimization of the source program which is the input to the compiler, and there is optimization of the object program which is the output of the compiler. Some optimization is with respect to a specific target computer, and some is independent of the computer. Efficient use of registers is an example of the former, since the number and types of registers varies from one machine to another. Examples of the latter include the evaluation of some expression at compile time rather than at run time, finding common expressions, and optimizing loops by pulling out of loops those expressions whose values do not change during the execution of the loop.

Optimization can occur at various points during the compilation process. Clever semantic analysis routines can achieve some optimization in producing the internal representation of the program. An optimizing compiler may do flow analysis on an internal tree representation prior to code generation. The result is a transformed tree representing a more efficient structure. Code generators may do some optimization, and, finally, the resultant code may be examined a few instructions at a time.

Some optimization alter the sequence of operations.

For purpose of evaluation, it may not matter whether the commutative operation a+b is evaluated as a+b or as b+a. However, some caution is needed. There are some cases in which changing the order of execution does have an effect upon the output of the program (for example, if evaluation of an operand involves calling a function which has side effects).

A particular optimization is usually not over an entire program but rather over several statements (such as a loop). The statements are organized so that execution can begin only with the first statement. A useful optimization is the pulling out of a loop those operations that need be performed only once instread of each time the loop is executed. If, say, the assignment a = b+c occurs within a loop and there are no changes to a, b, or c either within the loop or potentially from procedures called from within the loop, this statement can be pulled outside the loop and executed once prior to entering the loop. If a is modified but b and c are not, the addition can occur prior to entering the loop and the result stores in a temporary variable. The value assigned to a by this statement within the loop is then that of the temporary variable.

The above example of optimization is based on the elemination of redundant executions. It is also possible

to eliminate redundant, or common, expressions. Such expressions are evaluated once, and the result is stored in a temporary variable. Then, rather than reevaluate the expression, we can simply refer to the temporary variable. It is not always possible to tell when an expression is redundant, but checks on commutativity can recognize some cases.

CHAPTER II

AN ALGOL-60 SUBSET

Introduction

The purpose of this chapter is to present the definition description of a subset of the international algorithmic language Algol-60. This subset of Algol-60 is chosen for implimenting on EC 1020B Computer.

In the first article a survey of the basic constituents and features of the language is given, and the formal notation, by which syntactic structure is defined, is explained.

The second article lists all the basic symbols, and syntactic units known as identifiers and numbers are defined.

The third article explains the rules for forming expressions, and the meaning of these expressions. Two different type of expressions exist: arithmatic and Boolean.

The fourth article describes the operational units of the language, known as statements. The basic statements are: assignment statements(evaluation of a formula), go to statements (explicit break of the sequence of execution of statements), and procedure statements (call for execution

of a closed process, defined by a procedure declaration). The formation of more complex structures, having statement character, is explained. These includes conditional statements, compound statements and blocks.

In the fifth article the units known as declarations, serving for defining permanent properties of the units entring into a process described by the language, are defined.

DESCRIPTION OF THE REFERENCE LANGUAGE

2.1 Structure of the Language

The purpose of this algorithmic language is to describe computational processes. The basic concept used for the description of calculating rules is the well known arithmatic expression containing as constituents numbers and variables. From such expressions are compounded, by applying rules of arithmatic composition, self-contained units of the language-explicit formulae-called assignment statements.

To show the flow of computational processes, certain non-arithmetic statements and statement clauses are added. Since it is necessary for the function of these statements

that one statement refers to another, statements may be provided with labels. A sequence of statements may be enclosed between the the statement brackets begin and end to form a compound statement.

Statements are supported by declarations which are not themselves computing instructions, but inform the translator of the existence and certain properties of object appearing in statements, such as the class of numbers taken on as values by a variable. A sequence of declaration followed by a sequence of statements and enclosed between begin and end constitutes a block. Every declaration appears in a block in this way and is valid only for that block.

A program is a block or compound statement which is not contained within another statement and which makes no use of other statements not contained within it.

2.2 BASIC SYMBOLS, IDENFIFIERS,NUMBERS, AND BASIC CONCEPTS

The reference language is built up from the following basic symbols:

⟨basic symbol⟩ ::= ⟨letter⟩|⟨digit⟩|⟨delimiter⟩

2.2.1 LETTERS

⟨letter⟩ ::= A|B|C|D|E|F|G|H|I|J|K|L|M|
N|O|P|Q|R|S|T|U|V|W|X|Y|Z|
a|b|c|d|e|f|g|h|i|j|k|l|m|
n|o|p|q|r|s|t|u|v|w|x|y|z

This alphabet may arbitrarily be restricted. Letters do not have individual meaning. They are used for forming identifiers.

2.2.2. DIGITS

⟨digit⟩ ::= 0|1|2|3|4|5|6|7|8|9

Digits are used for forming numbers, identifiers.

2.2.3 DELIMITERS

⟨delimiter⟩ ::= ⟨operator⟩|⟨separator⟩|⟨bracket⟩|⟨declarator⟩

⟨operator⟩ ::= ⟨arithmetic operator⟩|⟨relational operator⟩|⟨sequential operator⟩

⟨arithmetic operator⟩ ::= + | - | * | /

⟨relational operator⟩ ::= < | ≤ | = | ≥ | > | ¬= | =< | =>

⟨sequential operator⟩ ::= goto | if | then | else

⟨separator⟩ ::= , | . | ; | : | := | comment

⟨bracket⟩ ::= (|) | begin | end

⟨declarator⟩ ::= integer | real | procedure

Delimiters have a fixed meaning which for the most part is obvious, or else will be given at the appropriate place in the sequel.

Typographical features such as blank space or change

to a new line have no significance in the reference language. They may, however, be used freely for facilitating reading.

For the purpose of including text among the symbols of a program the following 'comment' conventions hold:

The sequence of basic symbols:	is equivalent to
; comment ⟨any sequence not containing;⟩ ;	;
begin comment ⟨any sequence not containing ;⟩ ;	begin
end ⟨any sequence not containing end or;or else⟩	end

By equivalence is here meant that any of the three structures shown in the left-hand column may, in any occurrence outside of strings, be replaced by the symbol shown on the same line in the right-hand column without any effect on the action of the program. It is further understood that the comment structure encountered first in the text when reading from left to right has precedence in being replaced over later structures contained in the sequence.

2.2.4 IDENTIFIERS

2.2.4.1 SYNTAX

⟨identifier⟩ ::= ⟨letter⟩ | ⟨identifier⟩ ⟨letter⟩
| ⟨identifier⟩ ⟨digit⟩

2.2.4.2 EXAMPLES

X

V12

2.2.4.3 SEMANTICS

Identifiers have no inherent meaning, but serve for the identification of simple variables, labels, and procedures. They may be chosen freely.

The same identifier cannot be used to denote two different quantities except when these quantities have disjoint scopes as defined by the declarations of the program.

2.2.5 NUMBERS

2.2.5.1. SYNTAX.

⟨unsigned integer⟩::=⟨digit⟩|⟨unsigned integer⟩⟨digit⟩

⟨integer⟩::=⟨unsigned integer⟩|+⟨unsigned integer⟩
|-⟨unsigned integer⟩

⟨decimal fraction⟩::=.⟨unsigned integer⟩

⟨exponent part⟩::=E⟨integer⟩

⟨decimal number⟩::=⟨unsigned integer⟩|⟨decimal fraction⟩
⟨unsigned integer⟩⟨decimal fraction⟩

⟨unsigned number⟩::=⟨decimal number⟩⟨exponent part⟩|
⟨decimal number⟩⟨exponent part⟩

⟨number⟩::=⟨unsigned number⟩|+⟨unsigned number⟩
|-⟨unsigned number⟩

2.2.5.2 EXAMPLES

177	+7E5	2.3
- 3E7	+0.73	2E+3

2.2.5.3 SEMANTICS

Decimal numbers have their conventional meaning. The exponent part is a scale factor expressed as an integral power of 10.

2.2.5.4 TYPES

Integers are of type integer. All other numbers are of type real.

2.2.6 VALUES AND TYPES

A value is an ordered set of numbers (special case: a single number), an ordered set of logical values (special case: a single logical value), or a label.

Certain of the syntactic units are said to possess values. These values will in general change during the execution of the program. The values of expressions and their constituents are defined in Section 2.3.

The various types (integer, real, Boolean) basically denote properties of values. The types associated with syntactic units refer to the values of these units.

2.3 EXPRESSIONS

In the language the primary constituents of the programs describing algorithmic processes are arithmetic and boolean expressions. Constituents of these expressions, except for certain delimiters, are logical values, numbers, variables, and elementary arithmetic, relational, logical, and sequential operators. Since the syntactic definition of variables contains expressions, the definition of expressions, and their constituents, is necessary recursive.

⟨expression⟩ ::= ⟨arithmetic expression⟩|⟨boolean expression⟩

2.3.1 VARIABLES

2.3.1.1 SYNTAX

⟨variable⟩ ::= ⟨identifier⟩

2.3.1.2 EXAMPLES

X

A17

2.3.1.3 SEMANTICS

A variable is a designation given to a single value. This value may be used in expressions for forming other values and may be changed at will by means of assignment statements. The type of the value of a particular variable is defined in the declaration for the variable itself.

2.3.3 ARITHMETIC EXPRESSIONS

2.3.3.1 SYNTAX

⟨adding operator⟩::= + | -

⟨multiplying operator⟩::= * | /

⟨factor⟩::= ⟨unsigned number⟩|⟨variable⟩|
(⟨arithmetic expression⟩)

⟨term⟩::= ⟨factor⟩|⟨term⟩⟨multiplying operator⟩
⟨factor⟩

⟨arithmetic expression⟩::=
⟨arithmetic expression⟩ ⟨adding operator⟩
⟨term⟩| ⟨term⟩

2.3.3.2. EXAMPLES

factors:

(a-3/y) SUM

term:

U (A-3/Y)

arithmetic expression ::

U-V (A-3/Y)

2.3.3.3. SEMANTICS

An arithmetic expression is a rule for computing a numerical value. In case of simple arithmetic expressions

this value is obtained by executing the indicated arithmetic operations on the actual numerical values of the factors of the expression. The actual numerical value of a factor is obvious in the case of numbers. For variables it is the current value (assigned last in the dynamic sense). Finally, for arithmetic expressions enclosed in parentheses the value must through a recursive analysis be expressed in terms of the values of factors of the other two kinds.

2.3.3.4 OPERATORS AND TYPES

The constituents of simple arithmetic expressions must be type real or integer. The meaning of the basic operators and the types of the expressions to which they lead are given by the following rules.

2.3.3.4.1 The operators +,-, and * have the conventional meaning (addition, subtraction, and multiplication). The type of the expression will be integer if both of the operands are of integer type, otherwise real.

2.3.3.4.2 The operation term / factor
denote division, to be understood as a multiplication of the term by the reciprocal of the factor with due regard to the rules of precedence. The type of the expression will be integer if both operands are of integer type, otherwise real.

2.3.3.5 PRECEDENCE OF OPERATORS

The sequence of operations within one expression is generally done from left to right, with the following additional rules:

2.3.3.5.1 According to syntax given in section 2.3.3.1 the following rules of precedence hold:

first: * /

second: + -

2.3.3.5.2 The expression between a left parenthesis and the matching right parenthesis is evaluated by itself and this value is used in subsequent calculations.

2.3.3.6 ARITHMETICS OF REAL QUANTITIES

Numbers and variables of type real must be interpreted in the sense of numerical analysis, i.e., as entities defined inherently with only a finite accuracy. Similarly, the possibility of the occurrence of a finite deviation from the mathematically defined result in any arithmetic expression is explicity understood. No exact arithmetic will be specified, however, and it is indeed understood that different hardware representations may evaluate arithmetic expressions differently. The control of the possible consequences of such differences must be carried out by

the methods of numerical analysis. This control must be considered a part of the process to be described, and will therefore be expressed in terms of the language itself.

2.3.4 BOOLEAN EXPRESSIONS

2.3.4.1 SYNTAX

⟨relational operator⟩ ::= <|≤| =|>=| >|¬= | ≠⟩|=<

⟨boolean expression⟩::= ⟨boolean factor⟩
⟨relational operator⟩
⟨boolean factor⟩

⟨boolean factor⟩::= ⟨variable⟩|⟨number⟩

2.3.4.2 EXAMPLES

X = 2 5 - = 7

2.3.4.3 SEMANTICS

A boolean expression is a rule for computing a logical value. The principles of evaluation are entirely analogous to those given for arithmetic expressions.

2.3.4.4 THE OPERATORS

Relations take on the value true whenever the corresponding relation is satisfied for the expressions involved; otherwise false.

2.4 STATEMENTS

The units of operation within the language are called statements. They will normally be executed consecutively as written. However, this sequence of operations may be broken by go to statements, which define their successor explicitly, and shortened by conditional statements, which may cause certain statements to be skipped.

In order to make it possible to define a specific dynamic succession, statements may be provided with labels.

Since sequences of statements may be grouped together into compound statements and blocks, the definition of statement must necessarily be recursive. Also since declarations, described in Section 2.5, enter fundamentally into the syntactic structure, the syntactic definition of statements must suppose declarations to be already defined.

2.4.1 COMPOUND STATEMENTS AND BLOCKS

2.4.1.1 SYNTAX

⟨unlabelled basic statement⟩::=⟨assignment statement⟩|⟨go to statement⟩|⟨dummy statement⟩|⟨procedure statement⟩

⟨basic statement⟩::=⟨unlabelled basic statement⟩|⟨label⟩ : ⟨basic statement⟩

⟨unconditional statement⟩::= ⟨basic statement⟩ |
⟨compound statement⟩|⟨block⟩

⟨statement⟩::=⟨unconditional statement⟩|
⟨conditional statement⟩

⟨compound tail⟩::=⟨statement⟩ end |⟨statement⟩ ;
⟨compound tail⟩

⟨block head⟩::= begin ⟨declaration⟩|⟨block head⟩;
⟨declaration⟩

⟨unlabelled compound⟩::= begin ⟨compound tail⟩

⟨unlabelled block⟩::=⟨block head⟩ ; ⟨compound tail⟩

⟨compound statement⟩::=⟨unlabelled compound⟩ |⟨label⟩ :
⟨compound statement⟩

⟨block⟩::=⟨unlabelled block⟩ |⟨label⟩: ⟨block⟩

⟨program⟩::=⟨block⟩|⟨compound statement⟩

This syntax may be illustrated as follows: Denoting arbitrary statements, declarations, and labels, by letters S, D, and L, respectively, the basicsyntactic units take the forms:

Compound statement:

L:L:...begin S;S;...S;S end

Block:

L:L:...begin D;D;...D;S;S;...S;S end

It should be kept in mind that each of the statements S may again be a complete compound statement or block.

2.4.1.2 EXAMPLES

Basic statement:

```
A := P+Q
go to Next
```

Compound statement:

```
begin  X:=0;
       go to Next
end
```

Block:

```
Q: begin integer I, K; real W;
       I:=2*X/3; W:=I-5;
       K:=X↑7
   end   block Q
```

2.4.1.3 SEMANTICS

Every block automatically introduces a new level of nomenclature. This is realized as follows: Any identifier occuring within the block may through a suitable declaration be specified to be local to the block in question. This means (a) that the entity represented by this identifier inside the block has no existence outside it, and (b) that any entity represented by this identifier outside the block is completely inaccessible inside the block.

Identifiers (except those representing labels) occurring within a block and not being declared to this block will be non-local to it, i.e., will represent the same entity inside the block and in the level immediately outside it. A label separated by a colon from a statement, i.e., labelling that statement, behaves as though declared in the head of the smallest embracing block, i.e., the smallest block whose brackets begin and end enclose that statement. In this context a procedure body must be considered as if it were enclosed by begin and end and treated as a block.

Since a statement of a block may again itself be a block the concepts local and non-local to a block must be understood recursively. Thus an identifier, which is non-local to a block A, may or may not be non-local to the block B in which A is one statement.

2.4.2 ASSIGNMENT STATEMENTS

2.4.2.1 SYNTAX

⟨left part⟩ ::= ⟨variable⟩ :=|⟨procedure identifier⟩:=

⟨assignment statement⟩ ::=⟨left part⟩⟨arithmetic expression⟩

2.4.2.2 EXAMPLES

```
X: = 1
N: = N+1
A: = A*B
```

2.4.2.3 SEMANTICS

Assignment statements serve for assigning the value of an expression to one variable or procedure identifier. Assignment to a procedure identifier may only occur within the body of a procedure. The process will in the general case be understood to take place in two steps as follows:

2.4.2.3.1 The expression of the statement is evaluated.

2.4.2.3.2 The value of the expression is assigned to the left part variable.

2.4.2.4 TYPES

The type associated with all variables and procedure identifiers of a left part list must be the same. If the type is real or integer, the expression must be arithmetic. If the type of the arithmetic expression differs from that associated with the variables and procedure identifiers, appropriate transfer functions are understood to be automatically invoked.

2.4.3 GO TO STATEMENTS

2.4.3.1 SYNTAX

⟨go to statement⟩ ::= go to ⟨label identifier⟩

⟨label identifier⟩::= ⟨identifier⟩

2.4.3.2 EXAMPLES

go to NEXT

go to AGAIN

2.4.3.3 SEMANTICS

A go to statement interrupts the normal sequence of operations, defined by label identifier. Thus the next statement to be executed will be the one having these value as its label.

2.4.3.4 RESTRICTION

Since labels are inherently local, no go to statement can lead outside into a block. A go to statement may, however, lead from outside into a compound statement.

2.4.4 DUMMY STATEMENTS

2.4.4.1 SYNTAX

⟨dummy statement⟩::= ⟨empty⟩

2.4.4.2 EXAMPLES

```
L:
begin John:end
```

2.4.4.3 SEMANTICS

A dummy statement executes no operation. It may serve to place a label.

2.4.5 CONDITIONAL STATEMENTS

2.4.5.1 SYNTAX

```
⟨if clause⟩::= if ⟨boolean expression⟩ then
⟨unconditional statement⟩::= ⟨basic statement⟩|⟨block⟩|
                    ⟨compound statement⟩
⟨if statement⟩::=⟨if clause⟩ ⟨unconditional  statement⟩
⟨conditional statement⟩::=⟨if statement⟩ else⟨statement⟩
                    ⟨if statement⟩|⟨label⟩:⟨conditional
                     statement⟩
```

2.4.5.2 EXAMPLES

```
if X>0 then N:=N+1
if U=V then AA:begin
                    if q = V then A:=B
                             else Y:=2
                    end
```

2.4.5.3 SEMANTICS

Conditional statements cause certain statements to be executed or skipped depending on the running value of specified boolean expressions.

2.4.5.3.1 IF STATEMENT.

The unconditional statement of an if statement will be executed if boolean expression of the if clause is true. Otherwise it will be skipped and the operation will be continued with the next statement.

2.4.5.3.2 CONDITIONAL STATEMENT

According to the syntax two different form of conditional statements are possible. These may be illustrated as follows:

if B1 then S1 else if B2 then S2 else S3;S4

and

if B1 then S1 else if B2 then S2 else if B3 then S3;S4

Here B1 and B3 are boolean expressions, while S1 and S3 are unconditional statements. S4 is the statement following the complete conditional statement.

The execution of a conditional statement may be described as follows: The boolean expressions of the if

clauses are evaluated one after the other in sequence from left to right until one yielding the value true is found. Then the unconditional statement following this boolean is executed. Unless this statement defines its successor explicitly the next statement to be executed will be S4, the statement following the complete conditional statement. Thus the effect of the delimiter else may be described by saying that if defines the successor of the statement it follows to be the statement following the complete conditional statement.

The construction

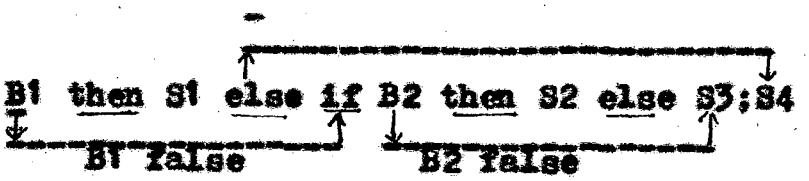
else ⟨unconditional statement⟩

is equivalent to

else if true then ⟨unconditional statement⟩

If none of the boolean expressions of the if clause is true, the effect of the whole conditional statement will be equivalent to that of a dummy statement.

For further explanation the following picture may be useful:



2.4.5.4 GO TO INTO A CONDITIONAL STATEMENT

The effect of a go to statement leading into a conditional statement follows directly from the above explanation of the effect of else.

2.4.6 PROCEDURE STATEMENTS

2.4.6.1 SYNTAX

⟨actual parameter⟩ ::= ⟨identifier⟩

⟨actual parameter list⟩ ::=⟨actual parameter⟩|
⟨actual parameter list⟩ , ⟨actual parameter⟩

⟨actual parameter part⟩ ::= (⟨actual parameter list⟩)|
⟨empty⟩

⟨procedure statement⟩ ::= ⟨procedure identifier⟩
⟨actual parameter part⟩

2.4.6.2 EXAMPLES

SFVR(A)

TRANS(A,B,C)

ABSL

2.4.6.3 SEMANTICS

A procedure statement serves to invoke (call for) the execution of a procedure body, where the procedure body is a statement written in Algol the effect of this execution

will be equivalent to the effect of performing the following operations on the program at the time of execution of the procedure statement:

2.4.6.3.1 ACTUAL-FORMAL CORRESPONDENCE

The correspondence between the actual parameters of the procedure statement and the formal parameters of the procedure heading is established as follows: The actual parameter list of the procedure statement must have the same number of entries as the formal parameter list of the procedure declaration heading. The correspondence is obtained by taking the entries of these two lists in the same order.

2.5 DECLARATIONS

Declarations serve to define certain properties of the quantities used in the program, and to associate them with identifiers. A declaration of an identifier is valid for one block. Outside this block the particular identifier may be used for other purposes.

Dynamically this implies the following: at the time of entry into a block (through the begin, since the labels inside are local and therefore inaccessible from outside) all

identifiers declared for the block assume the significance implied by the nature of the declarations given. If these identifiers had already been defined by other declarations outside they are for the time being given a new significance. Identifiers which are not declared for the block, on the other hand, retain their old meaning.

At the time of an exit from a block (through end, or by a go to statement) all identifiers which are declared for the block lose this local significance.

2.5.1 SYNTAX

⟨declaration⟩::= ⟨type declaration⟩|
⟨procedure declaration⟩

2.5.2 TYPE DECLARATIONS

2.5.2.1 SYNTAX

⟨typelist⟩ ::= ⟨variable⟩|⟨variable⟩ , ⟨type list⟩
⟨type⟩::= real | integer
⟨type declaration⟩ ::= ⟨type⟩⟨type list⟩

2.5.2.2 EXAMPLES

```
real    X,Y,Z
integer A,B,C
```

2.5.2.3 SEMANTICS

Type declaration serve to declare certain identifiers to represent simple variables of a given type, real declared variables may only assume positive or negative values including zero, integer declared variables may only assume positive and negative integral values, including zero.

In arithmetic expressions any position which can be occupied by a real declared variable may be occupied by an integer declared variable.

2.5.3.PROCEDURE DECLARATIONS

2.5.3.1 SYNTAX

⟨formal parameter⟩::= ⟨identifier⟩

⟨formal parameter list⟩::=⟨formal parameter⟩ |
⟨formal parameter list⟩ , ⟨formal parameter⟩

⟨formal parameter part⟩ ::=⟨empty⟩ | (⟨formal parameter list⟩)

⟨procedure heading⟩ ::=⟨procedure identifier⟩
⟨formal parameter part⟩

⟨procedure body⟩ ::= ⟨statement⟩

⟨procedure declaration⟩::= procedure⟨procedure heading⟩

2.5.3.2 EXAMPLE

```
procedure S(a);    integer a;
begin S:=a+1 end
```

2.5.4.3 SEMANTICS

A procedure declaration serves to define the procedure associated with a procedure identifier. The principle constituent of a procedure declaration is a statement, the procedure body, which through the use of procedure statements may be activated from other parts of the block in the head of which the procedure declaration appears. Formal parameters in the procedure body will, whenever the procedure is activated be assigned the value of or replaced by actual parameters. Identifiers in the procedure body which are not formal will be either local or non-local to the body depending on whether they are declared within the body or not. Those of them which are not non-local to the body may well be local to the block in the head of which the procedure declaration appears. The procedure body always acts like a block, whether it has the form of one or not. Consequently the scope of any label labelling a statement within the body or the body itself can never extend beyond the procedure body. In addition, if the identifier of a formal parameter is declared a new within the procedure body, it is thereby given a local significance and actual parameters which correspond to it are inaccessible throughout the scope of this inner local quantity.

CHAPTER III

DATA STRUCTURE

The compiler initiates translation of the source program by reading the program card. It identifies the tokens and accordingly the various tables such as symbol table, literal table, etc. are created. These tables provide the necessary information associated with each token. Tables like terminals are of fixed length while others are, of variable length. During the complete process it makes use of the various data structure.

This chapter critically describes the functions of the various data structure used, their format and their locations. Each such data structure is described below in details:

3.1 SYMBOL TABLE

This table completely describes all the programmer defined symbols or the identifiers used in the source program. The symbol table contains an entry for each identifier. The symbol table is created by the parser. Since we are using block structure in our source language, a block list is required to identify which identifier belongs to which block. This is described in section 3.5 of this chapter.

Entries in the symbol table are arranged in sequential order. New entries are added at the end of the table, with changing information in the block list. This table is searched sequentially to determine an entry in a block or in the whole symbol table. It has not been sorted in alpha betical order or entries placed in the same order to have a fast access to an entry since for small programs linear search is more efficient since no time is spent in sorting and the program to access on entry is also quite small.

The symbol table provides information in the following format:

Symbol	Declaration	Type	Address

Symbol of maximum nine characters in length is stored in nine bytes. Its declaration field gives the information about its declaration i.e., is this identifier declared or not. This field is an integral field occupying 2 bytes. If the identifier is not defined this field is zero and otherwise one.

Type of the symbol is also an important field and is made of 2 bytes. This field tells us the nature of

the identifier. The following type assignments have been used:

Type	Nature of the identifier
0	Type to be determined lateron
1	Label
2	Real
3	Integer
4	Entry

The address field consists of two bytes gives us the address of the label if identifier is label else it gives information about the storage space for the variable or starting address of the procedure.

Symbol table is located in the memory and can contain maximum of 1024 symbols.

3.2 LITERAL TABLE

The literal table consists of one entry for each literal in the source program. Its size is dynamic. The entries are added at the end of the table. Linear search is made to find out the value of the literal in the literal table. The literal table is stored on the disc. One record of 19 bytes is assigned to one entry in the literal table. The format of the literal table record is as

shown below:

Type	Length	Value

The type field of the literal table madeup of 2 bytes gives the nature of the literal as shown below:

Type	Nature of the literal
1	Integer
2	Fixed point real number
3	Floating point real number

Length field made up of 2 bytes describes the length of the literal. This will be used for next pass, when the value part of the literal will be converted in bits. Value field made up of 15 bytes contains the literal in the character form.

3.3 SPECIAL CHARACTER TABLE

The special character table is a permanent table that has one entry for each terminal symbol. Each entry is one byte, and contains a terminal character. Terminal characters are

(+ - / *) , ; : ¬ < = >

This table is organized in the main memory and searched linearly.

3.4 KEYWORD TABLE

Like special character table, keyword table is also a permanent table which is list of all keywords of the source language in symbolic form. There is one entry for each keyword. Keyword table consists of the following keywords:

if then else procedure begin goto read write end comment real integer

3.5 BLOCK LIST

Algol has nested block and procedure structure. The same identifier may be declared and used many times in different blocks and procedures, and each such declaration must have a unique symbol table entry associated with it.

Block list table gives us basic information about the local variables in various blocks. Each entry of the block list consists of four fields:

Surrounding number of blocks	Number of entries in this block	Pointer to the symbol table	Block identi-fication

Each entry consists of 2 bytes and is stored in the main memory. Total number of blocks that one can use is 128.

Surrounding number of blocks field tells us the level of the block. Number of entries field tells the number of local variables in this block. Pointer to symbol table field gives the pointer to the symbol table where local variables are stored. Last field i.e. block identification tells the nature of the block. If this field is zero, block is begin and if it is one then it is procedure.

3.6 TEMPORARY LABEL TABLE

This table is used to store temporary labels. Some labels are required, which cannot be identified until whole of the statement is passed and code for that statement is generated. The only entry in this table is the address of some intermediate code matrix entry. Each entry of this table is of two bytes. This table is on the disc.

Temporary Address

3.7 ERROR MESSAGE TABLE

This is a permanent table inside the memory. Each entry of this table is of variable size, depending upon the nature of the error. Message from this table are taken & printed in the error. Message file. The format of the error

message table is as follows:

Message identification	Error message

First field is of two bytes gives message identification. Second field i.e. error message field is of variable length gives the nature of the error.

3.8 ERROR MESSAGE FILE

Errors discovered in the source program during the compilation are stored in the error message file. The file is stored on the disc. Each record occupies 50 bytes and contains three fields. The entries in the table are made sequentially and they are printed out in the same order. Each entry of the error message table looks like as shown below:

S.No.	Statement No.	Error message

S.No. field is made up of two bytes. Statement number i.e. the statement in which error has occurred, is made up of two bytes. Error message field consists

of remaining 46 bytes contains a dfscription of error message. These error message are taken from the error message table. Each entry of error message file is printed out after the compilation.

3.9 INTERMEDIATE CODE

The intermediate code table contains one entry for each intermediate instruction. It includes both the machine instruction mnemonics and the pseudo instruction mnemonics. For each machine instruction we need to store a three byte character code of symbolic code and address field specification. Address field of each instruction consists of two fields, i.e. address of first operand, and address of second operand. Each address is again consists of three parts namely table, code and block number. With the above organisation the format of the intermediate code looks like:

Op-code	Tab1	Code1	Block1	Tab2	Code2	Block2

Op-code field is of three bytes, remaining all the six fields are of two bytes. Each entry consists of 15 bytes is organised on disc in sequential order.

CONCLUSIONS

These are main data structure used in the compiler implementation. The compiler uses other structures like stacks, pointers etc. are given in programming implementation. Since these are transient data structures and hence are not described here. These are described in respective implementation parts.

CHAPTER IV

LOGIC AND ALGORITHM OF THE COMPILER

This chapter describes in details the phases of the compiler. Each phase is assigned with a fixed task. How this task is done in these phases is described here. Logic and algorithm for each phase is given in detail. How these phases interact is also described. Here we study only first four phases of the compiler. These phases are lexical analysis, syntax analysis, semantic analysis and intermediate code generator.

All these phases are machine independent. These phases are combined to form first pass of the compiler. A second pass of the compiler can be written in which optimization and final object code generation can be done. The general model of the compiler can be visualized as follows:

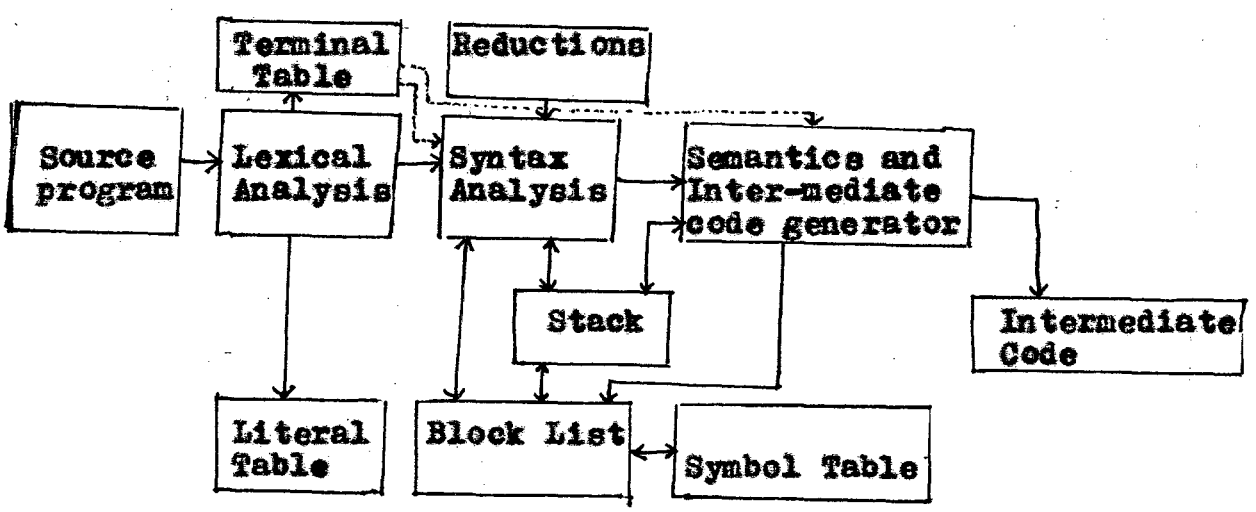


Figure 4.1 The interaction of lexical, syntax semantics and intermediate code generator.

4.1 LEXICAL PHASE

The purpose of the lexical analysis is (1) to parse the source program into the basic elements (tokens) of the language (2) to build a literal table.

Algorithm:

The input of the lexical analysis phase is a string of characters & output are basic elements. In our implementation lexical phase is called whenever there is a need for the next token. At end of the card file, conditions are set, which indicates that there is no more token available. The imput string consists of token is scanned character by character until one token is found. Each character is checked for legality, and tested to see if they are break characters. Consecutive non-break characters are accumulated into a token. Blanks are also treated as break characters. There are four types of tokens (1) identifiers, (2) literals, (3) keywords, and (4) special and operator characters. Lexical recognizes each one of these tokens. Comments and blanks are ignored in our implementation.

Initially input string characters are available in a buffer. There is one pointer to this buffer area from where next token is to be searched. This pointer is moved

forward equal to the length of the token while return is made. Lexical checks first the pointed character for letter. In case it is a letter, input string is scanned until an identifier is formed. Now there are two possibilities, one it may be a keyword, and second it may be a variable or label. This so formed identifier is compared with each keyword until a match is found. If there is a match then it means that a token so formed is a keyword and appropriate information about this keyword is returned. Otherwise it is label or a variable and hence necessary information is returned. If input pointed character is not a letter, it is checked for nummeric character. If it is a nummeric character, input string is scanned until a nummeric literal is found and literal so formed is stored in literal table for further necessary information, and lexical returns the token as a literal. Finally if input pointed character is not nummeric, it is checked for each delimiter. If it is not delimiter, error conditions are set and return is made. Otherwise next input pointed character is also tested for various combinations of double characters delimeters.

While doing so there can be some errors like length of the identifier etc. which are recorded in the

error file with card numbers, and error message number.

4.2 SYNTAX PHASE

The function of the syntax phase is to recognize the major constructs of the language and to call the appropriate action routines that will generate the intermediate form or mertix for these constructs. In our compiler this phase is implemented by one large program that recognizes each construct. Out parser has a input, a stack, and a parsing table. The input is read from left to right, one symbol at a time. The stack contains a string of the form $s_0X_1s_1X_2s_2.....X_ms_m$, where s_m is on top. Each X_i is a grammar symbol and each s_i is a symbol, called a state. Each state symbol summarizes the information contained in the stack below it and is used to guide the shift-reduce decision. In an actual implementation, the grammar symbols need not appear on the stack. We include them to explain the behaviour of the parser.

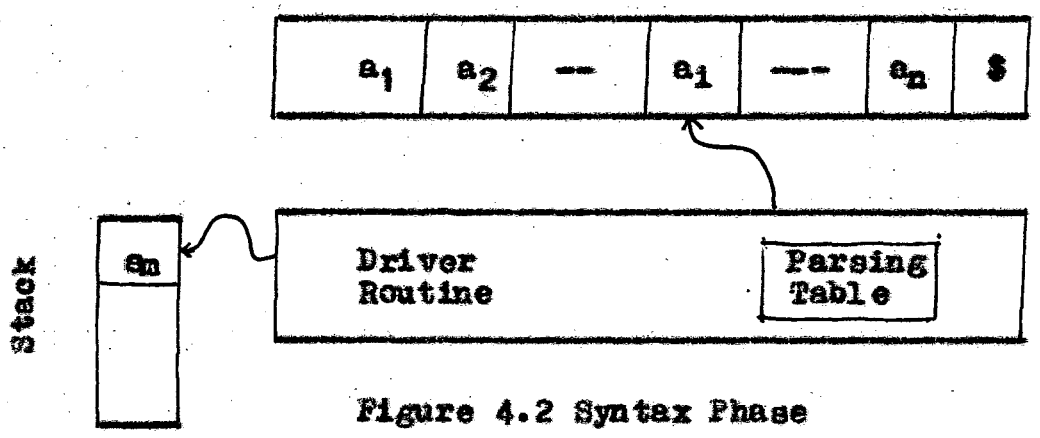


Figure 4.2 Syntax Phase

The program driving the parser behaves as follows. It determines s_m, the state currently on top of the stack, and a_i, the current input symbol. It then consults Action $(s_m a_i)$, the parsing action table entry for state s_m and the input a_i. The entry Action (s_m, a_i) can have one of four values:

1. Shift S
2. reduce $A \longrightarrow \beta$
3. accept
4. error

A configuration of the parser is a pair whose first component is the stack contents and whose second component is the unexpected input:

$$(s_0 X_1 s_1 X_2 s_2 \cdots\cdots X_m s_m, a_i a_{i+1} \cdots\cdots a_n \$)$$

The next move of the parser is determined by reading a_i, the current input symbol, and s_m, the state on top of the stack, and then consulting action table entry Action (s_m, a_i). The configurations resulting after each of the four types of move are as follows:

1. If Action (s_m, a_i)=shift s, the parser executes a shift move, entering the configuration
   $(s_0 X_1 s_1 X_2 s_2 \cdots X_m s_m a_i s, \quad a_{i+1} \cdots a_n \$)$
   Here the parser has shifted the current input symbol a_i and the next s=GOTO(s_m, a_i) onto the stack,

a_{i+1} becomes the new current input symbol.

2. If Action (s_m, a_i)=reduce $A \longrightarrow \beta$, then the parser executes a reduce move, entering the configuration $(s_0 X_1 s_1 X_2 s_2 \ldots\ldots X_{m-r} s_{m-r} A s, \; a_i, a_{i+1} \ldots a_n \$)$ where s=GOTO(s_{m-r}, A) and r is the length of β, the right side of the production. Here the parser first popped $2r$ symbols off the stack (r state symbols and r grammar symbols), exposing state s_{m-r}. The parser then pushed both A, the left side of the production, and s, the entry for Action (s_{m-r}, A), onto the stack. The current input symbol is not changed in a reduce move. For the parser we shall construct, $X_{m-r+1} \ldots\ldots\ldots X_m$, the sequence of grammar symbols popped off the stack, will always match β, the right side of the reducing production.

3. If Action (s_m, a_i)=accept, parsing is completed.

4. If Action (s_m, a_i)=error, the parser has discovered an error and calls an error recovery routine.

We illustrate the technique using the following grammar:

⟨prog⟩ ::= ⟨state⟩

⟨state⟩ ::=**if** ⟨expr⟩ **then** ⟨state⟩
⟨state⟩ ::= ⟨var⟩ := ⟨expr⟩
⟨expr⟩ ::= ⟨expr⟩ + ⟨var⟩ | ⟨var⟩ (1)
⟨var⟩ ::=i

We make up an Action matrix whose rows represent heads (which end in a terminal symbol) of right sides of rules which may appear in the stack, and whose columns represent the terminal symbols, including the sentence delimiter #. The elements of the matrix will be numbers or addresses of subroutines. These subroutines are action routines for interpretation phase.

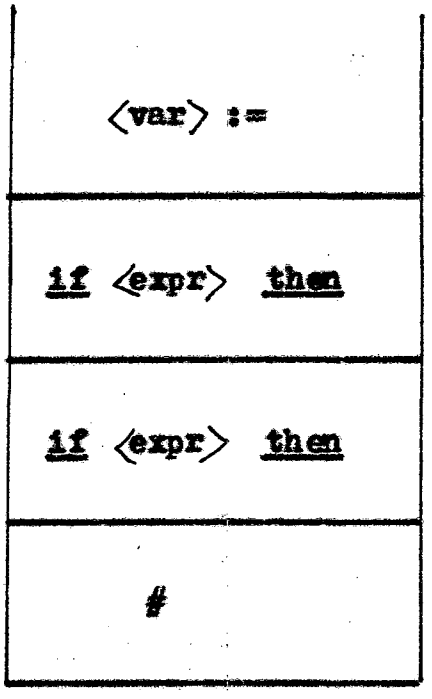
	#	**if**	**then**	:=	+	i
#		1				1
if						
if ⟨expr⟩ **then**						
⟨var⟩ :=						
⟨expr⟩ +						
i	2		2	2	2	

Figure 4.3 Partial filled Transition Matrix

The recognizer uses the typical stack S and incoming symbol variable R. We structure the stack a little differently, appear in a stack element. Strings appearing here will be heads (which end in a terminal symbol) of right parts of rules. For example, if the conventional stack at some point contained

if ⟨expr⟩ **then** **if** ⟨expr⟩ **then** ⟨var⟩ :=

The stack would look like



Note that all we are doing is keeping together those symbols which we know must be reduced at the same time. One final point we need a third variable U. It is either empty or contains the symbol to which the last prime phase

has been reduced. Thus, if the partial string parsed so far is

if ⟨expr⟩ then if ⟨expr⟩ then ⟨var⟩ := ⟨expr⟩

Then the stack would be as above and ⟨expr⟩ would be in U.

The recognizer uses the matrix as follows. At each step, the top stack element corresponds to some row of the matrix, since both represent the head (ending in a terminal symbol) of some right part. The incoming terminal symbol R determines a column of the matrix. These two together determine an element of the matrix which is the number of a subroutine to execute. This subroutine will perform the necessary reduction or will stack R and scan the next source symbol.

For example, when we start out we have # in the stack and U empty. The incoming symbol, according to grammar (1), must be either if or i. Since these begin right parts, we want to stack them and go on. The first subroutine is then

1: if U≠ ' ' then error; I=I-1;S(I)=R;scan.

where scan means to put the next symbol of the input string into R. The check on U being empty will become clear in a moment.

Suppose i is at the top of the stack, what can be

a valid symbol in R? i can be followed by the symbol #, then, := or +. What we want to do in all these cases is to change i to ⟨var⟩, making reduction

....⟨var⟩.....:=.....i.....

subroutine 2 to do this is

2:if U≠ ' ' then error; I:=I-1; U:= ⟨var⟩

Each matrix element, then is a number of a subroutine which either stack the incoming symbol or makes a reduction. The stacking is a little bit more involved because of the way each element looks. The complete matrix and subroutines for grammar (1) are in figure 4.4. and 4.5.

	#	if	then	:=	+	i
#	5	1	0	6	0	1
if	0	0	9	0	3	1
if ⟨expr⟩ then	7	1	0	6	0	1
⟨var⟩:=	8	0	0	0	3	1
⟨expr⟩ +	4	0	4	0	4	1
i	2	0	2	2	2	0

Figure 4.4 Transition Matrix for grammar (1)

```
1:    if U≠ ' ' then error;
      I=I-1; S(I)=R; scan;
2:    if U≠ ' ' then error;
      I=I-1; U=' ⟨var⟩ ';
3:    if U≠ '⟨expr⟩' or U≠ '⟨var⟩' then error
      I=I+1; S(I)='⟨expr⟩+'; U='  '; scan;
4:    ifU≠ '⟨var⟩'  then error;
      I=I-1; U= '⟨expr⟩';
5:    if U≠ '⟨prog⟩' or U≠ '⟨state⟩' then error;
      stop;
6:    if U≠ '⟨var⟩' then error;
      U= ' '; I=I+1;          S(I)='⟨var⟩:=';scan;
7:    if U≠ '⟨state⟩' then error;
      I=I-1; U='⟨state⟩';
8:    if U≠ '⟨var⟩' or U≠ '⟨expr⟩' then error;
      I=I-1;   U= '⟨state⟩';
9:    if U≠ '⟨var⟩' or U≠ '⟨expr⟩' then error;
      S(I)='if ⟨expr⟩then';
      U=' '; scan;
0:    error; stop;
```

Figure 4.5 Subroutines for Grammar(1)

This technique is used for our Algol-60 subset compiler, where the matrices are all constructed by hand in much the same manner as we have done. Each element of the matrix was filled in by determining from the language what the incoming symbol could be and what corresponding action should be performed. The use of the matrix allowed us to concentrate on one particular construct at a time and thus helped break the project up into a number of little ones. This technique is very fast, because no searching is required at all; each subroutine knows exactly what it has to do. Another nice point is that error recovery can be incorporated very easily. Each zero in the matrix corresponds to a syntax error and, since over half the elements are usually zero, one can write many routines to handle these errors and have several ways of recovering. With all the other techniques no good error recovery has been devised, because one cannot really break it up easily into sub cases.

STATEMENT PARSING

In this section the algorithm for the parsing of the statement is discussed. These routine accept the input from the lexical analysis and generate intermediate

	#	if	then	esle	Procedure.	Begin.	goto	Read	write.	end	<Rel>	;	:=	<I>
#	5	1	0	0	34	21	1	1	1	22	0	5	6	10
<if>	0	0	9	0	18	23	0	0	0	23	0	0	0	17
if <expr> then	0	1	0	20	18	21	1	1	1	25	0	7	6	10
if <expb> then <s> else	0	1	0	0	18	21	1	1	1	25	0	32	6	10
procedure	0	1	0	0	35	21	1	1	1	33	0	36	6	10
begin	0	1	0	0	34	21	2	1	1	22	0	26	6	10
goto	0	0	0	0	18	23	0	0	0	3	0	3	0	2
Read	0	0	0	0	18	23	0	0	0	28	0	24	0	13
write	0	0	0	0	18	23	0	0	0	28	0	24	0	14
<var> :=	0	0	0	0	18	23	0	0	0	8	19	8	0	15
<var><rel>	0	0	0	0	18	23	0	0	0	23	0	0	0	12
IF <I>	0	0	0	0	18	23	0	0	0	23	11	0	0	0
<I>	0	0	0	0	18	23	0	0	0	29	16	29	4	0
if <expb> then <s>	30	30	0	31	30	30	30	30	30	30	30	0	0	30

Figure 4.6 Decision Matrix for parsing of statement for ALGOL-60.

code in the matrix form. Since separate routines have been written for handling identifier, literals etc. while implementing the algorithm, their consideration is taken into account. To study the parsing of statement grammar is taken from chapter 2 section 2.4.1. Decision table for the grammar is given in figure 4.6. The entries of the decision matrix denoted by numbers are the semantic routines. The task of these routines is described as follows:

1: if S(I)='begin' then U=' '; if U≠' ' then error; I=I+1; S(I)=R; scan;

2: if U≠' ' then error; U='L'; US=R; scan;

3: if U≠'L' then error; U='S'; I=I-1; gen(BRN,US);

4: if U≠' ' then error; U='V'; I=I-1;

5: if U≠'S' then if U≠' ' then error; if R='#' then return;

6: if U≠'V' then error; I=I+1; S(I)='⟨var⟩:='; scan;

7: if U≠'S' then error; S(I)='if ⟨expr⟩ then ⟨s⟩ ';scan;

8: if U≠'E' then error; I=I-1; U='S'; gen(MOV US,UTEMP);

9: if U≠'B' then error; S(I)='if ⟨expr⟩ then';U=' '; gen (BRN CNCD,UTAB);

10: if S(I)≠'begin' then if U≠'S' then U=' '; if U≠' ' then error; I=I+1; S(I)= 1 ; scan;

11: if U≠' ' then error; U=' '; CNCD=CC(R); S(I)='if'; I=I+1; S(I)=' ⟨var⟩⟨relop⟩ '; scan;

```
12:  if U≠' ' then error; I=I-1; U='B'; gen (CMR US,R);
     scan;
13:  call io; U='S'; I=I-1;
14:  call io; U='S'; I=I-1;
15:  call expr; U='E';
16:  if U≠' ' then error; I=I-1; scan;
17:  if U≠' ' then error; US=R; S(I)='if⟨I⟩'; scan;
18:  error; I=I-1;
19:  call expr; U='E';
20:  if U≠'S' then error; S(I)='if ⟨expr⟩ then ⟨s⟩else⟩';
     U=' '; gen (BRN, 15, UTAB); scan;
21:  if S(I)='begin' then if U='S' then U=' '; If U≠' '
     then error; I=I+1; S(I)='begin';
22:  error; return;
23:  error; I=I-1;
24:  if U≠'S' then error; I=I-1;
25:  if U≠'S' then error; I=I-1;
26:  if U≠'S' then error; U=' '; scan;
27:  if U≠'S' then error; U='S'; I=I-1; scan;
28:  if U≠'S' then error; U='S'; I=I-1;
29:  if U≠' ' then error; U='S'; I=I-1; gen (JMS R);
30:  I=I-1;
31:  S(I)='if ⟨expr⟩then ⟨s⟩else'; U=' '; gen(BRN 15, UTAB);
     scan;
```

```
32:  if U≠'S' then error; I=I-1; U='S';
33:  error; U=' '; scan;
34:  if U≠' ' then error; I=I+1; S(I)='procedure';scan;
35:  error; I=I+1; S(I)='procedure', scan;
36:  if U≠'S' then error; gen (DNDP) scan;
```

ARITHMETIC EXPRESSIONS PARSING

In this section parsing method for an arithmetic expression is described. Here it is not taking the symbol tables, attributes etc. into consideration as these are used while implementing. The syntax for the arithmetic expression is as follows:

```
E ::= T|E+T|E-T|-T
T ::= F|T*F|T/F
F ::= I|(E)
```

Decision table for the above grammar is given in figure 4.7. The entries of the decision matrix denoted by the numbers are the semantic routines. Task of these routines is described as follows:

```
1:  if U≠' ' then error; I=I+1; S(I)=R; scan;
2:  if U≠'T' | U≠'E' then error; U='E'; return;
3:  if U≠'T' then error; I=I-1; U='E'; gena(UFLAG);
4:  if U≠'T' then error; I=I-1; gena (UFLAG);
5:  if U≠'T' then error; I=I-1; S(I)='e+'; U=' ';scan;
```

```
6:    if U≠'F' then error; I=I+1; S(I)='T/'; U=' ';scan;
7:    if U≠'F' then error; I=I+1; S(I)='T*'; U=' ';scan;
8:    if U≠'T' then error; I=I-1; genml(UFLAG);
9:    if U≠'T' then error; I=I-1; U='F'; genml(UFLAG);
10:   if U≠' ' then error; U='T'; US=R; UFLAG=1; scan;
11:   if U≠' ' then error; I=I+1; S(I)=R; US=R; UFLAG=1;
      scan;
12:   if U≠' ' then error; I=I-1; U='T';
      if S(I-1)='#'|S(I-1)='('|S(I-1)='-'then MOV(US);
13:   if U≠' ' then error; I=I-1; U='F'; MOV (US);
14:   if U≠'T'|U≠'E' then error; S(I)='(e)'; U=' ';scan;
15:   if U≠' ' then error; I=I-1; U='T';
16:   if U≠' ' then error; I=I-1;
      if S(I-1)='T*'|S(I-1)='T/' then U='T'; else U='F';
17:   if U≠' ' then error; if S(I-1)=1 then return; else
      goto 12;
18:   if U≠' ' then error; S(I)='<e>'; scan;
19:   if U≠'T' then error; I=I-1; genm(UTEMP);
20:   if U≠'T' then begin if U≠' 'then error;I=I+1;
                          S(I)='-'; scan; end;
            else I=I-1; S(I)='<e>'; U=' '; scan;
21:   if U≠'F' then error; I=I+1; genm (UTEMP);
```

The routine error is executed, whenever some error is invoked. The routine gena is called to generate an add instruction. If UFLAG=0 intermediate code generated will be

```
ADD   TEMP1, TEMP2
```

where TEMP2 is temporary location whose address is on the top of the stack UTEMP. TEMP1 is the next top of stack. Stack UTEMP is popped. If UFLAG=1 code is generated like

```
ADD   TEMP2, UR
```

where UR contains the address of some entry in the symbol table. In this case UFLAG is set to zero. Similarly, SUB instruction is generated by this routine. It is generated when top of the stack S is e -. Similar interpretation is for GENML (for multiplication and division), GENM (for unary operation minus) and MOV (for transferring contents).

	#	(	+	-	*	/	)	I
#	2	1	5	20	6	7	0	11
e +	3	1	4	4	6	7	4	11
e -	3	1	4	4	6	7	4	11
T *	8	1	8	8	9	9	8	10
T /	8	1	8	8	9	9	8	10
-	19	1	19	19	21	21	19	11
I	17	0	12	12	13	13	12	0
(	0	1	5	20	6	7	14	11
(⟨e⟩)	15	0	15	18	16	16	15	0

Figure 4.7 Decision Matrix for parsing of expression for our subset of Algol-60.

The entries in the decision matrix for expression are different from entries that appear in decision matrix for statement. These entries indicate the action routine to be executed next. A zero entry in the decision table indicates the syntax error in the source program. For example an expression cannot begin with right paranthesis. These action routines are executed and appropriate intermediate code is generated. A detailed information, about, what is done in these action routines, is given in the interpretation phase.

4.3 SEMANTICS AND INTERMEDIATE CODE GENERATOR PHASE

This phase is a collection of routines that are called when a construct is recognized in the syntactic phase. The purpose of these routines (called action routines) is to create an intermediate form of the source and update the identifier table. The separation of this phase from the syntactic phase is a logical division. The latter phase recognizes syntactic constructs while the former interprets the precise meaning into the matrix or identifier table.

ALGORITHM

Syntax analysis calls the interpretation phase. In our implementation there is no clear cut line where syntax

phase is over and where semantics starts. Once a construct is recognised then its intermediate code is generated. As we have seen in the syntax phase, there are some entries in the decision matrix, these entries are the numbers of the action routines to be executed. The purpose of these action routines is to check the information associated with the top of the stack and incoming symbol. If there is no matching an error message is recorded in the error message file. If some construct is recognized and there is no error of any kind, the intermediate code is generated for that construct. Semantics for various constructs is summarized as follows:

CONDITIONAL STATEMENT

The usual definition of a conditional statement is

⟨statement1⟩::= ⟨if clause⟩⟨statement2⟩else ⟨statement3⟩|
⟨if clause⟩⟨statement2⟩

⟨if clause⟩::= if ⟨exprb⟩ then

⟨exprb⟩::= ⟨var1⟩⟨relop⟩⟨var2⟩

We should generate one of the following sequences.

(1)	CMP	V1,V2	(1)	CMP	V1,V2
(2)	BRN	CC,q+1	(2)	BRN	CC,q
	⟨statement2⟩	OR		⟨statement2⟩	

(q) BRN 15,r (q)

(q+1) ⟨statement3⟩

(r)

Where CC is the condition code for which branching is to take place. If CC=15 then it is an unconditional branch. The code BRN is generated by the routine for ⟨if clause⟩::= if ⟨exprb⟩ then. Of cource, we do not know where to branch prior to parsing of ⟨statement2⟩. To overcome this difficulty we stack the address of this entry, and when ⟨statement2⟩ is passed the stack is poped and address of that entry is filled. In actual implementation these addresses are stored in a table and their references are made through the pointer. If it is a if then else statement then there is a little problem. In this case branching address is q+1 and at matrix entry q. There is an independent branch r. In order to know what is the actual address r, value q is stack and poped when ⟨statement3⟩ is parsed.

These routines are:

⟨if clause⟩ ::= if ⟨exprb⟩ then

Check ⟨exprb⟩ is boolean. If false then error. Generate the BRN code. Save the address of this newly generated entry in the stack. Stack will be poped when ⟨statement2⟩ will be parsed and code for it will be generated. Current value of the matrix entry is the address for branch instruction.

⟨statement1⟩ ::= ⟨if clause⟩⟨statement2⟩

Stack is poped which contained the matrix entry in which modification of the address portion is to be done. The next matrix entry is the actual address for the branch instruction.

⟨statement1⟩ ::= ⟨if clause⟩⟨statement2⟩
else ⟨statement3⟩

Stack is poped and address is modified. Stack is again pushed with the current entry which is to be modified after ⟨statement3⟩ will be parsed.

LABELS AND BRANCHES

In our syntax, label are defined as follows:

⟨statement1⟩ ::= I: ⟨statement 2⟩

where "I" is the nonterminal, meaning identifier. The labels can be referenced before their actual definitions. We write the following semantic routine:

⟨label definition⟩ ::= I :

Search the identifier in the current block. If it is not there, put it in the symbol table and make it as a label. If it is already there, make sure it is a label. If it is already declared print error message in error file. Otherwise set declaration bit and fill the address.

The semantic routine for branch goto I is

⟨statement1⟩ ::= goto I

Search the identifier in the current block. If it is not there put it in symbol table and make it as a label. Make it undeclared. If it is there; if it not label type print message. Generate the branch matrix code.

BLOCK STATEMENTS AND COMPOUND STATEMENTS

A compound statement is a block statement if there is no declaration statement available in it. So we will discuss only block statement. General block statement is

L1:begin D1,D2,...Dn,S1,S2...Sm end

where Li is a label. Di($1 \leq i \leq n$) are declarations and Sj($1 \leq j \leq m$) are statements which may be block statements. This syntax of the block can be written as:

(1) ⟨block begin⟩ ::= begin

(2) ⟨block⟩ ::= ⟨block begin⟩ ⟨declarations⟩ ;
⟨statements⟩ end

We open a block when routine (1) is executed and close it when routine (2) is executed. When a block is closed labels are local to the next surrounded block. More details about block structure has already been discussed in chapter III.

ARITHMETIC EXPRESSIONS

General form of our arithmetic expression syntax is as follows:

E::=T|E+T|E-T|-T
T::=F|T*F|T/F
F::=I|(E)

To study the Intermediate Code Phase, we use the rules of operator precedence, to associate the proper operands with an operator and then put the operations into the matrix in the sequence they should be executed.

We will discuss various routines as follows:

F::=I

This routine requires no quadruples to be generated because no binary operation is involved. We must only look up the identifier in the symbol table and associate its entry with F. We, thus, have the semantic routine

Look up the name associated with the identifier. If it is not available give error message. Associate the address of the table entry with ⟨F⟩.	Search(I Name,P) If P=0 then error ⟨F⟩. entry:= P

The F::= (E), requires no quadruples to be generated. Simply we have to associate entry with F. We thus have the semantic routine

F::= (E)

⟨F⟩.entry:=⟨E⟩. entry	Associate the address of expression entry with factor entry.

Semantic routines for E::=T, and T::=F are similar to that of F::=I, and F::=(E).

Next consider the semantic routines for

E::= E+T

In this routine first type of E and T is checked if they are of the same type and instruction is generated. If they are not of the same type, T is first converted to the type of E and then new entry is used for add instruction. Similarly we have semantic routines for

T::=T*F,T::=T/Fand E::=E-T.

For E::=-T

a simple instruction is generated i.e. NEG and such new entry is used as an expression entry. Its type is same as that of T.

CHAPTER V

PROGRAMMING DETAILS

The present work contains the implementation of the first end of the Algol-60 subset. This includes lexical analysis, syntax analysis, semantics analysis and the intermediate code generator. All this has been implemented in PL/I on EC 1020B. The whole implementation contains many subroutines with a main program to start and end the job. Subroutines for a well defined and predetermined task. The compilation starts with the main program when it reads a source program card. Various subroutines are called to compile it. The process of reading a card and compiling it is continued until the end of source program. This chapter describes the function of various subroutines taking part in compilation of Algol-60 subset statements.

5.1 MAIN PROGRAM

This program consists of declarations in which some necessary data is defined and initialized to meet the requirement. Files, error messages are also declared in this section of the program. Initialization of stack pointers, and other important variables is done in this section. It then passes control to STATE subroutine which

is a routine performing the task of parsing a statement. The STATE subroutine then controls the flow and calls the other various routines whenever required. When the source program ends this STATE returns control to the main program. It then prepares all the files for updation and closes them. After closing the files main program terminates to stop compilation.

5.2 STATE

This is the most important routine of the compiler. It is designed for parsing a statement of Algol-60 subset. It calls GETSYM, EXPR, BLKLS, NEXTSN, ERROR, IO, SEARCHB, SEARCH, INSERT and other routines to complete the process. How this routine parses a statement is explained in chapter IV. This routine uses a stack S and an incoming symbol, R. Depending on the top of the stack and incoming symbol a particular routine is executed. In case, top of the stack and incoming symbol are not matching according to the syntax of the language an error condition is detected. An error code of four characters is printed, first two characters show the top of the stack and last two conclude the illegal matching of keywords or delimeters. To distinguish whether the error occured in the statement structure or in expression

structure another character is printed along with the error code. An 'S' means an error in parsing of the statement, and an 'E' indicates that in the expression. After printing the error message in the error message file, the statement under consideration is ignored and the stack is poped up except when it is begin or procedure etc. Now input string is scanned until end of the current statement is found. In case, there is no error in the current statement, an error free intermediate code is generated for this parsed statement. Intermediate code matrix is available in CODEF file. At the end of the source program file, CFILE, this routine passes control to the main program.

5.3 GETSYM

This routine scans the input string until a token is found, so as to be identified as lexical analyzer. This routine is called whenever there is need of a token. At the end of source program file, CFILE, it returns TAB=0 which is a signal for the end of the source program. This routine limits the length of the identifier at 10 characters. In case it exceeds its length of 10 characters, an immediate action is taken to truncate the identifier and a warning alongwith the listing of the source file is printed out.

Literals are also dealt with similarly except that the literal length for integer is taken to be 10 characters and that for reals is 15 characters. This routine also checks every identifier for a possibility of its being keyword. Hence, every identifier token is compared with the entries in the keywords table. If a match is found code for keyword is returned, otherwise a code for an identifier is returned. Similarly code for numerical literals, operators and delimeters are returned.

5.4 EXPR

This routine scans the input string until a syntactically correct expression is found. Intermediate code for the parsed expression is then generated and stored in the intermediate code file CODEF. This routine is provided with a pointer to the input string from where expression is to be passed. On return, this pointer is at the end of the expression. It also returns the pointer to a temporary location, where the value of the parsed expression would be available.

Ituses a stack S and an incoming symbol R. Depending on the top of stack and incoming symbol a particular routineis executed. In case, top of the stack and incoming symbols are

not matching according to the syntax of the arithmetic expression, described in the text, an error condition is detected. An error code of four characters is printed, first two characters showing the top of the stack and last two the in coming symbol. One extra character 'E' is also printed with the error code which indicate that error has occured during parsing of the arithmetic expression. Once an error has been detected this routine transfers control to the STATE.

5.5 IO

This routine is called by STATE, for generating intermediate code for input/output statements. This routine is provided with a pointer to the input string from where, identifier list for input/output statement, is to be parsed. This routine scanes the input string until syntactically correct input/output statement is found. During this process it generates the intermediate code for the parsed statement, and on detection of some error, error condition is set and this routine transfers control to STATE, where it prints error message.

5.6 GENCOD

This routine is called to generate the intermediate code in proper format. This routine is called by STATE.

The input to this routine is already provided by STATE by means of global variables. Intermediate code is available in CODEF.

5.7 BLCKST

This routine is called by STATE. It is called for making a new entry in the block list. The input of this routine is a pointer to the current block entry and block identification. With this information it creates a new entry in the block list. The fields in the newly created entry are filled as described in chapter III.

5.8 CLBLCK

This routine is called by STATE. It is called whenever STATE finds end of a block. At this time all the entries belonging to the current block are transferred to the other end of the symbol table. If the closed block is a procedure block, all the label variables are local to this procedure and hence, checked for their declaration within this procedure. If these are not declared error message is recorded in the error message file. After closing a block, current block is next surrounded block. All the variables which are not local to the closed block are now local to the current block.

5.9 SEARCH

This routine is called for searching an identifier in the current block. It initializes DMCRBL variable with CVRRBL and then transfers control to a routine SRDMBL. It is nothing to do more because all searching work is done by SRDMBL. When SRDMBL transfers control to SEARCH, it transfers control back to the called routine.

5.10 SEARCHB

This routine does the same work, what is done by SEARCH. Instead of searching an identifier within the current block, it searches it in all the surrounded blocks.

5.11 SRDMBL

This routine is called by sEARCH and SEARCHB. The input to this routine is the block number within which an identifier is searched. It uses block list for that block. Block list provided it the number of entries in the given block and a pointer to the symbol table for that block. With this information it makes a linear search in the symbol table. If identifier is found, entry point to the symbol table for this identifier is transferred to the called program. If identifier is not found code=0 set and control is transferred to the calling program.

5.12 INSERT

This routine is used to insert a given identifier into a current block. Given identifier is inserted at the top of the stack. Inserted variable type is set to zero (which is a signal that identifier type is unknown) and its declaration bit is also sent to zero. In the block list entries for the current block are also modified. For example number of entries field is incremented. After insertion, control is transferred to the calling program.

5.13 OPPROC

This routine makes a new entry in block list. Block identification field is set to one(which tells that block represented by this block is a procedure). Number of entries field is set to zero. Pointer to the top of the symbol table is stored in pointer to the symbol table field. Other entries of block list are also modified. Then it transfers control to the PROCID, where parsing of procedure head is done. When it receives control from the PROCID then it transfer control to STATE.

5.14 PROCID

This routine is called by PPROC. It parses the input

string until the procedure head is defined. During this process, it calls DCLREN to parse declaration statements appearing in the procedure head. All the declared variables are stored on the top of the stack and various block list entries are also modified. During this process if some error is detected it is printed out.

5.15 OPBLCK

This routine is called by STATE to open a begin block. Its purpose is to create a block list exactly in the same manner as is done by OPPROC. It calls DCLREN routine for parsing declaration statements.

5.16 DCLREN

This routine is called by OPBLCK and OPPROC routines. This routine is provided with a pointer to the input string from where declaration statement is to be parsed. All the variables occuring in a declaration statement are placed on the top of the symbol table and block list entries are also modified. When there is no more declaration statement, it transfers control to the calling program.

5.17 NEXTEN

This routine is called by STATE routine. It is

provided with a pointer to the input string, and it scans the input string, until, end of the statement is found. This is done to ignore the current statement.

Besides the above described routines, there are a few more routines which perform other jobs to complete the process of compilation.

REFERENCES

Aho,A.V. and Ullman,J.D. (1979) Principles of Compiler Design, Addison-Wesley, Reading,Mass.

Aho,A.V., Hopcraft,J.E. and Ullman,J.D.(1974). The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading, Mass.

Aho,A.V. and Ullman,J.D.(1972) The theory of Parsing, Translating and Compiling, Vol.I: Parsing, Prentice-Hall, Englewood Cliffs,N.J.

Aho,A.V. and Ullman,J.D.(1973) The Theory of Parsing, Translating and Compiling, Vol.II: Compiling, Printice-Hall, Englewood Cliffs,N.J.

Bauer,F.L. and Eichel,J.(1974) Compiler Construction: An Advanced Course, Splinger-Verlog, New York,N.Y.

Donovan,J.J.(1972) Systems Programming: McGraw-Hill, KogaKusha Ltd. Japan.

Gries,D.(1971) Compiler Construction for Digital Computer, John Wiley and Sons, New York, N.Y.

Harning,J.J.(1974). "What the Compiler should tell the user", in Bauer and Eickel (1974) pp.525-48.

Hughes,J.K. (1979). PL/I Structured Programming: John Wiley & Sons, New York, N.Y.

Knuth,D.E.(1977). The Art of Computer Programming, Vol.III: Sorting and Searching, Addison-Wesley, Reading Mass.

Naur P (ed.) (1963). 'Revised report on the Algorithmic Language ALGOL 60,' Comm. ACM6:1, 1-77.

Pyster,A.B.(1980). Compiler Design and Construction. Van Nostrand Reinhold Co. New York. N.Y.

Rosen,S.(1967). Programming Systems and Languages, McGraw-Hill, New York, N.Y.

APPENDIX

COMPILER LISTING

```
&LAX    T=00004 IS ON CR00013 USING 00300 BLKS R=0000

0001
0002    ALGOLS: PROCEDURE OPTIONS(MAIN);
0003          DCL
0004               CARD           CHAR(80),
0005               CHR(72)        CHAR(1),
0006               IDNT           CHAR(9),
0007               IDNTS          CHAR(9),
0008               IDNT1          CHAR(72),
0009               INTGR          CHAR(15),
0010               INTGR1         CHAR(72),
0011               SYMB(1024)     CHAR(9),
0012               TYPE(1024)     FIXED BINARY(15),
0013               DCLR(1024)     FIXED BINARY(15),
0014               ADDSS(1024)    FIXED BINARY(15),
0015               UCODE(1024)    FIXED BINARY(15),
0016               BLCKEN(128)    FIXED BINARY(15),
0017               BLCKPT(128)    FIXED BINARY(15),
0018               BLCKLV(128)    FIXED BINARY(15),
0019               BLCKID(128)    FIXED BINARY(15),
0020               CODES          FIXED BINARY(15),
0021               DMCRBL         FIXED BINARY(15),
0022               CURRBL         FIXED BINARY(15),
0023               TOPEL          FIXED BINARY(15),
0024               LASTEL         FIXED BINARY(15),
0025               LASTBL         FIXED BINARY(15),
0026               REALS          FIXED BINARY(15),
0027               DCODE          FIXED BINARY(15),
0028               QTURPLF        FIXED BINARY(15),
0029               IFLAG          FIXED BINARY(15),
0030               TAB            FIXED BINARY(15),
0031               CODE           FIXED BINARY(15),
0032               I              FIXED BINARY(15),
0033               J              FIXED BINARY(15),
0034               K              FIXED BINARY(15),
0035               L              FIXED BINARY(15),
0036               M              FIXED BINARY(15),
0037               N              FIXED BINARY(15),
0038               T              FIXED BINARY(15),
0039               Z              FIXED BINARY(15),
0040               N              FIXED BINARY(15),
0041               CA             FIXED BINARY(15),
0042               LITNO          FIXED BINARY(15),
0043               NN             FIXED BINARY(15),
0044               TT2            FIXED BINARY(15),
0045               BB2            FIXED BINARY(15),
0046               CC2            FIXED BINARY(15),
0047               C1             FIXED BINARY(15),
0048               C2             FIXED BINARY(15),
0049               KUTEMPL        FIXED BINARY(15),
0050               KUTAB          FIXED BINARY(15),
0051               B1             FIXED BINARY(15),
0052               B2             FIXED BINARY(15),
0053               T1             FIXED BINARY(15),
0054               T2             FIXED BINARY(15),
0055               OP             CHAR(3),
0056               BLNK1          CHAR(72)INITIAL((72)' '),
0057               BLNK           CHAR(6)INITIAL((6)' '),
0058               CC(10)         FIXED BINARY(15)
0059               INITIAL        (10,6,12,8,2,4,4,2,4,2),
0060               CCD(10)        FIXED BINARY(15)
0061               INITIAL        (4,8,2,6,12,10,10,12,10,12),
0062               B(14,14)       FIXED BINARY(15)
0063               INITIAL        (5,0,0,0,0,0,0,0,0,0,0,0,0,30,
0064                              1,0,1,1,1,1,0,0,0,0,0,0,0,30,
```

```
0065                                0,9,0,0,0,0,0,0,0,0,0,0,0,0,
0066                                0,0,20,0,0,0,0,0,0,0,0,0,0,31,
0067                                34,18,18,18,35,34,18,18,18,18,18,18,18,30,
0068                                21,23,21,21,21,21,23,23,23,23,23,23,23,30,
0069                                1,0,1,1,1,1,0,0,0,0,0,0,0,30,
0070                                1,0,1,1,1,1,0,0,0,0,0,0,0,30,
0071                                1,0,1,1,1,1,0,0,0,0,0,0,0,30,
0072                                22,23,25,25,33,27,3,28,28,8,23,23,29,30,
0073                                0,0,0,0,0,0,0,0,0,19,0,11,16,30,
0074                                5,0,7,32,36,26,3,24,24,8,0,0,29,0,
0075                                6,0,6,6,6,6,0,0,0,0,0,0,4,0,
0076                                10,17,10,10,10,10,2,13,14,15,12,0,0,30),
0077                    CODEM(3,5)  FIXED BINARY(15)
0078                    INITIAL     (9,4,9,7,9,
0079                                8,1,2,9,3,
0080                                9,5,9,6,9),
0081                    A(14,14)    FIXED BINARY(15)
0082                    INITIAL     (2,3,3,8,8,19,17,0,15,
0083                                1,1,1,1,1,1,0,1,0,
0084                                5,4,4,8,8,19,12,5,15,
0085                                20,4,4,8,8,19,12,20,18,
0086                                6,6,6,9,9,21,13,6,16,
0087                                7,7,7,9,9,21,13,7,16,
0088                                0,4,4,8,8,19,12,14,15,
0089                                11,11,11,10,10,11,0,11,0),
0090                TEXT0 CHAR(13) INITIAL('TRUNCATED TO '),
0091                TEXT1 CHAR(24) INITIAL('THE RELATIONAL OPERATOR''('),
0092                TEXT2 CHAR(8) INITIAL(''')IN LINE'),
0093                TEXT3 CHAR(37) INITIAL
0094                ('DOES NOT MAKE A SENSIBLE COMBINATION'),
0095                TEXT4 CHAR(10) INITIAL('CHARECTER'''),
0096                TEXT5 CHAR(37) INITIAL('DOES NOT BELONG TO CHARARTER SET '),
0097                TEXT6 CHAR(23) INITIAL('THE FIXED POINT NUMBER '),
0098                TEXT7 CHAR(56) INITIAL
0099                ('EXCEEDS THE MAXIMUM FIXED,FLOAT POINT LENGTH WHICH IS 2'),
0100                TEXT8 CHAR(18) INITIAL('THE FLOATING POINT'),
0101                TEXT9 CHAR(63) INITIAL
0102                ('EXCEEDS THE MAXIMUM FLOATING POINT EXPONENT LENGTH WHICH I
0103                TEXT10 CHAR(26) INITIAL('LABEL IS NOT AN IDENTIFIER'),
0104                TEXT11 CHAR(33) INITIAL('VARIABLE ENTRY USED AS LABEL'),
0105                TEXT12 CHAR(22) INITIAL('STATEMENT SYNTAX ERROR'),
0106                TEXT13 CHAR(25) INITIAL('ERROR DETECTED AT COLUMN '),
0107                TEXT14 CHAR(11) INITIAL('THE INTEGER'),
0108                TEXT15 CHAR(47) INITIAL
0109                      ('EXCEEDS THE MAXIMUM INTEGER LENGTH WHICH IS 10 '),
0110          ER0    CHAR(25)INITIAL('ILLEGAL MATCH OF KEYWORDS'),
0111          ER2    CHAR(17)INITIAL('IDNT IS NOT A LABEL'),
0112          ER4    CHAR(29)INITIAL('IDNT IS NOT A VARIABLE'),
0113          ER16   CHAR(32)INITIAL('VARIABLE:ENTRY IS USED AS LABEL'),
0114          ER18   CHAR(50)INITIAL('PROCEDURE DELARATION IN STATEMENT'),
0115          ER20   CHAR(20)INITIAL('ILLEGAL ELSE CLAUSE'),
0116          ER22   CHAR(44)INITIAL
0117                ('LOGICAL END OF PROGRAM PRIOR TO PHYSICAL END'),
0118          ER23   CHAR(23)INITIAL('ILLEGAL BEGIN STATMENT'),
0119          ER24   CHAR(36)INITIAL('READ WRITE ST. NOT RECOGNIZED'),
0120          ER25   CHAR(20)INITIAL('INVALID STATEMENT'),
0121          ER26   CHAR(18)INITIAL('ST. NOT REC.'),
0122          ER28   CHAR(21)INITIAL('IDNT. IS NOT AN ENTRY'),
0123          ER29   CHAR(25)INITIAL('VARIABLE IS NOT AN ENTRY),
0124          ER32   CHAR(31)INITIAL('SYNTAX ERROR IN ELSE STATEMENT'),
0125          ER33   CHAR(37)INITIAL('ILLEGAL END WITH CLOSURE TO PROCEDURE'),
0126          ER34   CHAR(25)INITIAL('ILLEGAL BLOCK DEFINITION'),
0127          ER35   CHAR(30)INITIAL('ERROR IN OPENING A PROCEDURE'),
0128          ER36   CHAR(30)INITIAL('ERROR ER36 IN    A PROCEDURE'),
0129          ERTAB3  CHAR(20)INITIAL('INVALID KEYWORDS'),
0130          EROPPR CHAR(27)INITIAL('ILLEGAL OPENING OF A BLOCK'),
```

```
0131          EROPPR0 CHAR(23)INITIAL('ILLEGAL PROCEDURE NAME'),
0132          EROPPR3 CHAR(23)INITIAL('ILLEGAL PROCEDURE ENTRY'),
0133          DCLR0 CHAR(21)INITIAL('ILLEGAL DCL STATEMENT'),
0134          DCLR1 CHAR(18)INITIAL('ILLEGAL DELIMITER');
0135      DCL DELIM(13) CHAR(1)
0136          INITIAL('(','+','-','/','*',')',',',';',':','^','(','=',')');
0137      DCL KEYWRDS(12) CHAR(9) INITIAL
0138          ('IF','THEN','ELSE','PROCEDURE','BEGIN','GOTO','READ','WRITE',
0139          'END','COMMENT','REAL','INTEGER') ;
0140      DCL  BLCKF FILE OUTPUT ENV('FO=SQ,RT=Z,FL=80'),
0141           TEMPLF FILE OUTPUT ENV('FO=SQ,RT=Z,FL=30'),
0142           SYMBLF FILE OUTPUT ENV('FO=SQ,RT=Z,FL=30'),
0143           CODEF  FILE OUTPUT ENV('FO=SQ,RT=Z,FL=50'),
0144           LFILE  FILE OUTPUT ENV('FO=SQ,RT=Z,FL=50'),
0145           EFILE  FILE OUTPUT ENV('FO=SQ,RT=Z,FL=50'),
0146           PFILE  FILE OUTPUT ENV('FO=SQ,RT=Z,FL=132'),
0147           CFILE  FILE INPUT ENV('FO=SQ,RT=Z,FL=80');
0148      OPEN FILE(LFILE),FILE(EFILE),FILE(CFILE),FILE(CODEF),
0149           FILE(PFILE),FILE(TEMPLF),FILE(BLCKF),FILE(SYMBLF);
0150          ON ENDFILE(CFILE)GO TO EOJ;
0151          CURRBL=0;
0152          TOPEL=0;
0153          LASTBL=0;
0154          LASTEL=1024;
0155          CODE=6;
0156          LITNO=0;
0157          KUTEMPL=0;
0158          IFLAG=0;
0159          KUTAB=0;
0160          QTURPLE=1;
0161          CALL GTCARD;
0162          CALL GETSYM;
0163          CALL BLCKST;
0164               CALL STATE;
0165               GOTO EOJ;
0166           GTCARD:PROCEDURE;
0167               CA=CA+1;
0168               GET FILE(CFILE)EDIT(CARD)(A(80));
0169               PUT FILE(PFILE)EDIT(CA,CARD)(SKIP,X(7),F(6),A(80));
0170               DO I=1,72;
0171                CHR(I)=SUBSTR(CARD,I,1);
0172               END;
0173               I=1;
0174               J=1;
0175               RETURN;
0176               END GTCARD;
0177           GETSYM:PROCEDURE;
0178            INIT:
0179               IF I>72 THEN CALL GTCARD;
0180               J=I;
0181               I=I-1;
0182               L=0;
0183           FRESH;
0184               I=I+1;
0185               IF L=0 THEN DO;
0186                             IF CHR(I)=' 'THEN DO;
0187                                              I=I+1;
0188                                              GO TO INIT;
0189                                             END;
0190                              IF CHR(I)<'A' THEN GO TO NUMERIC;
0191                             END;
0192               IF CHR(I)=' 'THEN GO TO INSTALID;
0193               IF CHR(I)<'A'THEN GO TO SPLACHAR;
0194               IF CHR(I)<='I'THEN GO TO INCREMENT;
0195               IF CHR(I)<'J'THEN GO TO SPLACHAR;
0196               IF CHR(I)<='R'THEN GO TO INCREMENT;
```

```
0197              IF CHR(I)<'S'THEN GO TO SPLACHAR ;
0198              IF CHR(I)<='Z'THEN GO TO INCREMENT;
0199              IF L=0 THEN GO TO NUMERIC;
0200              IF CHR(I)<'0'THEN GO TO SPLACHAR;
0201              IF CHR(I)<='9'THEN GO TO INCREMENT;
0202             GO TO INSTALID;
0203         INCREMENT:
0204              L=L+1;
0205              IF L>=9 THEN GO TO IDENTERR;
0206         NXTACHAR:
0207              IF I<72 THEN GO TO FRESH;
0208              GO TO INSTALID;
0209          SPLACHAR:
0210              IF L=0 THEN GO TO NUMERIC;
0211          INSTALID:
0212              PUT STRING(IDNT)EDIT(BLNK)(A(9));
0213              J1=J+L-1;
0214              PUT STRING(IDNT)EDIT((CHR(K)DO K=J TO J1))(A(1));
0215              T=1;
0216          NEXTAKEY:
0217              IF IDNT=KEYWRDS(T)THEN DO;
0218                                      TAB=3;
0219                                      CODE=T;
0220                                      RETURN;
0221                                     END;
0222              T=T+1;
0223              IF T<=12 THEN GO TO NEXTAKEY;
0224                 TAB=1;
0225                 RETURN;
0226          NUMERIC:
0227              LXPNT=0;
0228              F1=0;
0229              F2=0;
0230              ESIGN=0;
0231              Z=0;
0232          DIGIT:
0233              IF CHR(I)<'0'THEN GO TO REALS;
0234              IF CHR(I)<='9'THEN DO;
0235                                  Z=Z+1;
0236                                  I=I+1;
0237                                  GO TO DIGIT;
0238                                 END;
0239          REALS:
0240              IF CHR(I)='.'THEN DO;
0241                                  F1=1;
0242                                 REALFXDP:
0243                                  I=I+1;
0244                                  Z=Z+1;
0245                                  IF CHR(I)<'0'THEN GOTO CHECHAE;
0246                                  IF CHR(I)<='9'THEN GOTO REALFXDP;
0247                                 END;
0248          CHECKAE:
0249               IF CHR(I)='E'THEN DO;
0250                                   I=I+1;
0251                                   Z=Z+1;
0252                                   IF CHR(I)='-'THEN DO;
0253                                                     ESIGN=1;
0254                                                     I=I+1;
0255                                                     Z=Z+1;
0256                                                     GO TO REALFLTP;
0257                                                    END;
0258                                   IF CHR(I)='+'THEN DO;
0259                                                     I=I+1;
0260                                                     Z=Z+1;
0261                                                    END;
0262                        REALFLTP:
```

```
0263                        PXPNT=I;
0264                      REALFLTPN:
0265                        IF CHR(I)<'0'THEN GO TO INSTALLI;
0266                        IF CHR(I)<='9'THEN DO;
0267                                          F2=1;
0268                                          Z=Z+1;
0269                                          I=I+1;
0270                                          LXPNT=LXPNT+1;
0271                                          GO TO REALFLTPN;
0272                                         END;
0273                                END;
0274      INSTALLI:
0275            IF Z=0 THEN GO TO DELIMITER;
0276            IF F2=1 THEN GO TO INSTALFL;
0277            IF F1=1 THEN GO TO INSTALFX;
0278      INSTALIN:
0279            SCALE=1;
0280            IF Z>10 THEN GO TO INTGRREC;
0281            J1=J+Z-1;
0282            PUT STRING(INTGR)EDIT((CHR(K)DO K=J TO J1))(10 A(1));
0283            GO TO INTGPT;
0284      INTGRREC:
0285            PUT STRING(INTGR1)EDIT(BLNK1)(72 A(1));
0286            J1=J+Z-1;
0287            PUT STRING(INTGR1)EDIT((CHR(K)DO K=J TO J1))(72 A(1));
0288            PUT FILE(PFILE)EDIT(TEXT14,INTGR1,TEXT15)
0289            (SKIP,X(11),A(11),SKIP,X(11),A(72),SKIP,X(11),A(47));
0290            J=J1-9;
0291            PUT STRING(INTGR)EDIT((CHR(K)DO K=J TO J1))(10 A(1));
0292            Z=10;
0293      INTGPT:
0294            LITNO=LITNO+1;
0295            PUT FILE(LFILE)EDIT(LITNO,SCALE,Z,INTGR,ESIGN)
0296            (SKIP,F(6),X(1),F(1),F(3),X(1),A(15),X(1),F(1));
0297            TAB=2;
0298            CODE=LITNO;
0299            RETURN;
0300      INSTALFX:
0301            SCALE=2;
0302            IF Z>15 THEN GO TO FIXEDERR;
0303            J1=J+Z-1;
0304            PUT STRING(INTGR)EDIT((CHR(K)DO K=J TO J1))(15 A(1));
0305            GO TO INTGPT;
0306      FIXEDERR:
0307            PUT STRING(INTGR1)EDIT(BLNK1)(72 A(1));
0308            J1=J+Z-1;
0309            PUT STRING(INTGR1)EDIT((CHR(K)DO K=J TO J1))(72 A(1));
0310            PUT FILE(PFILE)EDIT(TEXT6,INTGR1,TEXT7)
0311            (SKIP,X(11),A(23),SKIP,X(11),A(72),SKIP,X(11),A(56));
0312            J=J1-14;
0313            PUT STRING(INTGR)EDIT((CHR(K)DO K=J TO J1))(15 A(1));
0314            Z=15;
0315            GO TO INTGPT;
0316      INSTALFL:
0317            SCALE=3;
0318            IF Z>15 THEN GO TO FLOATERR;
0319            IF LXPNT>2 THEN DO;
0320                       J1=PXPNT+1;
0321                       PUT STRING(INTGR)EDIT
0322                       ((CHR(K)DO K=J TO J1))(15 A(1));
0323                       PUT STRING(INTGR1)EDIT(BLNK1)(72 A(1));
0324                       J1=J+Z-1;
0325                       PUT STRING(INTGR1)EDIT
0326                       ((CHR(K) DO K=J TO J1))(72 A(1));
0327                       PUT FILE(PFILE)EDIT(TEXT8,INTGR1,TEXT9)
0328                       (SKIP,X(11),A(23),SKIP,X(11),
```

```
0329                    A(72)),SKIP,X(11),A(63));
0330                     GO TO INTGPT;
0331                   END;
0332          J1=J+Z-1;
0333          PUT STRING(INTGR)EDIT((CHR(K)DO K=J TO J1))(15 A(1));
0334          GO TO INTGPT;
0335     FLOATERR:
0336          PUT STRING(INTGR1)EDIT(BLNK1)(72 A(1));
0337          J1=J+Z-1;
0338          PUT STRING(INTGR1)EDIT((CHR(K)DO K=J TO J1))(72 A(1));
0339          PUT FILE(PFILE)EDIT(TEXT8,INTGR1,TEXT7)
0340          (SKIP,X(11),A(18),SKIP,X(11),A(72),SKIP,X(11),A(56));
0341          IF LXPNT>2 THEN DO;
0342                          J1=PXPNT+1;
0343                          J=J1-14;
0344                          PUT STRING(INTGR)EDIT
0345                          ((CHR(K)DO K=J TO J1))(15 A(1));
0346                          GO TO INTGPT;
0347                         END;
0348          J1=PXPNT+LXPNT-1;
0349          J=J1-14;
0350          PUT STRING(INTGR)EDIT((CHR(K)DO K=J TO J1))(15 A(1));
0351          Z=15;
0352        GO TO INTGPT;
0353   DELIMITER:
0354        D=1;
0355   NXTADEL:
0356        IF CHR(I)=DELIM(D) THEN DO;
0357                             TAB=4;
0358                             CODE=D;
0359                             GO TO SECONDACH;
0360                            END;
0361         D=D-1;
0362         IF D<=13 THEN GO TO NXTADEL;
0363         GO TO NOACHAR;
0364   SECONDACH:
0365       I=I+1;
0366       IF I>72 THEN RETURN;
0367       IF D<9 THEN RETURN;
0368       D=1;
0369   NEXTADELM:
0370       IF CHR(I)=DELIM(D) THEN DO;
0371                          IF D<11 THEN GOTO COMBERR;
0372                          CODE1=D-10;
0373                          CODE2=CODE-8;
0374                          CODE3=CODEM(CODE1,CODE2);
0375       IF CODE3>8 THEN GO TO COMBERR;
0376                  ELSE DO;
0377                       TAB=4;
0378                       CODE=CODE3+13;
0379                       I=I+1;
0380                       RETURN;
0381                      END;
0382                              END;
0383            ELSE DO;
0384                       D=D+1;
0385                       IF D<=13 THEN GO TO NEXTADELM;
0386                       RETURN;
0387                       END;
0388   IDENTERR:
0389         I=I+1;
0390   AGAIN:
0391       IF I<=72 THEN DO;
0392            IF CHR(I)<'A' THEN GO TO RCVRIDNT;
0393            IF CHR(I)<='I' THEN GO TO INC;
0394            IF CHR(I)< 'J' THEN GO TO RCVRIDNT;
```

```
0395                  IF CHR(I)<='R' THEN GO TO INC;
0396                  IF CHR(I)<'S' THEN GO TO RCVRIDNT;
0397                  IF CHR(I)<='Z' THEN GO TO INC;
0398                  IF CHR(I)<'0' THEN GO TO RCVRIDNT;
0399                  IF CHR(I)<='9' THEN GO TO INC;
0400                              END;
0401          RCVRIDNT:
0402             J1=J+8;
0403             PUT STRING(IDNT)EDIT(BLNK)(A(9));
0404             PUT STRING(IDNT)EDIT((CHR(K)DO K=J TO J1))(9 A(1));
0405             PUT STRING(IDNT1)EDIT(BLNK1)(A(72));
0406             J1=J+L-1;
0407             PUT STRING(IDNT1)EDIT((CHR(K)DO K=J TO J1))(72 A(1));
0408             PUT FILE(PFILE)EDIT(IDNT1,TEXT0,IDNT)
0409             (SKIP,X(11),A(30),X(1),A(13),X(1),A(9));
0410           TAB=1;
0411           RETURN;
0412       INC:
0413           I=I+1;
0414           L=L+1;
0415           GO TO AGAIN;
0416       COMBERR:
0417           J1=I-1;
0418           PUT FILE(PFILE)EDIT(TEXT1,CHR(J1),CHR(I),TEXT2,CA,TEXT3)
0419           (SKIP,X(11),A(24),A(1),A(1),A(8),F(3),A(37));
0420           I=I+1;
0421           IFLAG=1;
0422           RETURN;
0423       NOACHAR:
0424           PUT FILE(PFILE)EDIT(TEXT4,CHR(I),TEXT2,CA,TEXT5)
0425                       (SKIP,X(11),A(8),A(1),A(8),F(6),A(37));
0426           I=I+1;
0427           IFLAG=1;
0428           RETURN;
0429           END GETSYM;
0430        STATE: PROCEDURE;
0431         DCL
0432                S(32)     FIXED BINARY(15),
0433                UTAB(32)  FIXED BINARY(15),
0434                UST       FIXED BINARY(15),
0435                USC       FIXED BINARY(15),
0436                USB       FIXED BINARY(15),
0437                GOOUT     FIXED BINARY(15),
0438                I         FIXED BINARY(15),
0439                U         CHAR(1);
0440           S(1)=1;
0441           U=' ';
0442           I=1;
0443        NEXT:
0444          IF TAB=0 THEN DO;
0445                           GOOUT=B(1,S(I));
0446                           GOTO OUT;
0447                           END;
0448          IF TAB=1 THEN DO;
0449                           GOOUT=B(14,S(I));
0450                           GOTO OUT;
0451                           END;
0452          IF TAB=2 THEN DO;
0453                           GOOUT=B(14,S(I));
0454                           GOTO OUT;
0455                           END;
0456          IF TAB=3 THEN DO;
0457                           IF CODE>9 THEN DO;
0458                            PUT FILE(EFILE)EDIT(CA,ERTAB3)
0459                            (SKIP,X(1),F(6),X(1),A(50));
0460                            GO TO NEXTS;
```

```
0461                        END;
0462                        GOOUT=B(CODE+1,S(I));
0463                        GOTO OUT;
0464                        END;
0465        IF TAB=4 THEN DO;
0466                        IF CODE =8  THEN DO;
0467                                         GOOUT=B(12,S(I));
0468                                         GO TO OUT;
0469                                        END;
0470                        IF CODE=21THEN DO ;
0471                                       GOOUT=B(13,S(I));
0472                                       GO TO OUT;
0473                                       END;
0474                        GOOUT=B(11,S(I));
0475                        GOTO OUT;
0476                        END;
0477      OUT;
0478            IF GOOUT>9 THEN GO TO GOOUT10;
0479            IF GOOUT=0 THEN DO;
0480                             GO TO ERRRR;
0481                             END;
0482            IF GOOUT=1 THEN DO;
0483                            IF S(I)=6 THEN
0484                               IF U='S' THEN U=' ';
0485                             IF U^=' ' THEN DO;
0486                                            GO TO ERRRR;
0487                                            END;
0488                              I=I+1;
0489                              S(I)=CODE+1;
0490                              GO TO NEXTS;
0491                        END;
0492        IF GOOUT= 2 THEN   DO;
0493                             IF U^=' '!TAB^=1 THEN  DO;
0494                              I=I-1;
0495                              U=' ';
0496                              PUT FILE(EFILE)EDIT(CA,ER2)
0497                             (SKIP,X(1),F(6),X(1),A(50));
0498                              GO TO NEXTS;
0499                             END;
0500                             U='L';
0501                             CALL SEARCH;
0502                             IF CODE=0 THEN CALL INSERT;
0503                             IF TYPE(DCODE)>1 THEN DO ;
0504                                 PUT FILE(EFILE)EDIT(CA,ER2)
0505                                 (SKIP,X(1),F(6),X(1),A(50));
0506                                 U=' ';
0507                                 I=I-1;
0508                                 GO TO NEXTS;
0509                                 END;
0510                             TYPE(DCODE)=1;
0511                             UST=1;
0512                             USC=CODE;
0513                             USB=CURRBL;
0514                             GO TO NEXTS;
0515                        END;
0516        IF GOOUT= 3 THEN   DO;
0517                             IF U^='L' THEN GO TO ERRRR;
0518                             U='S';
0519                             I=I-1;
0520                             OP='BRN';
0521                             T1=0;
0522                             B1=1;
0523                             T2=UST;
0524                             B2=USB;
0525                             C1=15;
0526                             C2=USC;
```

```
0527                                CALL GENCOD;
0528                                GO TO NEXT;
0529                            END;
0530          IF GOOUT= 4 THEN    DO;
0531                                IF U^=' ' THEN GOTO ERRRR;
0532                                U='V';
0533                                I=I-1;
0534                                IF UST^=1 THEN GO TO ERRRR;
0535                                PUT STRING(IDNT)EDIT(IDNTS)(A(9));
0536                                CALL SEARCHB;
0537                                IF CODE=0 THEN CALL INSERT;
0538                                IF TYPE(DCODE)<1 THEN TYPE(DCODE)=2;
0539                                IF TYPE(CODE)<2 THEN DO;
0540                                   PUT FILE(EFILE)EDIT(CA,ER4)
0541                                   (SKIP,X(1),F(6),X(1),A(50));
0542                                   U=' ';
0543                                   CALL NEXTEN;
0544                                   IF TAB=3 THEN GO TO NEXT;
0545                                   GO TO NEXTS;
0546                                END;
0547                                USC=CODE;
0548                                USB=DMCRBL;
0549                                CODE=21;
0550                                GO TO NEXT;
0551                            END;
0552          IF GOOUT= 5 THEN    DO;
0553                                IF U^='S' THEN
0554                                   IF U^=' 'THEN GO TO ERRRR;
0555                                IF TAB=0 THEN RETURN;
0556                                GO TO NEXTS;
0557                                END;
0558          IF GOOUT=6 THEN DO;
0559                                IF U^='V' THEN GO TO ERRRR;
0560                                I=I+1;
0561                                S(I)=10;
0562                                GO TO NEXTS;
0563                            END;
0564          IF GOOUT= 7 THEN    DO;
0565                                IF U^='S' THEN DO;
0566                                                  GO TO ERRRR;
0567                                                  END;
0568                                S(I)=14;
0569                                GO TO NEXTS;
0570                            END;
0571          IF GOOUT= 8 THEN    DO;
0572                                IF U^='E' THEN GO TO ERRRR;
0573                                I=I-1;
0574                                U='S';
0575                                OP='MOV';
0576                                T1=1;
0577                                C1=USC;
0578                                B1=USB;
0579                                CALL GENCOD;
0580                                GO TO NEXT;
0581                            END;
0582          IF GOOUT= 9 THEN    DO;
0583                                IF U^='B' THEN DO;
0584                                                  GO TO ERRRR;
0585                                                  END;
0586                                S(I)=3;
0587                                U=' ';
0588                                CALL GENCOD;
0589                                GO TO NEXTS;
0590                               END;
0591    GOOUT10:
0592          IF GOOUT=10 THEN    DO;
```

```
0593                              IF S(I)=6 THEN
0594                                 IF U^='S'THEN U=' ';
0595                              IF U^=' ' THEN DO;
0596                                             GO TO ERRRR;
0597                                             END;
0598                              IF TAB^=1 THEN GO TO ERRRR;
0599                              I=I+1;
0600                              S(I)=13;
0601                              UST=1;
0602                              PUT STRING(IDNTS)EDIT(IDNT)(A(9));
0603                              GO TO NEXTS;
0604                           END;
0605           IF GOOUT=11 THEN   DO;
0606                              IF U^=' ' THEN GOTO ERRRR;
0607                              U=' ';
0608                              IF CODE(11!CODE>19 THEN GO TO ERRRR;
0609                              CCCC=CODE-10;
0610                              CNCD=CC(CCCC);
0611                              S(I)=2;
0612                              I=I+1;
0613                              S(I)=11;
0614                              GO TO NEXTS;
0615                           END;
0616           IF GOOUT=12 THEN   DO;
0617                              JF U^=' ' THEN GOTO ERRRR;
0618                              I=I-1;
0619                              U='B';
0620                              IF TAB=1 THEN DO;
0621                              CALL SEARCH;
0622                              IF CODE=0 THEN CALL INSERT;
0623                              IF TYPE(DCODE)<1 THEN TYPE(DCODE)=2;
0624                              IF TYPE(DCODE)<2 THEN DO;
0625                               PUT FILE(EFILE)EDIT(CA,ER4)
0626                               (SKIP,X(1),F(6),X(1),A(50));
0627                               U=' ';
0628                               GO TO NEXTS;
0629                                  END;
0630                                END;
0631                              T1=UST;
0632                              C1=USC;
0633                              B1=USB;
0634                              T2=TAB;
0635                              C2=CODE;
0636                              B2=DBCRBL;
0637                              OP='CMP';
0638                              CALL GENCOD;
0639                              OP='BRN';
0640                              T1=0;
0641                              T2=5;
0642                              C1=CNCD;
0643                             .KUTEMPL=KUTEMPL+1;
0644                              C2=KUTEMPL;
0645                              KUTAB=KUTAB+1;
0646                              UTAB(KUTAB)=KUTEMPL;
0647                              B1=1;
0648                              B2=1;
0649                              GO TO NEXTS;
0650                           END;
0651           IF GOOUT=13 THEN   DO;
0652                              IF TAB^=1 THEN DO;
0653                                              GO TO ERRRR;
0654                                             END;
0655                              OP='GET';
0656                              CALL IO;
0657                              I=I-1;
0658                              IF IFLAG=1 THEN DO;
```

```
0659                                        GO TO ERRRR;
0660                                        END;
0661                          U='S';
0662                          GO TO NEXT;
0663                       END;
0664          IF GOOUT=14 THEN   DO;
0665                          IF TAB^=1 THEN DO;
0666                                        GO TO ERRRR;
0667                                        END;
0668                          OP='PUT';
0669                          CALL IO ;
0670                          I=I-1;
0671                          IF IFLAG=1 THEN DO;
0672                                        GO TO ERRRR;
0673                                        END;
0674                          U='S';
0675                          GO TO NEXT;
0676                       END;
0677          IF GOOUT=15 THEN   DO;
0678                          CALL EXP;
0679                          IF IFLAG=1 THEN DO;
0680                                        GO TO ERRRR;
0681                                        END;
0682                          U='E';
0683                          T2=TT2;
0684                          C2=CC2;
0685                          B2=BB2;
0686                          GO TO NEXT;
0687                       END;
0688            IF GOOUT=16 THEN DO;
0689                          IF U^=' ' THEN DO;
0690                                        GO TO ERRRR;
0691                                        END;
0692                          I=I-1;
0693                          IF CODE^=9 THEN DO;
0694                             PUT FILE(EFILE)EDIT(CA,ER16)
0695                             (SKIP,X(1),F(6),X(1),A(50));
0696                             CALL NEXTEN;
0697                             GOTO NEXT;
0698                             END;
0699                           PUT STRING(IDNT)EDIT(IDNTS)(A(9));
0700                           CALL SEARCH;
0701                           IF CODE=0 THEN CALL INSERT;
0702                           IF TYPE(DCODE))1 THEN DO;
0703                              PUT FILE(EFILE)EDIT(CA,ER16)
0704                              (SKIP,X(1),F(6),X(1),A(50));
0705                              GOTO NEXTS;
0706                              END;
0707                           TYPE(DCODE)=1;
0708                           DCLR(DCODE)=1;
0709                           ADDSS(DCODE)=QTURPLE;
0710                           GOTO NEXTS;
0711                           END;
0712            IF GOOUT=17 THEN DO;
0713                          IF U^=' ' THEN DO;
0714                                        GO TO ERRRR;
0715                                        END;
0716                          IF TAB=1 THEN DO;
0717                             CALL SEARCH;
0718                             IF CODE=0 THEN CALL INSERT;
0719                             IF TYPE(DCODE)<1THEN TYPE(DCODE)=2;
0720                              IF TYPE(DCODE)<2 THEN DO;
0721                              PUT FILE(EFILE)EDIT(CA,ER16)
0722                              (SKIP,X(1),F(6),X(1),A(50));
0723                              GO TO NEXTS;
0724                              END;
```

```
0725                                    END;
0726                                  UST=TAB;
0727                                  USC=CODE;
0728                                  USB=DMCRBL;
0729                                  S(I)=12;
0730                                  GOTO NEXTS;
0731                                  END;
0732            IF GOOUT=18 THEN DO;
0733                                  PUT FILE(EFILE)EDIT(CA,ER18)
0734                                  (SKIP,X(1),F(6),X(1),A(50));
0735                                  I=I-1;
0736                                  GO TO NEXT;
0737                                  END;
0738    IF GOOUT=19 THEN DO;
0739                      IF CODE>3 THEN GO TO ERRRR;
0740                      CALL EXPR;
0741                      IF IFLAG=1 THEN GO TO ERRRR;
0742                      U='E';
0743                      C2=CC2;
0744                      T2=TT2;
0745                      B2=BB2;
0746                      GO TO NEXT;
0747                    END;
0748    IF GOOUT=20 THEN DO;
0749                      IF U^='S' THEN DO;
0750                                     PUT FILE(EFILE)EDIT(CA,ER20)
0751                                     (SKIP,X(1),F(6),X(1),A(50));
0752                                     I=I-1;
0753                                     GO TO NEXTS;
0754                                     END;
0755                                     S(I)=4;
0756                                     KUTEMPL=KUTEMPL+1;
0757                                     KUTAB=KUTAB  +1;
0758                                      UTAB(KUTAB)=KUTEMPL;
0759                                      OP='BRN';
0760                                      T1=0;
0761                                      T2=5;
0762                                      C1=15;
0763                                      C2=KUTEMPL;
0764                                      B1=1;
0765                                      B2=1;
0766                                      CALL GENCOD;
0767                                      U=' ';
0768                                      GO TO NEXTS;
0769                                      END;
0770    IF GOOUT=21 THEN DO;
0771                      IF S(I)=6 THEN
0772                         IF U='S' THEN U=' ';
0773                         IF U^=' ' THEN GO TO ERRRR;
0774                         I=I+1;
0775                         S(I)=6;
0776                         CALL OPBLCK;
0777                         GO TO NEXT;
0778                       END;
0779    IF GOOUT=22 THEN DO;
0780                      PUT FILE(EFILE)EDIT(CA,ER23)
0781                      (SKIP,X(1),F(6),X(1),A(50));
0782                      RETURN;
0783                      END;
0784    IF GOOUT=23 THEN DO;
0785                      PUT FILE(EFILE)EDIT(CA,ER23)
0786                      (SKIP,X(1);F(6),X(1),A(50));
0787                      I=I-1;
0788                      GO TO NEXT;
0789                      END;
0790    IF GOOUT=24 THEN DO;
```

```
0791                    IF U^='S' THEN
0792                              PUT FILE(EFILE)EDIT(CA,ER24)
0793                              (SKIP,X(1),F(6),X(1),A(50));
0794                      I=I-1;
0795                       GO TO NEXT;
0796                       END;
0797    IF GOOUT=25 THEN DO;
0798                    IF U^='S'THEN DO;
0799                              PUT FILE(EFILE)EDIT(CA,ER25)
0800                              (SKIP,X(1),F(6),X(1),A(50));
0801                              I=I-1;
0802                              GO TO NEXT;
0803                             END;
0804                    I=I-1;
0805                    UCODE(UTAB(KUTAB))=QTURPLE;
0806                    KUTAB=KUTAB-1;
0807                    GO TO NEXT;
0808                   END;
0809    IF GOOUT=26 THEN DO;
0810                    IF U^='S' THEN
0811                         PUT FILE(EFILE)EDIT(CA,ER26)
0812                         (SKIP,X(1),F(6),X(1),A(50));
0813                         U=' ';
0814                    GO TO NEXTS;
0815                   END;
0816    IF GOOUT=27 THEN DO;
0817                    IF U^='S'THEN IF U^=' ' THEN DO;
0818                                  U='S';
0819                                  PUT FILE(EFILE)EDIT(CA,ER26)
0820                                  (SKIP,X(1),F(6),X(1),A(50));
0821                                 END;
0822                     I=I-1;
0823                     CALL CLBLCK;
0824                     GO TO NEXTS;
0825                    END;
0826    IF GOOUT=28 THEN DO;
0827                    IF U ^='S' THEN
0828                       PUT FILE(EFILE)EDIT(CA,ER28)
0829                       (SKIP,X(1),F(6),X(1),A(50));
0830                    U='S';
0831                    I=I-1;
0832                    GO TO NEXT;
0833                   END;
0834    IF GOOUT=29 THEN DO;
0835                    IF U^=' ' THEN GO TO ERRRR;
0836                    I=I-1;
0837                    U='S';
0838                    PUT STRING(IDNT)EDIT(IDNTS)(A(9));
0839                    CODES=CODE;
0840                    CALL SEARCHB;
0841                    IF CODE=0 THEN CALL INSERT;
0842                    IF TYPE(DCODE)=0 THEN TYPE(DCODE)=4;
0843                    IF TYPE(DCODE)^=4 THEN DO;
0844                          CODE=CODES;
0845                          PUT FILE(EFILF)EDIT(CA,ER29)
0846                          (SKIP,X(1),F(6),X(1),A(50));
0847                          GO TO NEXT;
0848                          END;
0849                    OP='JMS';
0850                    T1=1;
0851                    C1=CODE;
0852                    B1=DMCRBL;
0853                    T2=0;
0854                    C2=0;
0855                    B2=1;
0856                    CODE=CODES;
```

```
0857                    CALL GENCOD;
0858                   GO TO NEXT;
0859                  END;
0860   IF GOOUT=30 THEN DO;
0861                    I=I-1 ;
0862                    UCODE(UTAB(KUTAB))=QTURPLE;
0863                      KUTAB=KUTAB-1;
0864                    GO TO NEXT;
0865                   END;
0866   IF GOOUT=31 THEN DO;
0867                    S(I)=4;
0868                    U=' ';
0869                    OP='BRN';
0870                    T1=0;
0871                    C1=15;
0872                    T2=5;
0873                    UCODE(UTAB(KUTAB))=QTURPLE+1;
0874                    KUTEMPL=KUTEMPL+1;
0875                    C2=KUTEMPL;
0876                    UTAB(KUTAB)=KUTEMPL;
0877                    CALL GENCOD;
0878                    GO TO NEXTS;
0879                   END;
0880   IF GOOUT=32 THEN DO;
0881                    IF U^='S'THEN
0882                         PUT FILE(EFILE)EDIT(CA,ER32)
0883                         (SKIP,X(1),F(6),X(1),A(50));
0884                    I=I-1;
0885                    U='S';
0886                    UCODE(UTAB(KUTAB))=QTURPLE;
0887                    KUTAB=KUTAB-1;
0888                    GO TO NEXT;
0889                   END;
0890   IF GOOUT=33 THEN DO;
0891                    PUT FILE(EFILE)EDIT(CA,ER33)
0892                     (SKIP,X(1),F(6),X(1),A(50));
0893                    U=' ';
0894                    GO TO NEXTS;
0895                   END;
0896  GOOUT34:
0897   IF GOOUT=34 THEN DO;
0898                    IF U^=' 'THEN
0899                        PUT FILE(EFILE)EDIT(CA,ER34)
0900                        (SKIP,X(1),F(6),X(1),A(50));
0901                    CALL OPPROC;
0902                    IF IFLAG=1 THEN DO;
0903                       IFLAG=0;
0904                       GO TO NEXT;
0905                      END;
0906                    I=I+1;
0907                    S(I)=5;
0908                    GO TO NEXTS;
0909                   END;
0910   IF GOOUT=35 THEN DO;
0911                    PUT FILE(EFILE)EDIT(CA,ER35)
0912                    (SKIP,X(1),F(6),X(1),A(50));
0913                    GOOUT=34;
0914                    GO TO GOOUT34;
0915                   END;
0916   IF GOOUT=36 THEN DO;
0917                    IF U^='S' THEN DO;
0918                       PUT FILE(EFILE)EDIT(CA,ER36)
0919                       (SKIP,X(1),F(6),X(1),A(50));
0920                       GO TO NEXTS;
0921                      END;
0922                     OP='ENP';
```

```
0923                    PUT FILE(CODEF)EDIT(QTURPLE,OP)
0924                    (SIP,X(1),F(6),X(1),A(3));
0925                    QTURPLE=QTURPLE+1;
0926                    CALL CLBLCK;
0927                    GO TO NEXTS;
0928                   END;
0929   ERRRR:
0930        PUT FILE(EFILE)EDIT(CA,ER0)(SKIP,
0931            X(1),F(6),X(1),A(50));
0932   NXTSSS:
0933        IF S(I)=1 THEN DO;
0934                        CALL NXTEND;
0935                        GO TO NEXTS;
0936                       END;
0937        IF S(I)=5 THEN DO;
0938                        CALL NEXTEN;
0939                        GO TO NEXT;
0940                       END;
0941        IF S(I)=6 THEN DO;
0942                        CALL NEXTEN;
0943                        GO TO NEXT;
0944                       END;
0945        CALL NXTEND;
0946        U=' ';
0947        I=I-1;
0948   NEXTS:
0949       CALL GETSYM;
0950       IF IFLAG=1 THEN GO TO NEXTS;
0951       GO TO NEXT;
0952      END STATE;
0953   NXTEND:PROCEDURE;
0954   NXTSS:
0955         CALL NEXTEN;
0956         IF TAB=4 THEN RETURN;
0957         GO TO NXTSS;
0958        END NXTEND;
0959   IO:PROCEDURE;
0960   NEXTR:
0961        CALL SEARCHB;
0962        IF CODE=0 THEN CALL INSERT;
0963        B1=DBCRBL;
0964        T1=1;
0965        T2=0;
0966        C1=CODE;
0967        C2=0;
0968        B2=1;
0969        CALL GENCOD;
0970        CALL GETSYM;
0971        IF IFLAG=1 THEN RETURN;
0972        IF TAB=4 THEN
0973           IF CODE=7 THEN DO;
0974                          CALL GETSYM;
0975                          IF IFLAG=1 THEN RETURN;
0976                          IF TAB^=1THEN DO;
0977                                         IFLAG=1;
0978                                         RETURN;
0979                                        END;
0980                          OP='ARG';
0981                          GO TO NEXTR;
0982                         END;
0983        RETURN;
0984       END IO;
0985   GENCOD:PROCEDURE;
0986        PUT FILE(CODEF)EDIT(QTURPLE,OP,B1,T1,C1,B2,T2,C2)
0987        (SKIP,X(1),F(6),X(1),A(3),X(1),F(3),X(1),F(1),X(1),F(6),
0988        X(1),F(3),X(1),F(1),X(1),F(6));
```

```
0989          QTURPLE=QTURPLE+1;
0990          RETURN;
0991        END GENCOD;
0992    CLBLCK:PROCEDURE;
0993    CLOSB:
0994          IF BLCKEN(CURRBL)=0 THEN GO TO RTN;
0995          DO II=1 TO BLCKEN(CURRBL);
0996             LASTEL=LASTEL-1;
0997             ADDSS(LASTEL)=ADDSS(TOPEL);
0998             TYPE(LASTEL)=TYPE(TOPEL);
0999             SYMB(LASTEL)=SYMB(TOPEL);
1000             DCLR(LASTEL)=DCLR(TOPEL);
1001             TOPEL=TOPEL-1;
1002          END;
1003          BLCKPT(CURRBL)=LASTEL;
1004     RTN:
1005          IF BLCKID(CURRBL)=1 THEN DO;
1006                                          OP='END';
1007                                          PUT FILE(CODEF)EDIT(QTURPLE,OP)
1008                                          (SKIP,X(1),F(6),X(1),A(3));
1009                                          QTURPLE=QTURPLE+1;
1010                                          NM=CURRBL;
1011                                          CURRBL=BLCKLV(CURRBL);
1012                                          IF BLCKEN(NM)=0 THEN GOTO NXT;
1013                                          DO II=1 TO BLCKEN(NM);
1014                                           III=II+LASTEL-1;
1015                                            IF DCLR(III)=0 THEN DO;
1016                                            PUT STRING(IDNT)EDIT
1017                                               (SYMP(III))(A(9));
1018                                            CALL SEARCHB;
1019                                            IF CODE=0THEN DO;
1020                                             BLCKEN(CURRBL)=BLCKEN(CURRBL)+1;
1021                                             TOPEL=TOPEL+1;
1022                                             ADDSS(TOPEL)=ADDSS(III);
1023                                             TYPE(TOPEL)=TYPE(III);
1024                                             SYMB(TOPEL)=SYMB(III);
1025                                             DCLR(TOPEL)=0;
1026                                            END;
1027                                           END;
1028                                          END;
1029             NXT:
1030                   IF BLCKID(CURRBL)=1 THEN RETURN;
1031                   IF BLCKLV(CURRBL)=0 THEN RETURN;
1032                   GO TO CLOSB;
1033                  END;
1034               IF BLCKEN(CURRBL)=0 THEN RETURN;
1035               DO II=1 TO BLCKEN(CURRBL);
1036                  III=II+TOPEL=1;
1037                   IF DCLR(III)=0 THEN
1038                   PUT FILE(EFILE)EDIT(SYMB(III),
1039                       'LABEL NOT DECLARED')
1040                    (SKIP,X(19,A(9),X(1),A(30));
1041               END;
1042             RETURN;
1043           END CLBLCK;
1044    BLCKST:PROCEDURE;
1045          LASTBL=LASTBL+1;
1046          BLCKLV(LASTBL)=CURRBL;
1047          BLCKEN(LASTBL)=0;
1048          BLCKPT(LASTBL)=TOPEL+1;
1049          BLCKID(LASTBL)=CODE-4;
1050          CURRBL=LASTBL;
1051          RETURN;
1052        END BLCKST;
1053    SEARCHB:PROCEDURE;
1054          DMCRBL=CURRBL;
```

```
1055   NXTSRH:
1056        CALL SRDMBL;
1057        IF CODE^=0 THEN RETURN;
1058        DMCRBL=BLCKLV(DMCRBL);
1059        IF DMCRBL^=0 THEN GO TO NXTSRH;
1060        RETURN;
1061        END SEARCHB;
1062   SEARCH:PROCEDURE;
1063        DMCRBL=CURRBL;
1064        CALL SRDMBL;
1065        RETURN;
1066        END SEARCH;
1067   INSERT:PROCEDURE;
1068         BLCKEN(CURRBL)=BLCKEN(CURRBL)+1;
1069         TOPEL=TOPEL+1;
1070         DCODE=TOPEL;
1071         PUT STRING(SYMB(TOPEL))EDIT(IDNT)(A(9));
1072         TYPE(DCODE)=0;
1073         CODE=BLCKEN(CURRBL);
1074         DCLR(DCODE)=0;
1075         RETURN;
1076         END INSERT;
1077   OPPROC:PROCEDURE;
1078         CALL PROCID;
1079         IF IFLAG=1 THEN RETURN;
1080         CALL GETSYB;
1081         IF TAB^=4!CODE^=8 THEN DO;
1082                          PUT FILE(EFILE)EDIT(CA,EROPPR)
1083                          (SKIP,X(1),F(6),X(1),A(50));
1084                          GO TO NXT;
1085                         END;
1086         CALL BLCKST;
1087   NXTDLR:
1088         CALL GETSYB;
1089   NXT:
1090         IF TAB^=3 THEN RETURN;
1091         REALS=0;
1092         CALL REALSC;
1093         IF REALS=0 THEN RETURN;
1094   OUTT:
1095         CALL GETSYM;
1096         IF TAB^=1 THEN DO;
1097                    PUT FILE(EFILE)EDIT(CA,EROPPR)
1098                    (SKIP,X(1),F(6),X(1),A(50));
1099                    RETURN;
1100                   END;
1101         CALL DCLREN;
1102         GO TO NXTDLR;
1103        END OPPROC;
1104   OPBLCK:PROCEDURE;
1105         CALL BLCKST;
1106         OP='BGN';
1107         PUT FILE(CODEF)EDIT(QTURPLE,OP)
1108         (SKIP,X(1),F(6),X(1),A(3));
1109         QTURPLE=QTURPLE+1;
1110   NXTDCR:
1111         CALL GETSYM;
1112         IF TAB^=3 THEN RETURN;
1113         REALS=0;
1114         CALL REALSC;
1115         IF REALS=0!REALS=4 THEN RETURN;
1116         CALL GETSYM;
1117         IF TAB^=1 THEN DO;
1118                    PUT FILE(EFILE)EDIT(CA,EROPPR)
1119                    (SIKP,X(1),F(6),X(1),A(50));
1120                    RETURN;
```

```
1121                    END;
1122          CALL DCLREN;
1123          GO TO NXTDCR;
1124        END OPBLCK;
1125    DCLREN:PROCEDURE;
1126    NXTDL:
1127          IF TAB^=1 THEN RETURN;
1128          CALL SEARCH;
1129    NXT:
1130          IF CODE=0 THEN DO;
1131             CALL INSERT;
1132             DCLR(DCODE)=1;
1133             TYPE(DCODE)=REALS;
1134             GO TO NXTS;
1135           END;
1136          PUT FILE(EFILE)EDIT(CA,DCLR0)
1137          (SKIP,X(1),F(6),X(1),A(50));
1138    NXTS:
1139          CALL GETSYM;
1140          IF TAB^=4 THEN DO;
1141                     PUT FILE(EFILE)EDIT(CA,DCLR1)
1142                     (SKIP,X(1),F(6),X(1),A(50));
1143                     RETURN;
1144                    END;
1145          IF CODE=8 THEN RETURN;
1146          IF CODE^=7 THEN DO;
1147                  IFLAG=1;
1148                  RETURN;
1149                END;
1150          CALL GETSYM;
1151          GO TO NXTDL;
1152        END DCLREN;
1153    REALSC:PROCEDURE;
1154          IF CODE= 4THEN REALS= 4;
1155          IF CODE=11THEN REALS= 3;
1156          IF CODE=12THEN REALS= 2;
1157          RETURN;
1158        END REALSC;
1159    PROCID:PROCEDURE;
1160        CALL GETSYM;
1161        IF TAB^=1 THEN DO;
1162                   PUT FILE(EFILE)EDIT(CA,EROPPR0)
1163                   (SKIP,X(1),F(6),X(1),A(50));
1164                   IFLAG=1;
1165                   RETURN;
1166                  END;
1167        CALL SEARCH;
1168        IF CODE=0 THEN DO;
1169                   CALL INSERT;
1170                   DCLR(DCODE)=1;
1171                   TYPE(DCODE)=4;
1172                   OP='PRC';
1173                   PUT FILE(CODEF)EDIT(QTURPLE,OP,DMCRBL,TAB,CODE)
1174                  (SKIP,X(1),F(6),X(1),A(3),(X(1),F(3),X(1),F(1),X(1),F(6));
1175                  RETURN;
1176                  END;
1177                  PUT FILE(EFILE)EDIT(CA,EROPPR3)
1178                  (SKIP,X(1),F(6),X(1),A(50));
1179                  IFLAG=1;
1180                  RETURN;
1181                END PROCID;
1182    NEXTEN:PROCEDURE;
1183     NXT:
1184        CALL GETSYM;
1185        IF TAB<3 THEN GOTO NXT ;
1186           IF TAB=3 THEN DO;
```

```
1187              IF CODE=9 THEN RETURN;
1188              GO TO NXT;
1189             END;
1190           IF CODE=8 THEN RETURN;
1191           GO TO NXT;
1192          END NEXTEN;
1193   SRDMBL:PROCEDURE;
1194         DCL
1195           I      FIXED BINARY(15),
1196           J      FIXED BINARY(15),
1197           N      FIXED BINARY(15);
1198         DCODE=BLCKPT(DMCRBL);
1199         CODE=0;
1200         N=BLCKEN(DMCRBL);
1201         IF N=0 THEN RETURN;
1202         J=BLCKPT(DMCRBL);
1203         N=N+J-1;
1204         DO I=J TO N;
1205           IF SYMB(I)=IDNT THEN DO;
1206                               CODE=I-J+1;
1207                               DCODE=I;
1208                               RETURN;
1209                              END;
1210         END;
1211         RETURN;
1212        END SRDMBL;
1213    EXP:PROCEDURE;
1214             DCL US        FIXED BINARY(15),
1215                 UST       FIXED BINARY(15),
1216                 USB       FIXED BINARY(15),
1217                 C1        FIXED BINARY(15),
1218                 B1        FIXED BINARY(15),
1219                 B2        FIXED BINARY(15),
1220                 C2        FIXED BINARY(15),
1221                 T1        FIXED BINARY(15),
1222                 T2        FIXED BINARY(15),
1223                 J         FIXED BINARY(15),
1224                 I         FIXED BINARY(15),
1225                 UFLAG     FIXED BINARY(15),
1226                GOOUT      FIXED BINARY(15),
1227                S(32)      FIXED BINARY(15),
1228                TJ         FIXED BINARY(15),
1229                OP         CHAR(3),
1230                U          CHAR(1);
1231             TJ=0;
1232             S(1)=1;
1233             U=' ';
1234             I=1;
1235             IF TAB=1 THEN GO TO CHECKV;
1236        NEXT:
1237           IF TAB=0 THEN DO;
1238                         GOOUT=A(1,S(I));
1239                         GO TO OUT;
1240                         END;
1241           IF TAB=1 THEN DO;
1242                         GOOUT=A(8,S(I));
1243                         GO TO OUT;
1244                         END;
1245           IF TAB=2 THEN DO;
1246                         GOOUT=A(8,S(I));
1247                         GO TO OUT;
1248                         END;
1249           IF TAB=4 THEN DO;
1250                         IF CODE^<9 THEN DO;
1251                                          GO TO ERRRR;
1252                                        END;
```

```
1253                                IF CODE = 7 THEN DO;
1254                                                 GO TO ERRRR;
1255                                                 END;
1256                                IF CODE=8 THEN DO;
1257                                                 GOOUT=A(1,S(I));
1258                                                 GOTO OUT;
1259                                                END;
1260                           GOOUT=A(CODE+1,S(I));
1261                           GO TO OUT;
1262                           END;
1263          IF TAB=3 THEN DO;
1264                           GOOUT=A(1,S(I));
1265                           GO TO OUT;
1266                           END;
1267        OUT:
1268          IF GOOUT= 0 THEN   GO TO ERRRR;
1269          IF GOOUT= 1 THEN   DO;
1270                              IF U^=' ' THEN DO;
1271                                             GO TO ERRRR;
1272                                             END;
1273                              I=I+1;
1274                              S(I)=8;
1275                              GO TO NEXTS;
1276                           END;
1277          IF GOOUT= 2 THEN   DO;
1278                              IF U^='T'|U^='E' THEN DO;
1279                                             GO TO ERRRR;
1280                                             END;
1281                              U='E';
1282                              TT2=6;
1283                              BB2=1;
1284                              CC2=1;
1285                              RETURN;
1286                              END;
1287          IF GOOUT= 3 THEN   DO;
1288                              IF U^='T' THEN DO;
1289                                             GO TO ERRRR;
1290                                             END;
1291                              U='E';
1292                              I=I-1;
1293                              GO TO GENPOMN;
1294                              END;
1295          IF GOOUT= 4 THEN   DO;
1296                              IF U^='T' THEN DO;
1297                                             GO TO ERRRR;
1298                                             END;
1299                              I=I-1;
1300                              GO TO GENPOMN;
1301                              END;
1302          IF GOOUT= 5 THEN   DO;
1303                              IF U^='T' THEN DO;
1304                                             GO TO ERRRR;
1305                                             END;
1306                              I=I+1;
1307                              S(I)=2;
1308                              U=' ';
1309                              GO TO NEXTS;
1310                              END;
1311          IF GOOUT= 6 THEN   DO;
1312                              IF U^='F' THEN DO;
1313                                             GO TO ERRRR;
1314                                             END;
1315                              I=I+1;
1316                              S(I)=5;
1317                              U=' ';
1318                              END;
```

```
1319          IF GOOUT= 7 THEN    DO;
1320                               IF U^='F' THEN DO;
1321                                                GO TO ERRRR;
1322                                                END;
1323                               I=I+1;
1324                               S(I)=4;
1325                               U=' ';
1326                               GO TO NEXTS;
1327                               END;
1328          IF GOOUT= 8 THEN    DO;
1329                               IF U^='T' THEN DO;
1330                                                GO TO ERRRR;
1331                                                END;
1332                               I=I-1;
1333                               GO TO GENMPDV;
1334                               END;
1335          IF GOOUT= 9 THEN    DO;
1336                               IF U^='T' THEN DO;
1337                                                GO TO ERRRR;
1338                                                END;
1339                               I=I-1;
1340                               U='F';
1341                               GO TO GENMPDV;
1342                               END;
1343          IF GOOUT=10 THEN    DO;
1344                               IF U^=' ' THEN DO;
1345                                                GO TO ERRRR;
1346                                                END;
1347                               U='T';
1348                               GO TO STORE;
1349                               END;
1350          IF GOOUT=11 THEN    DO;
1351                               IF U^=' ' THEN DO ;
1352                                                GO TO ERRRR;
1353                                                END;
1354                               I=I+1;
1355                               S(I)=7;
1356                               GO TO STORE;
1357                          END;
1358     GOOUT12:
1359          IF GOOUT=12 THEN    DO;
1360                               IF U^=' ' THEN DO ;
1361                                                GO TO ERRRR;
1362                                                END;
1363                               I=I-1;
1364                               U='T';
1365                               IF S(I)=1 THEN GO TO EX1213;
1366                               IF S(I)=6 THEN GO TO EX1213;
1367                               IF S(I)=8 THEN GO TO EX1213;
1368                                GO TO NEXT;
1369                               END;
1370          IF GOOUT=13 THEN    DO;
1371                               IF U^=' ' THEN DO ;
1372                                                GO TO ERRRR;
1373                                                END;
1374                               U='F';
1375                               I=I-1;
1376                               GO TO EX1213;
1377                              END;
1378          IF GOOUT=14 THEN    DO;
1379                               IF U^='T' THEN DO;
1380                                                  IF U^='E' THEN
1381                                                GO TO ERRRR;
1382                                                END;
1383                               S(I)=9;
1384                               U=' ';
```

```
1385                                GO TO NEXTS;
1386                                END;
1387          IF GOOUT=15 THEN    DO;
1388                                IF U^=' ' THEN DO ;
1389                                               GO TO ERRRR;
1390                                               END;
1391                                I=I-1;
1392                                U='T';
1393                                GO TO NEXT;
1394                                END;
1395          IF GOOUT=16 THEN    DO;
1396                                IF U^=' ' THEN DO;
1397                                               GO TO ERRRR;
1398                                               END;
1399                                I=I-1;
1400                                U='F';
1401                                IF S(I)=4! S(I)=5 THEN U='T';
1402                                GO TO NEXT;
1403                                END;
1404          IF GOOUT=17 THEN    DO;
1405                                IF U^=' 'THEN DO;
1406                                              GO TO ERRRR;
1407                                              END;
1408                                IF S(I-1)=1 THEN DO;
1409                                                 TT2=US;
1410                                                 BB2=USB;
1411                                                 CC2=UST;
1412                                                 RETURN;
1413                                                 END;
1414                               GOOUT=12;
1415                               GO TO GOOUT12;
1416                               END;
1417          IF GOOUT=18 THEN    DO;
1418                                IF U^=' 'THEN GO TO ERRRR;
1419                                S(I)=3;
1420                                GO TO NEXTS;
1421                               END;
1422          IF GOOUT=19 THEN    DO;
1423                                IF U^='T' THEN DO;
1424                                               GO TO ERRRR;
1425                                               END;
1426                                I=I-1;
1427                                GO TO NEG;
1428                                END;
1429          IF GOOUT=20 THEN    DO;
1430                                IF U^='T' THEN DO;
1431                                               IF U^=' ' THEN DO;
1432                                                              GO TO ERRRR;
1433                                                              END;
1434                                               I=I+1;
1435                                               S(I)=6;
1436                                               GO TO NEXTS;
1437                                               END;
1438                                I=I+1;
1439                                S(I)=3;
1440                                U=' ';
1441                                GO TO NEXTS;
1442                                END;
1443          IF GOOUT=21 THEN    DO;
1444                                IF U^='F' THEN DO;
1445                                               GO TO ERRRR;
1446                                               END;
1447                                I=I-1;
1448                                GO TO NEG;
1449                                END;
1450        ERRRR:
```

```
1451              IFLAG=1;
1452              RETURN;
1453         NEXTS:
1454              CALL GETSYM;
1455              IF IFLAG=1 THEN RETURN;
1456    CHECKV:
1457              IF TAB=1 THEN DO;
1458                                CALL SEARCHB;
1459                                IF CODE=0 THEN CALL INSERT;
1460                                IF TYPE(DCODE)=0 THEN TYPE(DCODE)=2;
1461                                IF TYPE(DCODE)<2 THEN
1462                                   PUT FILE(EFILE)EDIT(CA,ER4)
1463                                   (SKIP,X(1),F(6),X(1),A(50));
1464                                END;
1465              GO TO NEXT;
1466         EX1213:
1467              UFLAG=0;
1468              OP='MOV';
1469              TJ=TJ+1;
1470              T1=6;
1471              T2=US;
1472              C2=UST;
1473              B2=USB;
1474              B1=1;
1475              C1=TJ;
1476              GO TO GENCOD;
1477         STORE:
1478              US=TAB;
1479              UST=CODE;
1480              USB=DMCRBL;
1481              UFLAG=1;
1482              GO TO NEXTS;
1483         NEG:
1484              OP='NEG';
1485              B1=1;
1486              B2=1;
1487              T1=6;
1488              C1=TJ;
1489              T2=0;
1490              C2=0;
1491              GO TO GENCOD;
1492         GENPOMN:
1493              IF S(I+1)=2 THEN OP='ADD';
1494                          ELSE OP='SUB' ;
1495              T1=6;
1496              IF UFLAG=1 THEN DO;
1497                                UFLAG=0;
1498                                C1=TJ;
1499                                T2=US;
1500                                C2=UST;
1501                                B2=USB;
1502                                GO TO GENCOD;
1503                               END;
1504              C2=TJ;
1505              TJ=TJ-1;
1506              C1=TJ;
1507              T2=6;
1508              B1=1;
1509              B2=1;
1510         GENCOD:
1511            PUT FILE(CODEF)EDIT(QTURPLE,OP,B1,T1,C1,B2,T2,C2)
1512           (SKIP,X(1),F(6),X(1),A(3),X(1),F(3),X(1),F(1),X(1),F(6),
1513           X(1),F(3),X(1),F(1),X(1),F(6));
1514           QTURPLE=QTURPLE+1;
1515              GO TO NEXT;
1516         GENMPDV:
```

```
1517            IF S(I+1) =4 THEN OP='MPY';
1518                         ELSE OP='DIV';
1519            T1=6;
1520            IF UFLAG=1 THEN DO;
1521                                UFLAG=0;
1522                                C1=TJ;
1523                                T2=US;
1524                                C2=UST;
1525                                B2=USB;
1526                                GO TO GENCOD;
1527                                END;
1528            B1=1;
1529            B2=1;
1530            C2=TJ;
1531            TJ=TJ-1;
1532            C1=TJ;
1533            T2=6;
1534            GO TO GENCOD;
1535            END EXP;
1536        EOJ:
1537         DO K=1 TO KUTEMPL;
1538          PUT FILE(TEMPLF)EDIT(K,UCODE(K))
1539          (SKIP,X(1),F(4),X(1),F(6));
1540         END;
1541         DO K=1 TO LASTBL;
1542          PUT FILE(BLCKF)EDIT(K,BLCKLV(K),BLCKEN(K),
1543          BLCKPT(K),BLCKID(K))
1544          (SKIP,X(1),F(4),X(1),F(6)),X(1),F(4),
1545          X(1),F(6),X(1),F(1));
1546          N=BLCKEN(K);
1547          IF N=0 MEN GO TO EOF;
1548          I=BLCKPT(K);
1549          N=N+I-1;
1550          DO J=I TO N;
1551           PUT FILE(SYMBLF)EDIT(J,SYMB(J),TYPE(J),DCLR(J),
1552           ADDSS(J))
1553          (SKIP,X(1),F(4),X(1),A(9),X(1),F(1),X(1),F(1),X(1),F(6));
1554          END;
1555    EOF:
1556          M=0;
1557         END;
1558        CLOSE FILE(PFILE),FILE(CFILE),FILE(SYMBLF),FILE(CODEF),
1559              FILE(EFILE),FILE(LFILE),FILE(TEMPLF),FILE(BLCKF);
1560       END ALGOLS;
```