Introduction to Compiler Writing

R. C. T. Lee

Institute Of Computer and Decision Sciences

National Tsing Hua University

Hsinchu, Taiwan

Republic Of China

C. W. Shen

First International Computer Corporation

Taipei, Taiwan

Republic Of China

S. C. Chang

Department Of Electrical Engineering and Computer Sciences

Northwestern University

Evanston, Illinois

U. S. A.

ABSTRACT

This chapter introduces compiler writing techniques. It was written with the following observations made by the authors:

(1) While compiler writing may be done without too much knowledge of formal language or automata theory, most books on compiler writing devote most of their chapters on these topics. This inevitably gives most readers a false impression that compiler writing is a difficult task. In fact, it is our observation that may people many easily develop some inferior complex out or reading these difficult books.

We have deliberately avoided this formal approach. Each aspect of compiler writing is presented in such an informal way that any naive reader can finish reading this chapter within, say five or six, casual hours. As a matter of fact, a lot of readers finished reading this chapter before they went to sleep. They almost treated this chapter as a bed-time novel.

(2) There are several different aspects of compiler writing. Most books would give detailed presentations on every aspect and still fail to give the reader some feeling on how to write a compiler. We decided to use a different approach; we shall give an anatomy of a simple compiler. Most important of all, the internal data structure of this simple compiler will be clearly presented. Examples will be given and a reader will be able to understand how a compiler works by studying these examples.

In other words, we make sure that a reader will not lose his global view of compiler writing; he sees not only trees, but also forests.

Section 1. Introduction.

A compiler is a software which translates a high-level language program into a low-level language program which can be executed by a computer. In many compilers, the low-level language is the machine language. For our purpose, we assume that our compiler translates a high-level language into an assembly language simply because it is easier for us to comprehend assembly languages.

It is our experience that a student, while learning compiler writing technique, has to spend a major part of his time studying an assembly language. This is rather unfortunate because the code generation part of a compiler is quite straightforward. One just has to remember many details. By spending too much time in learning assembly languages, a student loses his precious time to study the other important techniques in compiler writing.

To make sure that a student would not have to spend too much time in studying assembly languages, we decided to use a very simple assembly language. The assembler language is described in Section 5. As the reader can see, it is very easy to learn .

Yet, it contains enough instructions that it will not be difficult for a student to translate a high-level language into this assembly language.

If a student is asked to write a compiler with our assembly language as its target language, he will not be bogged down by the details of the assembly language. Instead, he can concentrate all his mind to study the other important compiler writing skills.

So far as the high-level language is concerned, it is the same story; we do not want to use a very complicated high-level language, such as FORTRAN, or PASCAL. We do not want to use a subset of FORTRAN, either for various reasons which will become clear later. We therefore designed a high-level language, called FRANCIS, for our compiler writing purpose. FRANCIS is sufficiently complicated that writing a compiler for it is a non-trivial job. Still, FRANCIS is ideal for compiler writing because we deliberately designed it for this purpose. The reader will gradually appreciate this point.

Essentially, FRANICS is similar to FORTRAN, except that it contains some special features from PASCAL. It contains most of the well-known FORTRAN instructions, such as DIMENSION, SUBROUTINE, GO TO and so on. So far as the IF statement is concerned, we have an IF……. THEN……. ELSE type of statement, a typical feature borrowed from PASCAL. However, to simplify the syntax analysis discussion, we do not allow IF inside IF. All of the variables have

to be declared. Again, a feature borrowed from PASCAL. We shall first assume that FRANCIS is not a recursive language in the sense that subroutines in FRANCIS can not call themselves. After making sure that students understand the basic concept of compiler writing, we then add the recursive feature and also allow the IF statement to contain IF statements. By that time, it should not be difficult at all to understand how these can be done.

The reader may ask: Why is the language called FRANCIS. Well, we name it FRANCIS in memory of the great saint, St. Francis of Assisi.

This chapter was written when the world was facing an energy crisis. A lot of people were quite worried because they might face a dry summer (not enough gasoline for their cars) and a cold winter (not enough heating oil for their homes). The fact that millions of homeless people having not enough food to eat and not enough water to drink did not seem to bother them.

We believe that the world does not lack energy. What we need is more warmth and more love. As the saying goes, "When the world grows colder, God sent the little man from Assisi to warm it up." We certainly need him again.

The appendix contains all of the syntactical rules of FRANCIS.

In the appendix, we have used the Backus Normal Form (BNF for short) to describe the syntax rules of our language. The following

meta-symbols belong to the BNF formalism and do not appear in the FRANCIS language:

: = | { } < >

English words enclosed by < and > are elementary constructs in FRANCIS, The symbol ::= means "is equivalent to". The symbol | means "or" and the brackets { and } denote possible repetition of the symbols zero or more times.

Let us consider an example. In FRANCIS, an identifier always starts with a letter and this letter is followed by letters, numbers or nothing. This definition can be expressed as follows:

<identifier> ::= <letter>{<letter>|<digit>}.

In FRANCIS reserved works are used. These reserved words can not be used as ordinary identifiers. They are all underlined in the definition. For instance, the following expression describes the syntax rules of array declaration instructions:

<array declaration part ::= DIMENSION array declaration ; For the definition of array definition, consult the appendix.

Section 2. A Glimpse of Compilers.

Before getting into the details of a compiler, let us consider a very simple program and see what the compiler should do. The program is as follows:

```
VARIABLE INTEGER: X, Y, Z;
X = 7;
Y = 9;
Z =X - Y;
END;
```

For the first instruction declaring X, Y and Z to be integers, the compiler will generate the following assembly language instructions:

```
X DS 1
Y DS 1
Z DS 1
```

For the instruction

```
X = 7;
```

the compiler will create a constant, say called Il, and assembly language codes as follows:

```
I1 DC 7
   LD I1
   ST X
```

Similarly, for instruction

```
Y = 9;
```

the compiler will generate the following codes:

```
I2 DC 9
   LD I2
   ST Y
```

Finally, for the instruction

```
Z = X + Y;
```

the compiler will generate

```
LD X
AD Y
ST T1
LD T1
ST Z
```

The reader may wonder why the above sequence of codes are so inefficient. They should simply be

```
LD X
AD Y
ST Z
```

In fact, we deliberately showed the codes which are not

efficient for reasons that will become clear later.

Ultimately, the high-level language program will be translated into the following assembly language codes:

```
X DS 1
Y DS 1
Z DS 1
I1 DC 7
I2 DC 9
   LD I1
   ST X
   LD I2
   ST Y
   LD X
   AD Y
   ST T1
   LD T1
   ST Z
```

This is what a compiler would do. But, how does it do it?

Basically, a compiler is divided into three parts: the lexical analyzer, the syntax analyzer and the code generator, as described below:

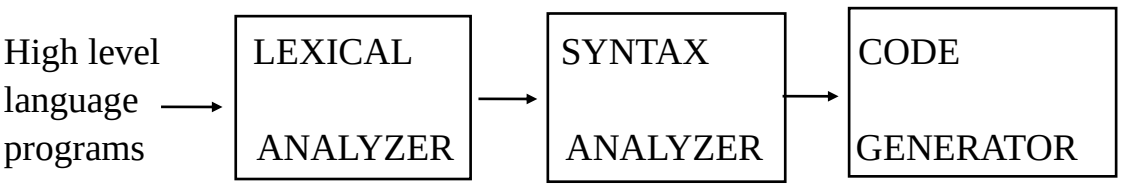


fig. 2.1

(1) The input of the lexical analyzer is a sequence of characters representing a high-level language program. The major function of the lexical analyzer is to divide this sequence of characters into a sequence of tokens. For instance, consider the instruction.

VARIABLE INTEGER: X, Y, Z;

The output of the lexical analyzer is

VARIABLE

INTEGER

:

X

,

Y

,

Z

;

(2) The input of the syntax analyzer is a sequence of tokens which is the output of the lexical analyzer. The syntax analyzer divides the tokens into instructions. For each instruction, using the (grammatical rules) already stored, the syntax analyzer determines whether this instruction is grammatically correct. If not, error messages will be printed out. Otherwise, this instruction is decomposed into a sequence of basic instructions and these instructions are fed into the code generator to

produce assembly language codes.

For instance, consider the instruction

X=Y+U*V;

The syntax analyzer would determine that this instruction is indeed correct and later produce three basic instructions as follows;

T1=U*V

T2=Y+T1

X=T2

(3) The code generator accepts the output from the syntax analyzer. Every basic instruction is now translated into a sequence of assembly language instructions. This can be easily done because the code generator can examine the pattern of the basic instruction and determine which sequence of codes it must generate.

To make sure that the code generator can recognize the pattern easily, we may assume that each basic instruction produced by the syntax analyzer is of a standard from. In our case, we may assume that the standard form is a quadruple as follows:

(operator, $Operand_1$, $Operand_2$, Result).

For instance, for basic instruction

X=Y+Z,

its standard form will be

(+, Y, Z, X)

For the instruction

X=Y-Z,

The standard form will be

(- , Y, Z, X)

The code generator generates assembly language codes by examining the operator, operands and result.

Section 3. The Lexical Analyzer

As we indicated before, the function of the lexical analyzer is to group the input sequence of the characters into tokens. This is the major function. Actually, as the reader will see, the lexical analyzer does more than identifying tokens.

Section 3.1 The Identification of Tokens.

So far as the language FRANCIS is concerned, the identification of tokens is trivial. Note that tokens in FRANCIS are separated by delimiters, such as "+", "-", "=" and so on, or blanks. Moreover, each delimiter itself is a token and no token consists of more than one delimiter. We therefore scan the input sequence sequentially and examine each character to see whether it is a delimiter or a blank. If it is a blank, the previous string already formed is a token unless the previous string consists of blanks. If it is a delimiter, we have identified two tokens; the previous string formed is a token (assuming that it is not empty) and the present delimiter is also a token.

An algorithm to identify tokens for the language FRANCIS is shown in Fig. 3.1.

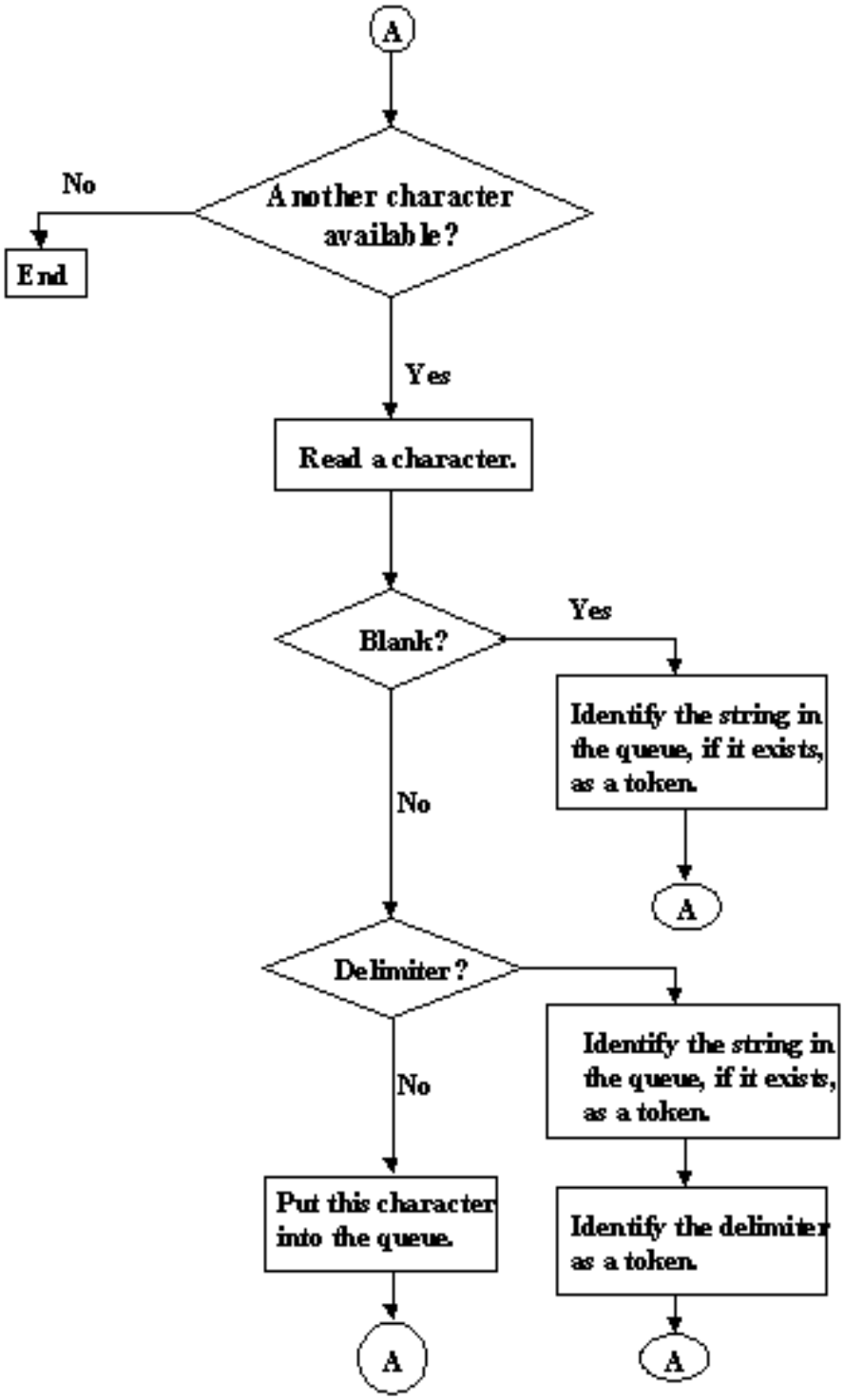

A
Another character available?
No
End
Yes
Read a character.
Blank?
Yes
Identify the string in the queue, if it exists, as a token.
A
No
Delimiter?
Identify the string in the queue, if it exists, as a token.
No
Put this character into the queue.
Identify the delimiter as a token.
A
A

Fig. 3.1

Note that for some other high-level languages, it is by no means easy to identify tokens. A typical example is FORTRAN. In Fortran, "." is sometimes a delimiter and sometimes not. In logical expressions, one may write

X. EO. Y.

Here, "." is a delimiter. However, for a real number

35.76,

the "." inside it is not a delimiter. Thus, in FORTRAN, the detection of "." does not automatically mean that a token is found.

Besides, in FORTRAN, when we encounter the sign "*", we can not say that this is a token because it may be a part of "**" That is , we have to conduct some kind of looking ahead in this case.

We have deliberately defined our high-level language to make sure that the identification of tokens is an easy job. In order to check whether a character is a delimiter or not, we set up a table, called the delimiter table and designated it as Table 1. Since every input symbol will be checked against Table 1, Table 1 is a table which is searched very frequently. It is therefore appropriate to use hashing to store the data into Table 1.

In the following, We show all of the delimiters used in FRANCIS. Note that "." is not contained in Table1.

Delimiters	
1	;
2	(
3	)
4	=
5	+
6	-
7	*
8	/
9	↑
10	‘
11	’
12	:

Table 1

Section 3.2. The Types of Tokens Recognized by the Lexical Analyzer.

We have shown how the lexical analyzer identifies tokens. In this section, We shall illustrate some other functions of the lexical analyzer.

Note that the major job of the lexical analyzer is to prepare everything for the syntax analyzer. Imagine that the syntax analyzer encounters the following instruction:

VARIABLE INTEGER; X, Y, Z;

In this case, the syntax analyzer recognizes that this instruction is a declaration instruction because it detects the token “VARIABLE”. In FRANCIS, the word “VARIABLE” has its particular meaning and can not be used in any other way. For instance, one can not write the following instruction:

VARIABLE = VARIABLE + 1;

The word “VARIABLE” is a reserved word for FRANCIS. Similarly, the words “IF” , “THEN” , “ELSE” and so on are all reserved words in FRANCIS which can not be used as ordinary variables.

It is appropriate to point out here that in FORTRAN, there is no such a reserved word concept. Indeed, one may have

IF = IF + 1

in FORTRAN and this is absolutely not allowed in FRANCIS.

Since the recognition of reserved words is so important for the syntax analyzer, it is appropriate for the lexical analyzer to do the job for it. Table 2 contains all of the reserved words used in FRANCIS. Since this table is also

1. AND
2. BOOLEAN
3. CALL
4. DIMENSION
5. ELSE
6. ENP
7. ENS
8. EQ
9. GE
10. GT
11. GTO
12. IF
13. INPUT
14. INTEGER
15. LABEL
16. LE
17. LT
18. NE
19. OR
20. OUTPUT
21. PROGRAM
22. REAL
23. SUBROUTINE
24. THEN
25. VARIABLE

Table 2 (Reserved Word Table)

searched very frequently, we may use hashing to store and retrieve data for this table.

In addition to recognizing reserved words, we may also recognize numerical constants in the lexical analyzer phase. This can be easily done. If a token starts with a number, it must either be a real number or an integer. It is easy to differentiate these two because a real number contains the ".".

In summary, for any token, the lexical analyzer determines whether it is a delimiter, a real number, an integer, a reserved word or an identifier. We, of course, have never defined identifiers. An identifier is a token which is not a delimiter, a real number, an integer or a reserved word. In general, an identifier usually denotes a variable, an array, a label, the title of a program or the title of a subroutine.

Section 3.3. The Representation of Tokens.

We now ask the question: How does the lexical analyzer let the syntax analyzer know what the type of a token is .

Before answering this question, let us note another problem. Our tokens are of different lengths. For internal representation, this is not an ideal situation.

Note that a token, after the lexical analysis is executed, must belong to one of the following types:

1. Delimiters.

2. Reserved words.

3. Integers.

4. Real numbers.

5. Identifiers.

Each class of tokens has a table associated with it. We have already shown that delimiters are contained in Table 1 and reserved words are contained in Table 2. We now discuss why the other tokens also need tables.

Whenever a token is identified as an integer, we put this integer into Table 3. This is necessary because finally, the code generator is going to use this table to generate constants. For instance, consider the following instructions;

X = 5 ;

Y = 7 ;

After the execution of the lexical analysis phase, Table 3 will contain two integers as below:

5
7
.
.
.

Similarly, we shall designate Table 4 to contain real numbers.

For each identifier, we shall put it into some location in Table 5. the Identifier Table. The Identifier Table is designed to contain three entries: the subroutine to which the identifier belongs, the type of the identifier and a pointer pointing to some other tables. These will be explained in detail later. Meanwhile, we merely have to note the following:

(1) Every identifier is hashed into Table 5.

(2) If an identifier appears more than once in the same subroutine, this identifier is put into Table 5 only once.

(3) If the same identifier appears in different subroutines, it will occupy different locations in Table 5.

We shall use the second column entry of Table 5 to point to the subroutine in which identifier appears.

Let us consider the following example.

```
PROGRAM MAIN;
VARIABLE INTEGER: U, V, W, Z;
U = 56;
V = 79;
W = U + V;
CALL A1(W, 136, Z);
ENP;
SUBROUTINE A1(INTEGER: X, Y, Z);
Z = X +2.7 * Y;
ENS;
```

In the above example, there are two subroutines: MAIN and A1 (The main program is also considered to be a subroutine.). The identifiers which occur in the main program are

MAIN

U

V

W

Z

and the identifiers occurring in the subroutine A1 are

A1

X

Y

Z.

Note that the identifier Z occurs in both MAIN and A1. It will appear twice in Table 5.

After the execution of the lexical analyzer, Table 5 may look like the table shown below:

	Identifier	Subroutine	Type	Pointer
1				
2	W	3		
3	MAIN			
4	Z	3		
5	V	3		
6	X	8		
7				
8	A1	3		
9	U	8		
10	Y	8		
11	Z			

Table 5

Consider the second entry of the above table. This entry contains Identifier W which points to the third location of Table 5. Since the third location of Table 5 contains MAIN, this identifier W appears in Subroutine MAIN. Similarly, in the tenth location, we have Identifier Y pointing to the eighth location of this table. Since the eighth location corresponds to Subroutine A1, Y appears in Subroutine A1.

We have shown that each token, after the lexical analysis phase, is found to be associated with a unique location of one of the tables from Table 1 to Table 5. The meaning of each table is summarized as follows:

1. Delimiter Table
2. Reserved Word Table
3. Integer Table
4. Real Number Table
5. Identifier Table.

We may say that each token is characterized by two numbers i and j if it occupies the jth location of the ith table. Indeed, to solve the problem that tokens are of different lengths, we may simply use this 2-tuple (i , j)to represent this token. Note that this representation is one-to-one in the sense that given (i , j), we can uniquely identify the token. For instance, for the above example, the 2-tuple (5,2) denotes Identifier W and (5,10) denotes Y.

Let us now give a complete example to illustrate our idea by

considering the following programs:

```
PROGRAM MAIN;
VARIABLE INTEGER: U, V, M;
U = 5;
V =7;
CALL S1(U, V, M);
ENP;
SUBROUTINE S1(INTEGER: X, Y, M);
M = X +Y +2.7;
ENS;
```

After the execution of the lexical analyzer, Table 3 will be as follows:

1	5
2	7

Table 3 (Integer Table)

The Real Number Table will contain only one element as below:

1	2.7

Table 4 (Real Number Table)

Let us assume that after identifiers are put into Table 5 Table 5. looks as follows:

	Identifier	Subroutine	Type	Pointer
1	U	3		
2				
3	MAIN			
4	Y	10		
5	V	3		
6	M	3		
7				
8	X	10		
9	M	10		
10	S1			

Table 5 (Identifier Table)

The entire program will now be represented as shown below:

PROGRAM MAIN;
(2,21) (5,3)(1,1)

VARIABLE INTEGER: U , V , M ;
(2,25) (2,14) (1,12) (5,1) (1,11) (5,5) (1,11) (5,6) (1,1)

U = 5 ;
(5,1) (1,4) (3,1) (1,1)

V = 7 ;
(5,5) (1,4) (3,2) (1,1)

CALL S1 (U , V , M) ;
(2,3) (5,10) (1,2) (5,1) (1,11) (5,5) (1,11) (5,6) (1,3) (1,1)

ENP ;
(2,6) (1,1)

SUBPOUTINE S1 (INTEGER : X , Y , M) ;
(2,23) (5,10) (1,2) (2,14) (1,12) (5,8) (1,11) (5,4) (1,11) (5,9) (1,3) (1,1)

M = X + Y + 2.7 ;
(5,9) (1,4) (5,8) (1,5) (5,4) (1,5) (4,1) (1,1)

ENS ;
(2,7) (1,1)

Section 4. The Syntax Analyzer

The functions of the syntax analyzer are as follows:

(1) The syntax analyzer divides the input tokens into instructions.

(2) For each instruction, the syntax analyzer determines whether this instruction is grammatically correct.

(3) If the instruction is not grammatically correct, it is rejected and some error message is printed out. Otherwise, it is decomposed into a sequence of basic instructions; each basic instruction is in a standard form. In our case, we shall use quadruples to represent basic instructions.

(4) Some other information obtained by the syntax analyzer will be put into the various tables of the compiler.

The first job of the syntax analyzer is to divide the input token stream into instructions. This is easy for FRANCIS because in FRANCIS, every instruction is terminated by the delimiter ";". We would like to point out that this is not the case in FORTRAN, for instance, because FORTRAN assumes that an instruction is contained in one card unless in the next card, a symbol appears in the 6th column. This makes the job of determining the termination of an instruction much harder.

Note that even in PASCAL, the separation of tokens into instructions is more complicated. For instance, we do not allow an instruction as follows:

VARIABLE INTEGER : X, Y, Z; REAL : U, V;

which is perfectly legal in PASCAL. In FRANCIS, we shall have two instructions instead:

VARIABLE INTEGER : X, Y, Z:

VARIABLE REAL : U, V;

In PASCAL, compound statements are allowed. They start with BEG and end with END. Within BEG and END, there can be a sequence of instructions, each of which is terminated with ";". The reader should realize that we deliberately designed FRANCIS to have such kind of syntax rules so that a compiler can be easily written. In FRANCIS, we only allow an IF …… THEN …… ELSE instruction as follows:

IF P AND Q THEN X = X+ 1 ELSE X=X-1;

Note that the entire statement is only one instruction. It should not be difficult for the reader to modify his FRANCIS compiler to handle an instruction as follows:

IF P AND Q THEN BEG X = X + 1 ; Y = X + 2; END;

ELSE BEG X = X + 2 ; Y = X + L; END;

Section 4.1. The Recognition of Different Types of Instructions.

In FRANCIS, instructions may be classified into the following categories:

(1)program heading instructions

(2)subroutine heading instructions

(3)variable type declaration instructions

(4)label declaration instructions

(5)dimension declaration instructions

(6)IF THEN ELSE instructions

(7)GO TO instructions

(8)assignment instructions

(9)CALL instructions

(10)I/O instructions

Each type of instruction has its own syntax rules. The syntax analyzer has to recognize each instruction and branch to a subroutine to handle this instruction. For instance, consider a GO TO instruction. This instruction must start with a reserved word GTO. This GTO must be followed by a label. This label must be of the type of an identifier. That is , it must start with a letter and followed by integers, letters or nothing. Then, finally, we expect a ";" following this label. Since we have stipulated that every label must have been declared before it is used, we

have to check whether the identifier following GTO is indeed a declared label or not. If the token following GTO is not a identifier or it was not declared as a label before, error messages will be printed.

Let us consider another case. Imagine that we have already recognized the first token to be the reserved word “VARIABLE”. We then check whether the next token is “INTEGER”, “REAL” or “BOOLEAN” If yes, we go ahead to check whether “:” follows. If no, an error message is printed. After finding “:”, we expect a sequence of distinct identifiers, none of which appeared before and each one is followed by “,”. Finally, we expect a “;” to terminate this instruction. For each identifier appearing in this instruction, we have to take some semantic action which will be explained in detail later. If the variables are not distinct, or some variable is not followed by “,” or some variable appeared before, error messages will be printed. The entire process of analyzing a variable type declaration instruction can be illustrated in Fig 4.1.

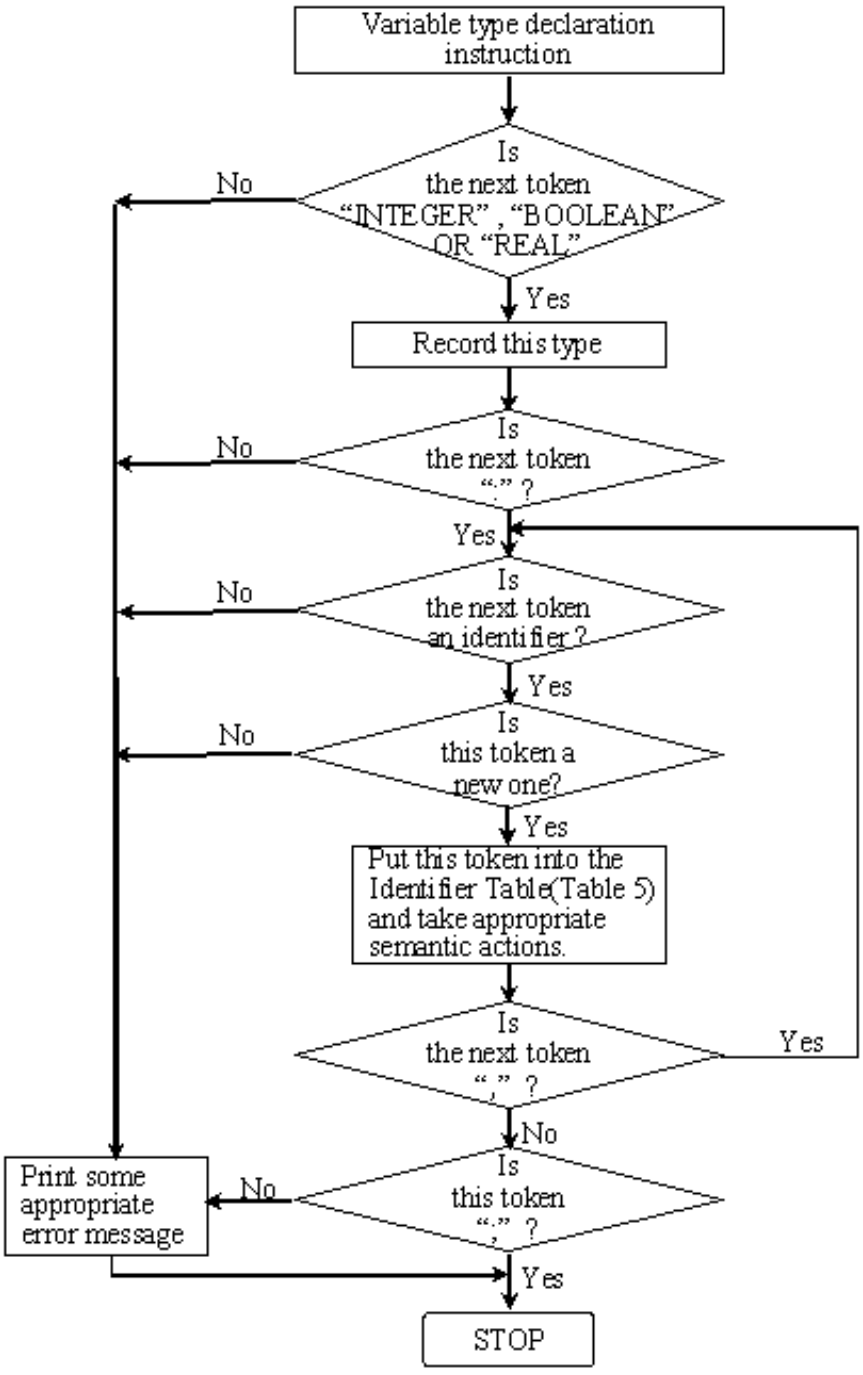

Variable type declaration instruction
Is the next token "INTEGER" , "BOOLEAN" OR "REAL"
No
Yes
Record this type
Is the next token ":" ?
No
Yes
Is the next token an identifier ?
No
Yes
Is this token a new one?
No
Yes
Put this token into the Identifier Table(Table 5) and take appropriate semantic actions.
Is the next token "," ?
Yes
No
Is this token ";" ?
No
Print some appropriate error message
Yes
STOP

Fig 4.1

The reader can see that it is absolutely necessary to recognize the type of an instruction being analyzed. Let us temporarily assume that none of our instructions starts with a label (We shall discuss the handling of labeled instructions later.).If this is the case, every instruction in FRANCIS, except the assignment instructions, starts with a reserved word. For instance, the program heading instruction starts with the reserved word PROGRAM and the IF THEN ELSE instructions start with the reserved word IF. In the following table, Table 4.1 we show how every instruction, except assignment instruction, starts with a particular reserved word. If an instruction does not start with any reserved word, then it must be an assignment instruction.

Instructions	Reserved Word
Program heading instructions	PROGRAM
Subroutine heading instructions	SUBROUTINE
Variable type declaration instructions	VARIABLE
Label declaration instructions	LABEL
Dimension declaration instructions	DIMENSION
IF THEN ELSE instructions	IF
GO TO instructions	GTO
Assignment instructions	xxx
Call instructions	CALL
I/O instructions read	INPUT
I/O instructions went	OUTPUT

Table 4.1.

Again, we like to point out here that in FORTRAN, we can not identify the type of an instruction by checking the first token, because FORTRAN has no reserved words. However, those who are familiar with COBOL or BASIC will be able to note that every COBOL or BASIC instruction begins with a reserved word and this reserved word uniquely determines the type of the instruction.

Remember that in our compiler, every token is characterized by two integers. To check whether a token is a reserved word or not, we merely have to check whether the first integer is 2 because Table 2 contains all of the reserved words.

We assumed that labels did not exist in the previous discussions. This is, of course, not a valid assumption because we do allow labels to appear in instructions. Actually, given an instruction, our first job is to check whether the beginning token is a label or not.

Note that labels were not detected in the lexical analysis phase; they were merely identified as identifiers. Nevertheless, in FRANCIS, it is agreed that a label must have been declared before it is used. In other words, there must be an instruction I declaring label L before it is used. After instruction I is processed by the syntax analyzer, as the reader will see later, the syntax analyzer will put a note in the Identifier Table to declare that I identifier L is a label. This is done by putting an appropriate entry in the type entry of the Identifier Table. Later, whenever we want to

know whether any token is a label or not, we merely have to check this token in the Identifier Table. If it is indeed a label, the type entry of this token in the Identifier Table will declare this fact.

At the top of the syntax analyzer, the type of each instruction is determined as follows:

(1)Check the first token of an instruction. If this is a reserved word, the type of this instruction is determined by this reserved word.

(2)If the first token is not a reserved word, check whether this token is a label. If this is not a label, this instruction must be an assignment instruction.

(3)If the first token is a label, the type of this instruction is determined by the second token. If the second token is a reserved word, then the type of this instruction is determined by this token. Otherwise, it must be an assignment instruction.

Section 4.2. The Processing of the Program Heading Instruction.

There can be only one program heading instruction in the entire program and this instruction must appear as the first instruction. The reserved word corresponding to this instruction is PROGRAM. After this instruction is processed, the heading of this program will now be classified as an identifier and the into the Identifier Table. In this Identifier Table, we do the following:

(1)In the subroutine entry of the Program heading in the Identifier Table, we have nothing. Later, whenever we notice that nothing appears in the subroutine entry of an identifier table, we know that this identifier is the name of a subroutine. Note that the main program can also be considered as a subroutine.

(2)In the pointer entry, we put a “1” there. We do this to indicate that the quadruples corresponding to this program start from the first location of the Quadruple Table. The Quadruple Table is now illustrated in Fig. 4.2. For

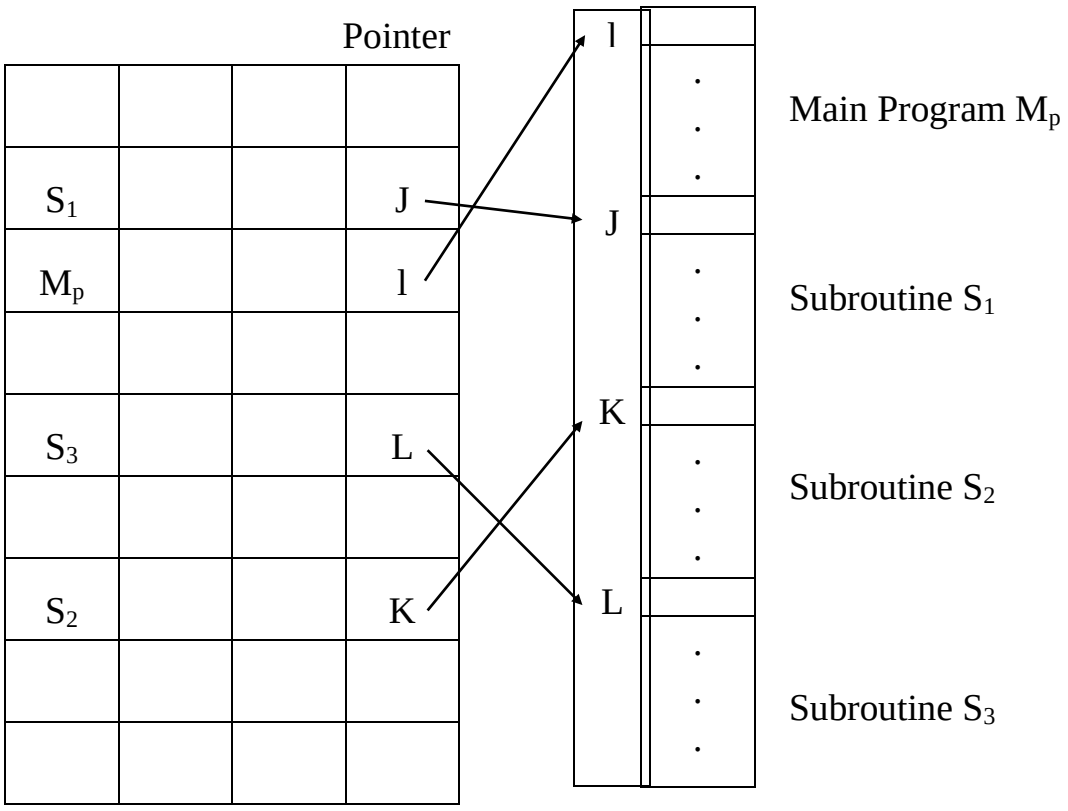


Fig. 4.2

every subroutine, a sequence of quadruples are generated and stored consecutively in the Quadruple Table. In the Identifier Table, we have a pointer for every subroutine pointing to the starting point of the corresponding sequence of quadruples.

Section 4.3. The Processing of variable Type Declaration Instructions.

The reserved word to identify a variable type declaration instruction is VARIABLE. After processing such an instruction, the following actions will be taken:

(1)Let X_1，X_2，……，X_n be variables declared in this instruction. In the subroutine entry of X_i in the Identifier Table, put a pointer pointing to the subroutine it appears in. That is, suppose X_i belongs to Subroutine S which appears in Location L_s in the Identifier Table, then put L_s in the subroutine entry corresponding to X_i in the Identifier Table.

(2) Put a code indicating the type of X_i in the type entry of X_i in the Identifier Table. We may arbitrarily set the codes as follows:

Array	1
Boolean	2
Character	3
Integer	4
Label	5
Real	6

If the syntax analyzer finds out that the type of X_i is boolean, “2” will be put into the type entry of X_i in the Identifier Table.

(3)For each variable X_i, a quadruple is generated in the Quadruple Table. This quadruple will be used by the code generator later to assign the necessary space. Let the location in the Identifier Table occupied by X_i be L_i. Then the quadruple corresponding to X_i is ((5,L_i), , ,).

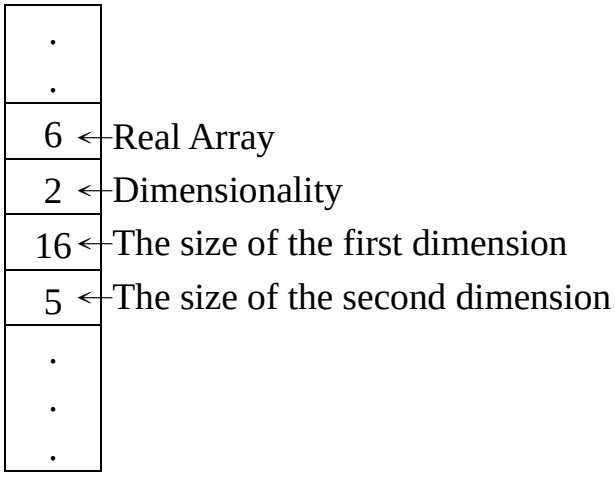
Let us imagine that a certain variable X is declared as a real variable and the location occupied by this variable is 15 in the Identifier Table. Then the quadruple corresponding to this variable is ((5,15), , , ,). In the type entsy, we have a “6”, indicating that it is a real variable. The code generator, after noting the 2-tuple (5,15), will examine the 15th location of the Identifier Table. It will notice the existence of “6” in the type entry and thus will generate the following assembly language codes:

X DS 2

because a real number variable will occupy two memory locations.

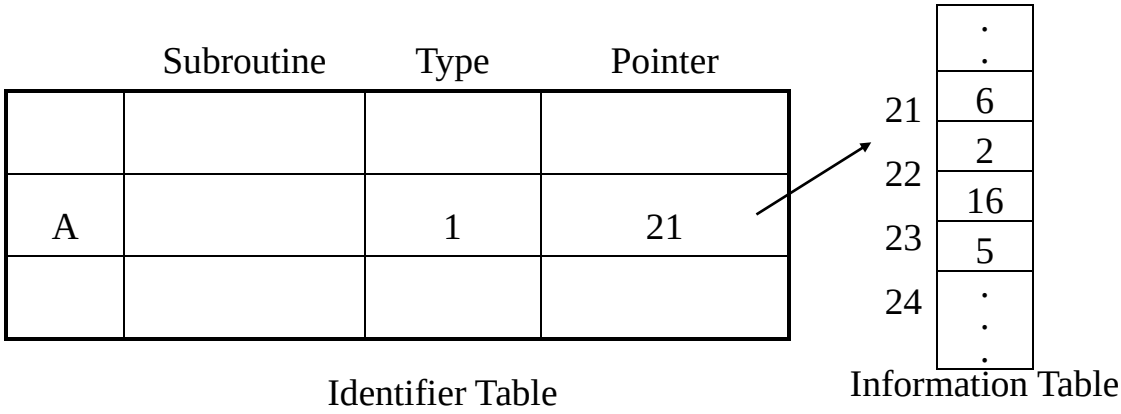
Section 4.4. The Processing of Dimension Instructions.

A dimension declaration instruction declares arrays. For every array declared, we have to record the type of the array, the dimensionality of the array and the size of the array in each dimension. For instance, suppose the size of the array in each dimension. For instance, suppose the size of a real array is declared as (16,5). We now have to record the fact that the array is real, the dimensionality of this array is 2, the size of the first dimension is 16, and the size of the second dimension is 5. To store this information, we need a new table, called Information Table. We shall denote is Table 7. Table 7 is a one-dimensional table . In the above case, the information will be stored as follows:



Information Table

In the Identifier Table, a pointer will be made to point to the Information Table as follows:



Let us summarize the actions taken by the syntax analyzer to handle a dimension declaration instruction as follows:

(1)Suppose A is declared as an array. In the subroutine entry of A in the Identifier Table, Put an appropriate pointer to indicate the subroutine it belongs to.

(2)In the type entry of A in the Identifier Table, put “1” into it, indicating that A is an array.

(3)Suppose that the type code of an array is j, the dimensionality of A is i and the sizes of A corresponding to different dimensions of A are S_1，S_2 …… ，S_i respectively. Put S_1，S_2，S_2，…...S_i into the Informations Table (Table 7). Suppose, in the Information Table , the next available location is N, then the information j. i，S_1，S_2，….. ，S_i will be put into locations N, N+1, …. , N+i+l respectively. Put N into the pointer entry of A in the Identifier Table.

(4)In the Quadruple Table , generate a quadruple in which the first element identifies the location occupied by Array A in the Identifier Table.

That is , if the array occupies the 35th location of the Identifier Table, then the quadruple generated will be

((5,35), , ,)

The information Table will not only contain information concerning arrays. As we shall see later ,when we want to store other types of information, such as the information concerning parameters of a subroutine, we can also use this table. We may view the Information Table as a warehouse where many people can store things. The only thing that we have to do is to keep pointers pointing to the location where information is stored.

The reader may wonder why we don’t keep separate tables for different types of information. For instance, why don’t we keep on table to store information concerning arrays and another table for information concerning subroutines? Note that if we keep several tables, we have to reserve quite a lot of memory locations for these tables. That is, we have to keep one array for each table. Since we do not know how large the size of each table should be, we are forced to keep many rather large arrays and later we may find out that many arrays are largely empty and a lot of memory space is wasted. By keeping one information table, we have decreased the degree of memory space wasting.

There is another important point to be made here. Suppose we write the compiler in FORTRAN. We certainly can not have an array which will store integers as well as characters. The Information Table, Table 7, contains only integers. The this is possible is due to the clever design to make sure that every piece of information is represented by integers.

Section 4.5 The Processing of Label Instructions.

Label instructions are easy to identify; they all start with the reserved word LABEL. After recognizing a label instruction, the following actions will be taken.

(1)Arrange a pointer to be put into the subroutine entry of this label in the Identifier Table.

(2)In the type entry, put "5" into it to indicate that it is a label.

Section 4.6 The Processing of Labeled Instructions.

In FRANCIS, labels must be declared before they are used. Besides, they must not be numbers. These instructions make life easier when have to process labeled instructions.

Let us imagine that we have instructions as follows:

.
.
.
LABEL L100;
.
.
.
L100 X=Y+Z;

In this case, the following events must take place:

(1)The lexical analyzer recognizes L100 as an identifier and hashes it into the Identifier Table.

(2)The syntax analyzer recognizes the reserved word LABEL and consequently decides that Identifier L100 must be a label. It then indicates so by putting "5" in the type entry of L100 in the Identifier Table, declaring that L100 is a label.

(3)When the syntax analyzer encounters L100 again, it first checks through the Identifier Table and notices that it is a label. Suppose that in the Quadruple Table, the quadruple corresponding to

L100 X=Y+Z;

starts from location N of the Quadruple Table, then put N into the pointer entry of L100 in the Identifier Table.

(4)Suppose L100 is hashed into some location M in the Identifier Table, the first quadruple corresponding to this label instruction is

((5,M), , , ,).

Later, when the code generator encounters this quadruple, it will realize that the next assembly language codes next generated should be labeled. For

X=Y+Z;

the assembly language codes are

```
LD  Y
AD  Z
ST  X
```

Because this instructions is labeled, the code generator will generate a special label, say L3, in this case. Thus we may have

```
L3  LD  Y
    AD  Z
    ST  X
```

Imagine that we later encounter an instruction as

GTO L100;

In this case, the syntax analyzer checks the pointer entry of L100 in the Identifier Table and notices that the starting quadruple corresponding to the instruction labeled by L100 starts from location N in the Quadruple Table. It will therefore generate a quadruple expressing "GTO N" when the code generator generates the following assembly language code:

```
JMP  L3
```

We hope that the reader can appreciate the rule that we do not use integers to represent labels, as is done in many languages. In FRANCIS, every integer recognized in the lexical analysis phase is indeed a number and it is therefore appropriate to put it into the Integer Table. If we also used numbers to represent labels, then we have to do some syntax analysis even in the lexical analyzer which means that the lexical analyzer and the syntax analyzer can not be separated clearly.

Section 4.7 The Processing of Assignment Instructions.

We are now going to discuss the most interesting and important part of the syntax analyzer: the processing of assignment instructions. As we noted before, assignment instructions are the only instruction which do not start with a reserved word and must contain an "=" sign.

In the following discussion, we shall first assume that the assignment instructions contain arithmetic expressions only. The method that we present can be easily extended to handle assignment instructions with logical expressions. Besides, let us temporarily assume that arrays do not appear in the assignment instructions.

Consider the instruction

X=Y+Z;

It is very easy for the syntax analyzer to recognize the operator "+" and generate the following quadruple:

(+,Y,Z,X).

It is appropriate to point out here that every entry is a token represented by a pair of integers. The plus sign "+" is represented by (1,5) because it occupies the 5th location of Table 1, the Delimiter Table. Variables X,Y and Z will all be represented by pairs of integers and the first integer of each pair is 5 because they are all identifiers (The Identifier Table is Table 5.). Thus the quadruple may look like:

((1,5),(5,3),(5,10),(5,13)).

if X, Y and Z are stored in Locations 13,3 and 10 respectively in the Identifier Table.

To simplify our discussion, we shall use (+,Y,Z,X) instead of the above representation.

Unfortunately, a general assignment instruction is rather complicated as it may involve several operations and even parentheses. A typical assignment instruction may be

X=Y+U*V;

In this case, we have to remember that we should perform the multiplication first and addition next. That is, we should generate the following quadruples:

(*,U,V,T1)

(+,Y,T1,X)

where T1 is a temporary variable.

Our question is :How does the computer know that we should first execute multiplication and then addition ?

This problem can be solved by converting an assignment instruction into its Reverse Polish Notation. To obtain the Reverse Polish Notation of an assignment instruction involving arithmetic operations only, we shall first assume that there is an ordering of operators as follows:

↑
*,/
+,-
(,)
=

The procedure to generate the Reverse Polish Notation for an assignment instruction is as follows:

(1)The symbols of the source string move left towards the junction(See Fig. 4.3).

(2)When an operand (In our case, an operand is a variable.) arrives, it passes straight through the junction.

(3)When an operator (In our case, an operator is a delimiter) arrives, it pauses at the junction. If the stack is empty, this incoming operator slides down into the stack.

Otherwise, it looks at the top of the stack. While the priority of the incoming operator is not greater than that of the operator at the top of the stack, the operators in the stack will pop out and move to the left of the junction one by one. After the above process is finished, the incoming operator is pushed into the stack.

(4)(always goes down to the stack.

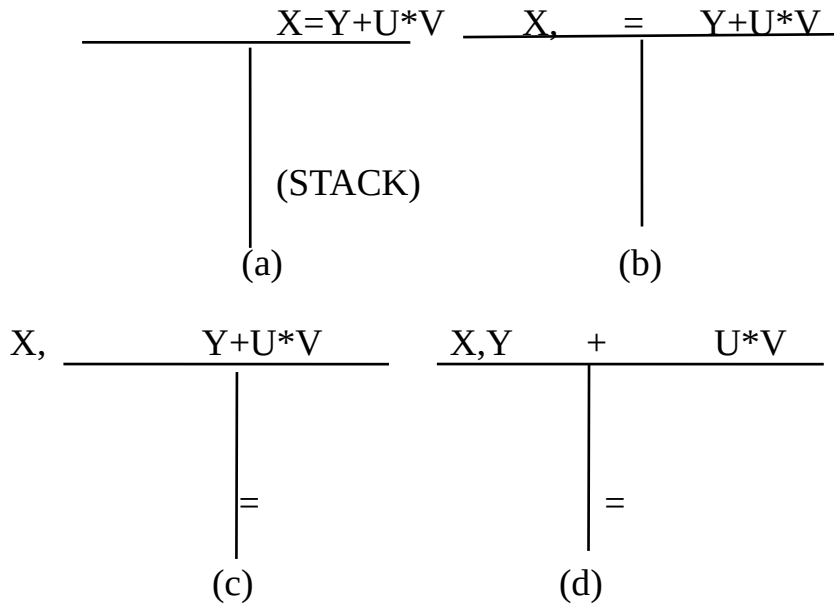
(5)When) reaches the junction, it causes all of the operators in the stack back as far as (to be shunted out and then both parentheses disappear.

(6)When all symbols have been moved to the left. pop out all of the operators in the stack to the left junction.

Let us consider the case of

X=Y+U*V;

The steps of obtaining the Reverse Polish Notation of the above instruction is illustrated in Fig. 4.3.



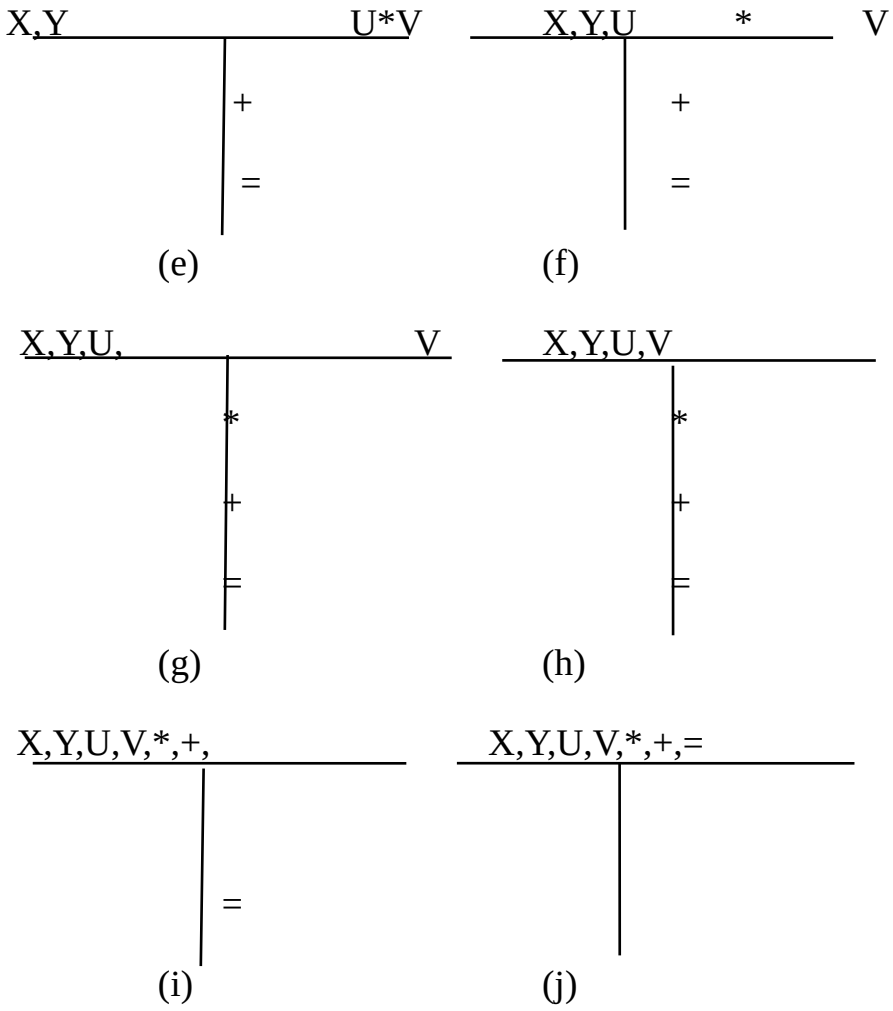


Fig. 4.3

The transformed Reverse Polish Notation is thus

X,Y,U,V,*,+,=.

After transforming an assignment instruction into its Reverse Polish Notation, one can decompose it into a sequence of basic instructions very easily. The procedure is as follows: (We assume that every operator is a binary operator here. It is easy to extend this method to handle unary operators.)

Step 1. Let i=1.

Step 2. Pick the first operator from left to right. Let this operator be O_i.

Step 3. Scan back and pick up two operands immediately preceding O_i . Let these two operands be V_1 and V_2 . Generate a basic instruction $O_i(V_1, V_2) = T_i$.

Step 4. If the last symbol has already been scanned, stop. Otherwise, let i=i+1 and go to Step 2.

Let us consider the assignment instruction.

X=Y+U*V;

again. The Reverse Polish Notation of the above instruction is

X,Y,U,V,*,+,=.

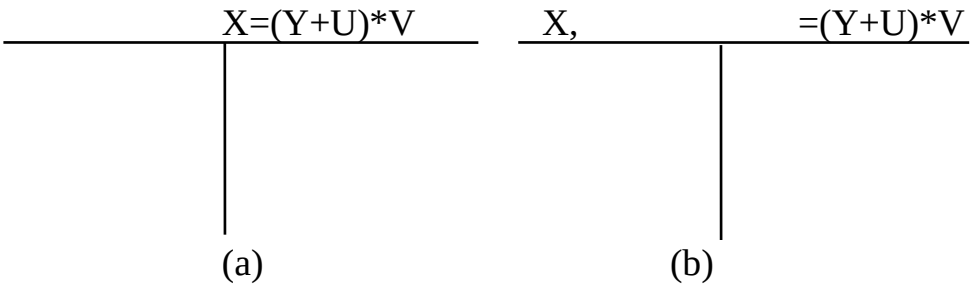
A sequence of basic instructions are now generated:

	Basic Instruction	Quadruple
X,Y,U,V,*,+,=	T1=U*V	(*,U,V,T1)
X,Y,T1,+,=	T2=Y+T1	(+,Y,T1,T2)
X,T2,=	X=T2	(=,T2, ,X)

Let us consider another example:

X=(Y+U)*V;

The Reverse Polish Notation of the above instruction is obtain as follows:



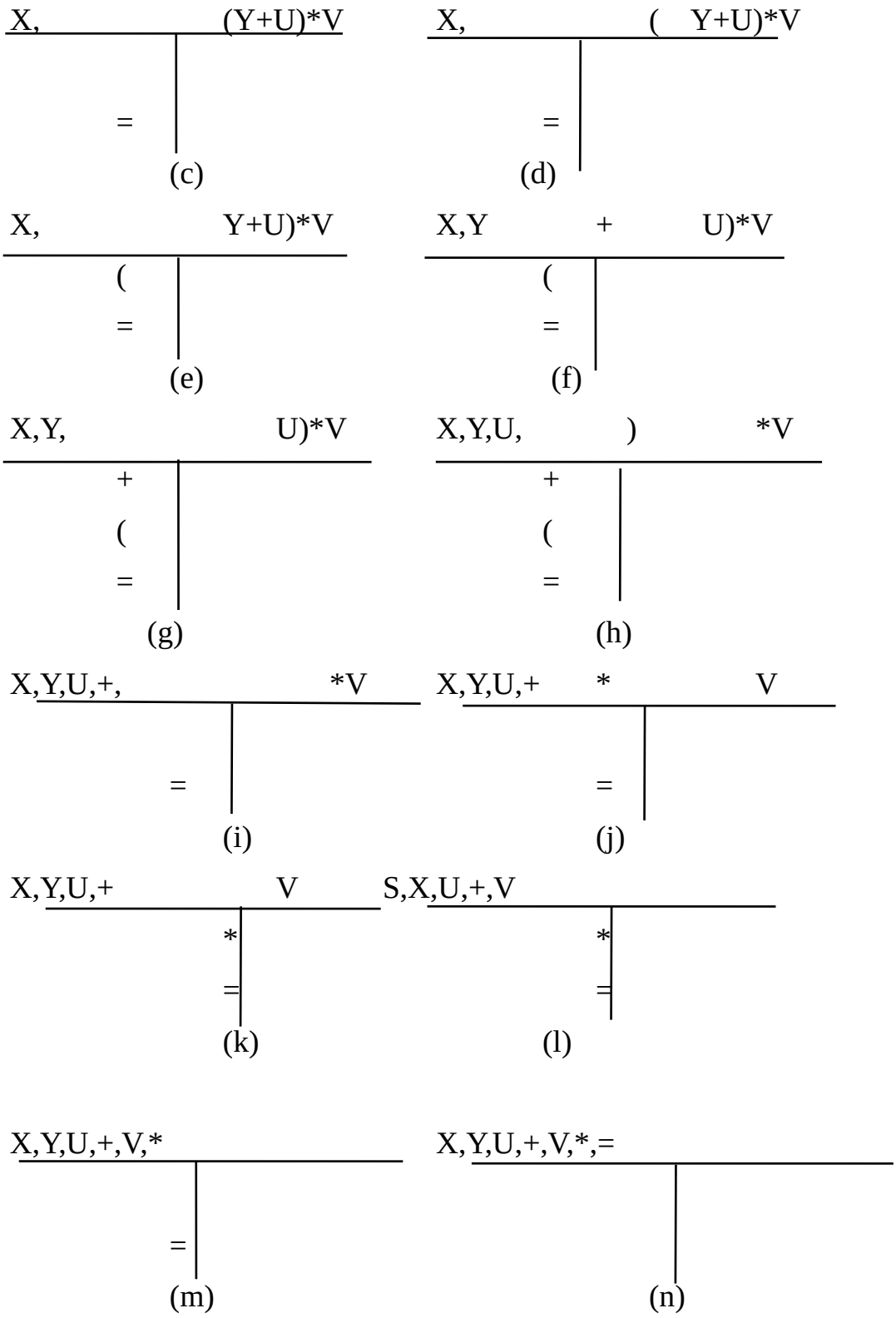

X, (Y+U)*V
=
(c)
X, (Y+U)*V
=
(d)
X, Y+U)*V
(
=
(e)
X,Y + U)*V
(
=
(f)
X,Y, U)*V
+
(
=
(g)
X,Y,U,) *V
+
(
=
(h)
X,Y,U,+, *V
=
(i)
X,Y,U,+ * V
=
(j)
X,Y,U,+ V
*
=
(k)
S,X,U,+,V
*
=
(l)
X,Y,U,+,V,*
=
(m)
X,Y,U,+,V,*,=
(n)

Fig. 4.4

The Reverse Polish Notation for

X=(Y+U)*V

is thus

X,Y,U,+,V,*,=

A sequence of basic instructions are generated as follows:

	Basic Instructions	Quadruples
X,Y,U,+,V,*,=	T1=Y+U	(+,Y,U,T1)
X,T1,V,*,=	T2=T1*V	(*,Y,T1,V,T2)
X,T2,=	X=T2	(=,T2, ,X)

In the previous discussion, we have only dealt with assignment instructions with arithmetic operations. In FRANCIS, we allow logical assignments as well. That is, we may have an instruction as

X=P AND Q OR R;

where P,Q and R are boolean variables. It should be easy for the reader to see that the above instruction can be transformed into its Reverse Polish Notation as follows:

X,P,Q, AND, R, OR,=

if we agree that AND should be performed before OR. The basic instructions are generated as below:

	Basic Instructions	Quadruples
X,P,Q,AND,R,OR,=	T1=P AND Q	(AND,P,Q,T1)
X,T1,R,OR,=	T2=T1 OR R	(OR,T1,R,T2)
X,T2,=	X=T2	(=,T2, ,X)

Section 4.8 The Processing of Instructions Containing Arrays.

In the previous discussions, we assumed that arrays did not exist. If they do exist. If they do exist, we have to take special action. Consider the following instruction:

A(I)=B(2,3)+4;

In this case, the parameters appear because of the arrays A and B and they must be eliminated first.

Note that arrays must have been declared before they are used. Let us assume that A and B appear in the 7th and the 16th locations in the Identifier Table respectively. Then A is represented by (5,7) and B is represented by (5,10). In the Identifier Table, both of these identifiers will be indicated to be arrays. The syntax analyzer, when it encounters the symbol (5,7), will now examine the 7th location of Table 5 and will find out that this identifier represents a array. Immediately, it will branch to a subroutine to handle this array.

Let us consider a simple example first:

X=B(I,J)+4;

In this case, we want to replace B(I,J) by a temporary variable. Let us assume that Array B was defined as follows:

DIMENSION INTEGER:B(3,3);

The traditional method of memory management will arrange the array as follows:

B(1,1)

B(2,1)

B(3,1)

B(1,2)

B(2,2)

B(3,2)

B(1,3)

B(2,3)

B(3,3)

That is , B(I,J) will be the ((J-1)M+I)th location of the B array if the dimensionalities of B are M and N respectively. Thus, to replace B(I,J), we have to generate the following quadruples:

(-,J,1,T1)	T1=J-1
(*,T1,M,T2)	T2=T1*M
(+,T2,I,T3)	T3=T2+I

Finally, we need the following quadruple:

(=,B,T3,T4),

which is interpreted as

T4=B(T3).

The assembly language codes corresponding to

(=,B,T3,T4)

are

```
LD      B+T3-1
ST      T4
```

The entire sequence of quadruples for

X=B(I,J)+4;

are

(-,J,1,T1)	T1=J-1
(*,T1,M,T2)	T2=T1*M
(+,T2,I,T3)	T3=T2+I
(=,B,T3,T4)	T4=B(T3)
(+,T4,4,T5)	T5=T4+4
(=,T5, ,X)	X=T5

Let us consider an example in which an array appears at the left side of the "=" sign:

A(I)=B(I,J);

In this case, B(I,J) is replaced by a temporary variable the same way as we discussed before. Suppose the temporary variable is T4. We then have the situation as follows:

A(I)=T4;

The reader can see that we have three kinds of quadruples where the operators are all "=":

(=,Y ,X)	X=Y
(=,A,I,X)	X=A(I)
(=,X,A,I)	A(I)=X.

That the Code Generator will not be confused is due to the fact that A is a declared array while X and Y are not.

We discussed how to handle the 2-dimensional integer arrays. It should be easy for the reader to extend the above idea to three, or four dimensional arrays. It should not be difficult to handle real arrays either.

Section 4.9. The Processing of IF THEN ELSE Instructions.

The IF THEN ELSE instructions can be easily recognized because they all start with the reserved word IF. A general format of this kind of instruction is

IF E THEN IN_1 ELSE IN_2;

where E is a logical expression and IN_1 and IN_2 are two simple instructions.

The syntax analyzer, after analyzing the above instruction, would generate the following quadruple:

(IF,P,Q P_1, QP_2)

where P is a logical variable, evaluated to 1 or 0, depending on the expression E and the values of its variable during the run time and QP_1 is the starting quadruple corresponding to IN_1. This quadruple will be interpreted as "If P is true, execute the codes corresponding to QP_1; otherwise, execute codes corresponding to QP_2." Suppose the quadruple corresponding to IN_1 starts from the 16th location of the Quadruple Table (Table6), then QP_1 will be represented by (6,16).

Let us illustrate these ideas by an example. Consider

IF P THEN X=X+1 ELSE X=X+2;

The syntax analyzer will generate the following quadruples, assuming that the quadruples will start from the 15th location of the Quadruple Table.

15	(IF,P,(6,16),(6,18))	
16	(+,X,1,T1)	X=X+1
17	(=,T1, ,X)	
18	(GTO, , ,(6,20))	
19	(+,X,2,T2)	X=X+2
20	(=,T2, ,X)	

When the first quadruple of the above sequence is generated, the syntax analyzer knows that QP_1 will start from the 16th location, but it does not

know where QP_2 will start. Therefore, the syntax analyzer will have to wait for the generation of the 19th quadruple. When this quadruple is generated, the syntax analyzer comes back to Quadruple 15 and fills in (6,19). The situation is the same for Quadruple 18. When this quadruple is generated, the syntax analyzer does not know how long the sequence of quadruples corresponding to X=X+2 will be. It therefore waits until all of the quadruples corresponding to X=X+2 are generated.

Let us consider a more complicated case:

IF P AND Q THEN X=X+1 ELSE X=X+2;

In this case, we first have to evaluate P and Q, assuming that P and Q are boolean variables. The quadruples for this instruction with be as follows:

15	(AND,P,Q,T1)
16	(IF,T1,(6,17),(6,20))
17	(+,X,1,T2)
18	(=,T2 ,X)
19	(GTO, , ,(6,22))
20	(+,X,2,T3)
21	(=,T3, ,X)

Quadruple 15 means that the value of T1 is the AND of P and Q. the assembly language codes for this quadruple are:

```
1,D     P
AND     Q
ST      T
```

In general, a logical expression can be as complicated as follows:

X GT Y OR Y LT Z AND U EQ V.

To evaluate the above expression, we may use the Reverse Polish Notation to transform it into the following form (We have assumed that we should evaluate AND operations before OR operations.):

X,Y,GT,Y,Z,LT,U,V,EQ,AND,OR

The sequence of quadruples generated will be

(GT,X,Y,T1)

(LT,Y,Z,T2)

(EQ,U,V,T3)

(AND,T2,T3,T4)

(OR,T1,T4,T5)

Section 4.10. The Processing of Go TO Instructions.

It is very simple to process GO TO instructions. For

GTO LX;

the syntax analyzer first has to find out whether LX is a label or not. This can be done by checking through the Identifier Table. If it has not been declared as a label or it was declared to be something else, we produce an error message. If LX is indeed a label, we check whether

this instruction with the label has appeared before. If it appeared, we may generate the following quadruple:

(GTO, , ,QP_X)

where QP_X is the starting quadruple corresponding to the instruction labeled with LX. Otherwise, we have to wait until the syntax analyzer encounters the instruction with the label. If this instruction never appears or it appears more than once, we also produce error messages.

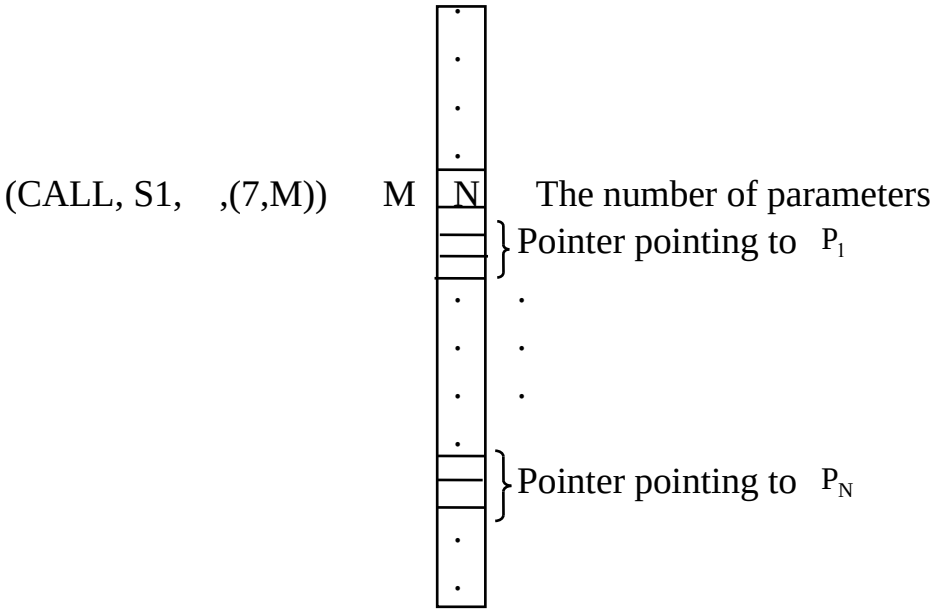
Section 4.11 The Processing of CALL Instructions.

When a CALL instruction is encountered, the syntax analyzer will have to record the name of the subroutine being called and the arguments appearing in the instruction. To do this, we again use the Information Table.

A CALL instruction is of the following form:

CALL S1($P_1, P_2, ..., P_N$);

The quadruple corresponding to the above instruction will be of the following form:



Information Table (Table 7)

Since we store identifiers and numbers in tables, we have to use two numbers to represent each P_i. Let us assume that our CALL instruction is

CALL S1(W,136,A,57,9);

Suppose that S1, W and A occupy the 36th, 15th and 27th locations of the Identifier Table (Table 5) respectively. Suppose 136 occupies the 3rd location of the Integer Table (Table3), and 57.9 occupies the 2nd location of the Real Table (Table 4). Note that the reserved word CALL occupies the 3rd location of the Reserved Word Table (Table 2). Let us assume that the next available location in the Information Table is 59. The quadruple corresponding to the above instruction will be as follows:

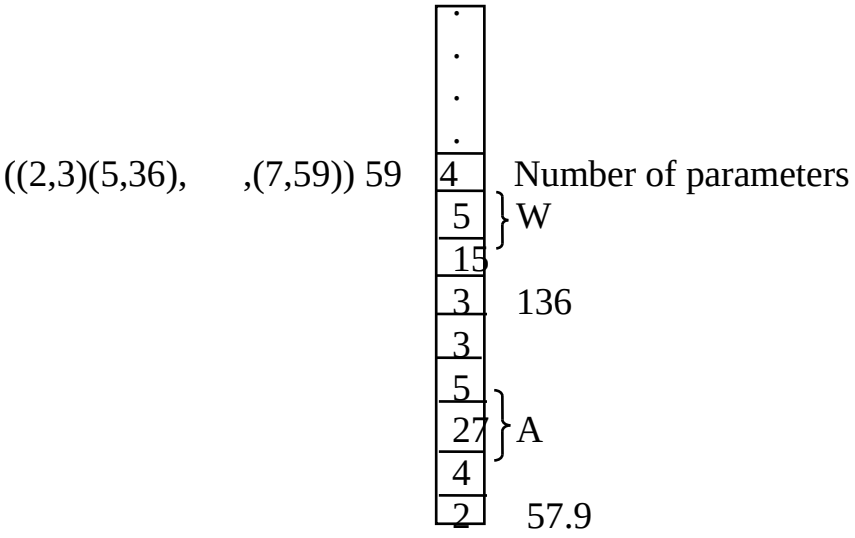


Table 7

Section 4.12. Examples.

In this section we shall present two examples designed to show how quadruples are generated and how the various tables will look after the syntax analyzer is executed.

Example 4.1

```
    PROGRAM A1;
    VARIABLE INTEGER:X,Y,I;
    DIMENSION INTEGER:A(12);
    LABEL L91, L92;
    I=1;
    X=5;
    Y=11;
L91 IF  X GT Y  THEN  GTO L92  ELSE X=X+2;
    A(I)=X;
    I=I+1;
    GTO L91;
L92 ENP;
```

SUBROUTINE TYPE POINTER

1					
2	I	5	4		
3					
4					
5	A1			1	→ (Quadruple Table)
6					
7	A	5	1	1	→ (Information Table)
8	X	5	4		
9					
10					
11	Y	5	4		
12					
13					
14	L91	5	5	10	→ (Quadruple Table)
15	L92	5	5	18	(Quadruple Table)

Identifier Table (Table5)

1	12
2	1
3	5
4	11
5	2

Integer Table (Table3)

1	4	Array A
2	1	
3	12	

Information Table (Table 7)

1	((5, 8),	,	,	)	X
2	((5,11),,	,	)	Y	
3	((5, 2), ,	,	)	I	
4	((5, 7), ,	,	)	A	
5	((5,14), ,	,	)	L91	
6	((5,15), ,	,	)	L92	
7	((1, 4),(3,2),	,(5,2))	I=1		
8	((1, 4),(3,3),	,(5,8))	X=5		
9	((1, 4),(3,4),	,(5,11))	Y=11		
10	((2,10),(5,8),(5,11),(0,1))	T1=X GT Y			
11	((2,12),(0,1),(6,12),(6,13))	IF T1 GO TO 12, ELSE GO TO 13			
12	((2,11),,	,(6,18))	GTO L92		
13	((1, 5),(5,8),(3,5),(0,2))	T2=X+2			
14	((1, 4),(0,2),	,(5,8))	X=T2		
15	((1, 4),(5,8),(5,7),(5,2))	A(I)=X			
16	((1, 5),(5,2),(3,2),(5,2))	I=I+1			
17	((2,11),,	,(6,10))	GTO L91		
18	((2,6),	,	,	)	L92 ENP

Quadruple Table (Table 6)

Example 4.2.

```
PROGRAM A2;
VARIABLE INTEGER: I,J,K;
DIMENSION INTEGER:    A(20),B(4,5);
I=2;
J=3;
CALL A3(I,J,K);
A(K)=B(I,J)+2.7;
ENP;
SUBROUTINE A3(INTEGER:X,Y,K) ;
VARIABLE INTEGER:Z
Z=6;
K=(X-Z) ↑ 2+Y;
ENS;
```

	Subroutine	Type	Pointer	Type	Pointer	
1						
2	X	11	4			
3	I	6	4			
4						
5	A	6	1	1	→	(Information Table)
6	A2			1	→	(Quadruple Table)
7	K	6	4			
8	K	11	4			
9	B	6	1	4	→	(Information Table)
10	J	6	4			
11	A3			16	→	(Quadruple Table)
12						
13	Y	11	4			
14						
15	Z	11	4			
16						

Identifier Table (Table 5)

1	4		
2	1		Array A
3	20		
4	4		
5	2		Array B
6	4		
7	5		
8	3		
9	5	I	
10	3		
11	5	J	A3(I,J,K)
12	10		
13	5		
14	7	K	

1	20	1
2	4	
3	5	
4	2	
5	3	
6	6	
7	1	

Integer Table
(Table 3)

2.7

Real Table
(Table 4)

Information Table
(Table 7)

Quadruples:

1	((5,3), , ,)	I	
2	((5,10), , ,)	J	
3	((5,7), , , ,)	K	
4	((5,5), , , ,)	A	
5	((5,9), , , ,)	B	
6	((1,4),(3,4), ,(5,3))	I=2	
7	((1,4),(3,5), ,(5,10))	J=3	Program A2
8	((2,3),(5,11),,(7,8))	CALL A3(I,J,K)	
9	((1,6),(5,10),(3,7),(0,1))	T1=J-1	
10	((1,7),(0,1),(3,2),(0,2))	T2=T1*4	
11	((1,5),(5,3),(0,2),(0,3))	T3=I+T2	
12	((1,4),(5,9),(0,3),(0,4))	T4=B(T3)	
13	((1,5),(0,4),(4,1),(0,5))	T5=T4+2.7	
14	((1,4),(0,5),(5,5),(5,7))	A(K)=T5	
15	((2,6), , ,)	ENP	
16	((5,2), , ,)	X	
17	((5,13), , ,)	Y	
18	((5,8), , ,)	K	
19	((5,15), , ,)	Z	
20	((1,4),(3,6), ,(5,15))	Z=6	
21	((1,6),(5,2),(5,15),(0,6))	T6=X-Z	Subroutine A3
22	((1,9),(0,6),(3,4),(0,7))	T7=T6 ↑ 2	
23	((1,5),(0,7),(5,13),(0,8))	T8=T7+Y	
24	((1,4),(0,8), ,(5,8))	K=T8	
25	((2,7), , ,)	ENS	

Section 5. Code Generation

The function of a code generator of a compiler is to generate assembly or machine language code. Conceptually, this is the easiest phase of the compiler writing. In practice, it is always quite difficult to write because it is heavily machine-dependent. One must be able to master the target assembly language, the operating system, the linkage editor, the file system and many other features of the target machine. In order to simplify our discussion, we assume that we are implementing our compiler on a very simple machine whose assembly language instructions are shown below:

ADD	add
AND	and operation
ARG	argument
BN	branch if negative
BP	branch if positive
BZ	branch if zero
DC	define constant
DIV	divide
DOS	define storage
EXP	exponential
END	end
INC	increase by one
JMP	unconditional jump

JSB	jump to subroutine
LD	load
LDI	load-indirect
MPY	multiply
NOP	no operation
OR	or operation
RTN	return
ST	store
STI	store-indirect
SUB	subtract
XI	input
XOR	exclusive of operation
XP	output

We shall first assume that our language is non-recursive.

The code generation of recursive language will be discussed later. Let us consider Example 4.2. the code generator will first examine Table 3 and 4 and generate the following instructions:

```
I1    DC    20
I2    DC    4
I3    DC    5
I4    DC    2
I5    DC    3
I6    DC    6
I7    DC    1
R1    DC    2.7
```

Then the code generator will examine the quadruples one by one and generate a sequence of instructions for each quadruple. For instance, consider the first quadruple. This quadruple is

((5,3), , ,)

After examining this quadruple, the code generator will find out that this quadruple calls for the following instruction:

I DS 1

because the third element in Table 5 corresponds to an identifier I which is an integer occupying one word of memory space.

Consider Quadruple 4. This quadruple is

((5,5), , ,)

We shall first find out that (5,5) corresponds to an array, as indicated by its type in Table 5. The other characteristics of this array can be found by its pointer pointing to Table 7. In Table 7, we will note that Array A is a one-dimensional array and occupies twenty memory locations. Thus, the code generator will generate the following assembly language instruction:

A DS 20

Similarly, the assembly language instruction for

Quadruple

((5,9), , ,)

will be

B DS 20

Consider Quadruple 6, which is

((1,4),(3,4), ,(5,3)).

(1,4) will be found to correspond to "=", (3,4) will be found to correspond to "2" and (5,3) will be found to correspond to "I". Thus this quadruple will be translated into the following assembly language instructions:

```
LD    I4
ST    I
```

similarly, Quadruple 11 will be translated into

```
LD    I
ADD   T2
ST    T3
```

Section 5.1 The Code Generation for Subroutine Calls

Let us consider Quadruple 8 of Example 4.2:

((2,3),(5,11), ,(7,8)).

The first 2-tuple corresponds to CALL and (5,11) corresponds to A3. Thus the first assembly language instruction corresponding to the above quadruple will be

JSB A3.

A subroutine call will usually invoke the passing of arguments. The arguments of this jump-to-subroutine instruction can be found by examining the last element of the quadruple. Since the last elements is (7,8), we examine the 8th element of Table 7. This will inform us that this calling instruction involves three parameters, namely I, J, and K.

There are many methods of passing the arguments. Let us introduce the simplest kind first.

Method 1--Call by Address

In this method, after the

JSB A3

instruction, we shall have a sequence of instructions to store the addresses of arguments which are to be passed. Thus the instructions will be as follows:

```
JSB     A3
ARG     I
ARG     J
ARG     K
```

After these instructions, an instruction corresponding to

```
T1=J-1
```

will follow. That is, we shall have a sequence of assembly language instructions as follows:

```
        JSB     A3
L1      ARG     I
L2      ARG     J
L3      ARG     K
L4      LD      J
        SUB     ONE
        ST      T1
```

When the program is finally executed, the addresses of I,J and K will be stored in L1, L2 and L3 respectively.

Let us now take a look as to how the arguments are passed to the called subroutine A3, in this case. At the very beginning of the instructions corresponding to A3, we shall have a define-storage instruction

```
A3      DS      1
```

When the calling instruction

```
JSB     A3
```

is executed, the machine will store the address immediately after the JSB instruction, namely L1, into the location A3. This action serves to build a link between a calling instruction and the called subroutine. As will become clear later, the arguments as well as the return address are all passed through this mechanism.

Note that in Example 4.2, we have to pass parameters I,J, and K to X,Y and K (These three variables are local ones.) respectively. To pass I to X, we may use the following instructions:

```
LDI     A3
ST      X
```

The load-indirect instruction stores L1 into the register. Since the address of I is stored in L1, the store instruction puts the address of I into location X.

The other parameters can be transferred in a similar way For instance, the transferring of J to Y will be as follows:

```
INC     A3
LDI     A3
ST      Y
```

To transfer the last argument, we do the following:

```
INC     A3
LDI     A3
ST      K
```

At the end of the code corresponding to the subroutine A3, there should be a return instruction:

```
INC     A3
RTN     A3
```

Note that the final A3 point to L4 and control will eventually return to L4, as expected.

It is important to note that for every parameter inside a subroutine, which is not an ordinary variable, its address, not its value, is stored. Thus every time that the subroutine refers to a parameter, it should use an indirect command so that the command will actually be referring to the variable in the calling procedure. For example, if Y is a parameter, then the statement

```
Y=Y+1;
```

is translated into

```
LDI     Y
ADD     ONE
STI     Y
```

Through this way, the argument Y is changed in the calling program which is desired.

Method 2.--Call by Value

The above method of passing parameters during subroutine calls is called “call by address” because the address of a variable is used. Since the address is passed, it may sometimes cause trouble. For instance, let us consider the calling of AADD1(X) again.

```
SUBROUTINE ADD1(I:Integer);
I=I+1;
RETURN;
END;
```

There is no problem if we have the following instructions:

```
X=2;
CALL ADD1(X);
OUTPUT X;
```

In this case, “3” will be printed out.

Suppose we have the following program:

```
CALL ADD1(2);
J=2;
OUTPUT J;
```

In this case, after

```
CALL ADD1(2)
```

is executed, the constant “2” will become “3”. Therefore

J will not be set to 2. Instead, it will become 3, which may be very baffling to many naive users.

In contrast to “call by address”, “call by value” avoids the above problem because only the temporary values of the parameters are passed. Thus the changes to the parameters will be local within the subroutine. They will not have any lasting effect.

To implement the “call by value” method, we shall store the values, instead of the addresses, of the parameters, immediately after the jump-to-subroutine instruction. For instance, in the above case, we shall have

```
        JSB     A3
L1      DC      I
L2      DC      J
L3      DC      K
```

Within the subroutine , we may use the load-indirect instruction to transfer the parameters to local variables. All instructions in the subroutine would then refer to these local variables.

Actually, if the language uses call by address, it is easy to simulate call by value by explicitly having the subroutine copy the arguments into local variables. The subroutine would then only refer to these copies, and never to the original variables.

One simple way to protect the variables from the undesirable effect of call by address is to simply rename the variables. For example, we

may do the following:

```
Y=X;
CALL ADD1(Y);
```

or the subroutine may use a local variable Y with the statement

```
Y=X;
```

as the first statement and all X's replaced by Y's in the program body. Then the calling programs would not need Y=X and X is not changed by the subroutine call.

In the following, we shall assume that call by address is used unless stated otherwise.

Section 5.2 Code Generation for Recursive Languages

So far we have assumed that within a Subroutine S, there is no calling of S. If a subroutine calls itself, what will happen？ To make sure that the reader understands this problem, let us first consider the following problem.

Suppose that we like to compute the factorial function defined as below:

FACTORIAL (1)=1

FACTORIAL (N)=N*FACTORIAL (N-1)

where N is a positive integer.

The following FRANCIS program correctly computes the above function:

```
        PROGRAM FACTORIAL;
        VARIABLE INTEGER: N;
        VARIABLE REAL: FAC;
            INPUT N;
            CALL FSB(N,FAC);
AD1         OUTPUT FAC;

            END;
SUBROUTINE FSB(INTEGER: I,REAL:R);
            VARIABLE INTEGER:M;
            LABEL L1;
        IF I LE 1 THEN GTO L1;
            M=I-1;
```

```
                CALL FSB(M,R);
AD2             L1  IF I LE 1 THEN R=1.0 ELSE R=R*I;
                ENS;
```

We have now denoted two addresses: AD_1 and AD_2. AD_1 is the return address for CALL FSB in the main program and AD_2 is the return address for CALL FSB within itself. Imagine that the first CALL FSB is executed. In this case, AD_1 is stored indirectly as the return address. Then, within this subroutine, we have a CALL FSB instruction. This time, AD_2 is the return address. Since there is only one memory location prepared for the return address of Subroutine FSB, AD_2 will necessarily override AD_1, in some sense. This is quite undesirable because we know that the control has to go eventually back to AD_1.

Many programming languages, such as FORTRAN, solve the above mentioned problem by simply prohibiting a subroutine from calling itself. A programming language which allows a subroutine to call itself is called a recursive programming language. ALGOL, LISP and PASCAL are all famous recursive programming language. We would like to emphasize here that a programming language can be easily made to be recursive. In fact, the authors have used many FORTRAN compilers which can handle recursive calls.

Let us now see why our program does compute the factorial if our compiler is appropriately implemented. Imagine that

N is equal to 3. The following actions should take place:

(1)FSB is first called, with N being equal to 3 and FAC unknown. The return address is AD_1.

(2)Inside FSB, I is now set to be 3. Since it is not less than or equal to 1,M is set to be 2. FSB is called again and the return address is set to be AD_2. It is important to note that I will be equal to 3. To simplify the discussion, let us denote this return address as AD_{21}.

(3)Again, inside FSB, I is now equal to 2, M is set to be 1 and FSB is called. The return address is set to be AD_2 with I equal to 2. To simplify the discussion, let us now call this return address as AD_{22}.

(4)Finally, inside FSB, I is equal to 1. Control goes to AD_2 and R is computed to be equal to 1.0.

(5)Control then goes to AD_{22}. Since I is equal to 2, R is computed to be equal to 2.

(6)Control then goes to AD_{21}. Since I is equal to 3 and R is equal to 2, R is computed to be equal to 3.

(7)The value of FAC now becomes 6 and control goes to AD_1.

There is another problem in recursive program language compiler design. In a non-recursive language compiler, a variable is assigned a fixed memory location. For a recursive language compiler, this obviously can not be done. Consider the variable I within FSB. When the first time the subroutine is called, I is equal to3. This value has to be

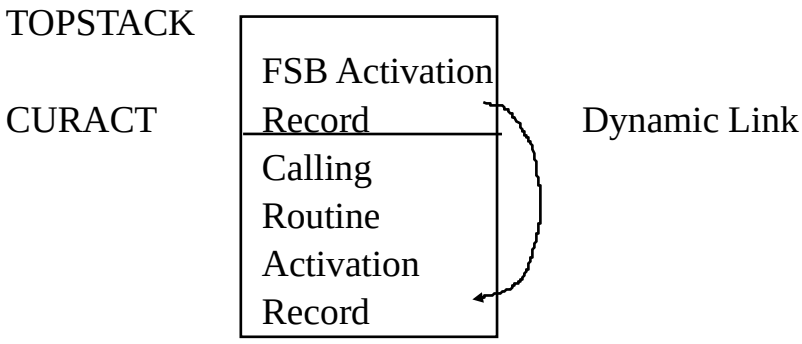
saved in some way because we are going to use it later when control comes back to AD_2. If I is assigned to a fixed location, we shall have serious problems.

To facilitate these recursive language properties, we may use a stack. During the run time, each time a subroutine is called, an activation record containing arguments and the appropriate return address is pushed into a stack. To build a link between the calling instruction and the called subroutine, a global variable pointing to the current activation record may be used. That is, this global variable changes each time a subroutine is called. All of the return addresses and parameters can be found by using this global variable. Let us now denote this global variable CURACT (current activation record). We shall also use another global variable TOPSTACK to denote the top of the stack.

Consider the above factorial program with the input N equal to 3. When the first

CALL FSB

is executed, an activation record will be pushed into the stack as shown below:



We shall note that the activation record may easily be of different lengths. Therefore, we shall store the stack as a linked list, so that we can pop it out later. The link to pointing to the calling procedure's activation record is called the dynamic link.

What should be stored in the activation record? Evidently, the following must be included:

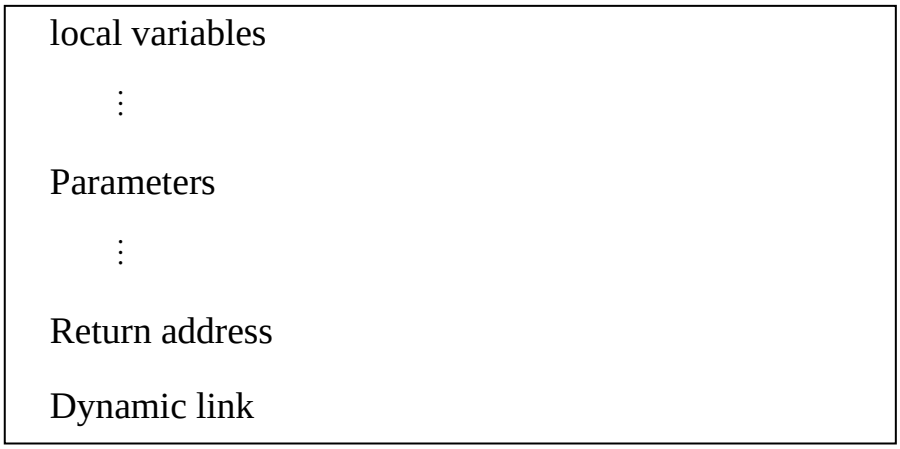
(1)The return address.

(2)The parameters which should be passed.

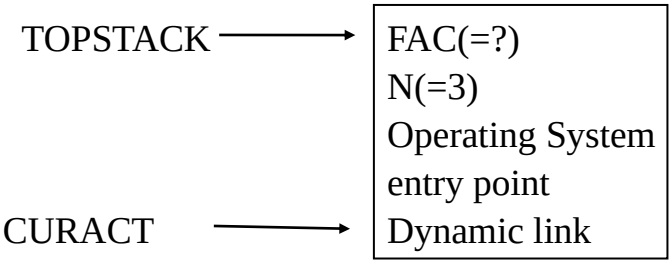
(3)The local variables.

(4)The dynamic link.

We may arrange an activation record as follows:

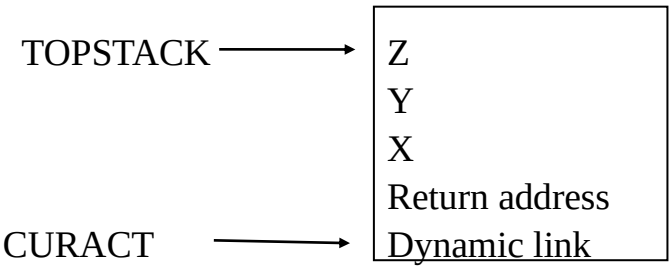


For instance, when the program is executed, the activation record will appear as below:



Note the difference between a non-recursive language compiler and a recursive language compiler. In a recursive language compiler, the variables are assigned to locations in an activation record which is pushed into a stack. As will be explained later, every time a subroutine call is made, it is necessary to put all of its variables into an activation record and pushed into a stack.

Suppose we have an activation record as follows:



Let us consider the statement:

Z=X+Y;

We shall not use the following kind of instructions any more:

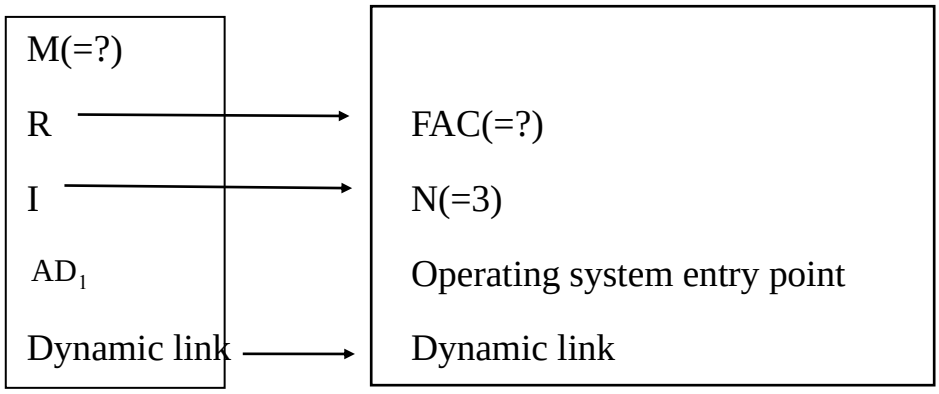
```
LD      X
ADD     Y
ST      Z
```

Instead, we shall do the following:

```
LD      CURACT(2)
ADD     CURACT(3)
ST      CURACT(4)
```

In other words, there is a table relating X,Y and Z to the offsets from the address CURACT. This information may be easily stored in the identifier Table.

When the first FSB is called, a new activation record is pushed into the stack as below:



For this new activation record, explanations are in order:

(1)Dynamic link, AD_1, I and R are put into the activation record before the

JSB FSB

instruction.

(2)M is put into the activation record by the codes inside the subroutine.

(3)Since we assume that call by address is used, I and R are pointers pointing to N and FAC respectively.

The following instructions can be used to construct the new activation record and jump to the subroutine FSB. (The reader has to remember that before the new activation record is constructed, CURCAT and TOPSTACK all refer to the old activation record.)

```
INC     TOPSTACK
LD      TOPSTACK
ST      TEMP   (The present TOPTACK is temporarily stored.)
LD      CURACT
STI     TOPSTACK (The dynamic link is now put into the new activation record.)
INC     TOPSTACK
LD      AD1
STI     TOPSTACK (AD1 is put into the new activation record.)
INC     TOPSTACK
LD      CURACT
ADD     TWO
STI     TOPSTACK(I is put into the new activation record and it point to N.)
INC     TOPSTACK
```

```
LD      CURACT
ADD     THREE
STI     TOPSTACK(R is put into the new activation record and it points to FAC.)
LD      TEMP
ST      CURACT(The new CURACT is now established.)
JSB     FSB     (Jump to FSB).
```

Note that this new activation record is only partially constructed before the

```
JSB     FSB
```

instruction because there is no way for the code generator to know the exact size of the new activation record. The job of constructing this new activation record will be completed within Subroutine FSB. In other words, within FSB, there will be instruction which put Variable M into the new activation record.

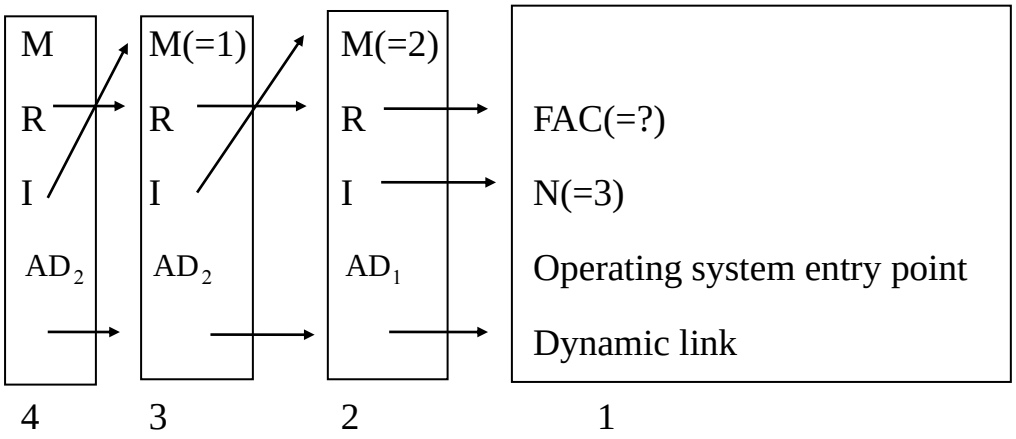
After the new activation record is constructed, whenever a variable is mentioned, the code generator uses its offset with respect to CURACT. For instance, within FSB, the instruction

```
M=I-1;
```

will be translated into the following codes:

```
LD      CURACT(2)
SUB     ONE
ST      CURACT(4)
```

Finally, the stack will appear as follows:



At this point, no more subroutine calls are necessary and control goes to AD_2. R is computed to be equal to 1. Activation Record 4 is popped out. I is pointing to 2 as registered in Activation Record 3. Activation Record is pooped out with R computed to be equal to 2. Control goes to AD_2 with R computed to be equal to 6 and FAC is set to be equal to 6 because of an indirect store for R. Activation Record 2 is now popped out. Control goes to AD_1 and FAC is printed out as equal to 6.

We would like to emphasize here that FAC has to be permanently changed after the subroutine calls. Therefore call by address must be used here.

Section 5.3 Interpreter

A compiler translates a high-level language into a low-level language to be executed later by the target machine. There are many cases where we do not want such a thorough translation because we can translate the source program into some intermediate code and directly execute this intermediate code. In such cases, we are not compiling; we are merely interpreting. After compiling, we have a machine-language object program, while we do not have that after interpreting. Everytime we want to rum this program, we have to interpret it again.

Interpreting can be explained by considering an example. Consider the following high-level instruction:

X=Y+U*V

The intermediate code for the above instruction may consist of the following two quadruples:

(*,U,V,T1)

(+,Y,T1,X)

A compiler will examine the above quadruples and then generate assembly language instructions as follows:

```
LD      U
MPY     V
ST      T1
⋮
```

Note that this is the only thing that a compiler does. It does not bother to have these instructions executed.

An interpreter will not generate these assembly language instructions. It is more straightforward; it simply executes the first quadruple by causing U to be multiplied by V and storing the result in T1. Later, it examines the second quadruple and will immediately execute it by adding Y to T1 and storing the result into X.

Since there is no object program generated, an interpreter requires much less space. This is why in many computers where memory space is limited, interpreters, instead of compilers, are used. The most famous language which uses the interpreter concept is BASIC. We began to see BASIC compilers coming out only recently.

When one is debugging a program , it will be vary useful if we have an interpretive mode incorporated into the compiler. That is, we may give the user a choice between generating an object code program and interpreting it . This will give the user a quick way to find out whether his program is correct or not. After he is satisfied with his program, he can then request that his program be compiled.

Section 5.4 The Relationship between a Compiler and the Operating System

Up to now, we have carefully avoided the issue of compiling input/output instructions. To translate an input/output instruction into machine dependent assembly language code, one can not avoid using the software facilities provided by the operating system. For instance, we never write our own input/output drivers unless it is absolutely necessary to do so , because we can and should use the drivers provided by the operating system. Besides, we usually can not skip the input/output system provided by the operating system even if we want to do so. This is due to the fact that no one can arbitrarily write data onto disks, as these actions may cause calamities. Note that there are system programs as well as other users' data on the disk, and your writing action may destroy them totally and crash the system.

Further more, a compiler may allow a user to use subroutines already defined and stored in the system. It may also allow the use of overlay facilities. In both cases, a compiler writer has to know quite a lot of the operating system under which this compiler is going to work.

In conclusion, there is no way, in practice, to separate a compiler from the operating system. In this chapter, we have not touched those problems because these issues can not be easily discussed and the

techniques used are usually not considered to be compiler writing techniques.

Section 6 The Implementation Techniques.

We have discussed compiler writing techniques. We now ask a crucial question: What language should we use to write compilers? Should we use machine languages or some high-level language?

Of course, it is much easier to use a high-level language. The only reason for using some low-level language is that it produces a more efficient compiler. But the problem of maintaining such a compiler can be quite serious.

There are several problems arising from using high-level languages to write compilers. Let us imagine that we want to write a PASCAL compiler and we have concluded that PASCAL is the ideal language to be used to write this compiler. Here is the dilemma. We do not have a PASCAL compiler yet. How can we use PASCAL to write any program?

To solve this problem, we may use some language available to us to write a very small compiler which will work for a small subset of PASCAL instructions. We shall call the language comprising this PASCAL subset PASCAL_1. Using PASCAL_1, we can write a compiler for a larger subset of instructions in PASCAL. Thus a PASCAL_2 compiler is now produced. We may repeat this process until the entire set of PASCAL instructions are included in some compiler and this compiler is a PASCAL compiler.

Let us consider another problem. Suppose we want to write a PASCAL compiler for PRIME 250. Yet, for some reason, we like to use a VAX machine. One reason may be that there is a very good PASCAL compiler on the VAX machine which means that we can use the excellent PASCAL language to write our compiler. Another possible reason is that the PRIME 250 has not arrived yet.

If the PRIME 250 is available to us, we can use it to test the correctness of our compiler. That is ,we can always execute the object code program on the PRIME 250. If the PRIME 250 is not available, we must have an emulator of the PRIME 250 on the VAX machine. We can use this emulator to test our compiler object program.

After the correctness of our compiled is established, we still have to transport it to the PRIME 250. How do we do that? Remember that this compiler is written in PASCAL which does not exist in the PRIME 250 machine. What we do is to compiler our own compiled. The result is a PRIME 250 assembly language program which is a PRIME 250 PASICAL compiler.

Can we build a compiler which works for several different machines? This appears to be a stupid question because the code generation part of a compiler is heavily machine dependent.

Yet, this still can be done and there is a trend to

build such kind of compilers. We may use some high-level language, say FORTRAN, to emulate a pseudo machine. Our compiler will generate the assembly language instructions of this pseudo machine. Since this pseudo machine is emulated by using a high-level language which works on many computers, our compiler is portable.

We may also use a high-level language to build an interpreter which executes the intermediate codes. In this case, the compiler is almost like an interpreter expect that we can save the intermediate codes as our object code output. This kind of compilers are again portable because the interpreting part may be written by a portable high-level language.

There are clear advantages of developing machine-independent compilers. In a software house where many different compilers are produced, it is a standard practice to have a set of intermediate codes which are used for different compilers. That is , compilers for different languages will use the same set of intermediate codes. Therefore, it saves us a lot of effort if one interpreter is used to execute the intermediate codes. Yet we pay a price to produce this kind of machine independent compilers. These compilers are usually quite inefficient.

Finally, let us ask a most important question. Can we automate the compiler writing process? Our answer is “No.” compiler writing can be automated only partially, not totally. We have succeeded in automating the parsing process of a compiler. That is ,after the syntax rules of a

language are specified, we can build an automatic parser which accepts a program and checks whether this program contains any syntactical error or not. However, we must note that syntax checking is not the only function of a parser; it must take semantic actions. The reader may now know that inside a compiler, the data structure is quite crucial and a compiler designer spends a larger part of his time to design the data structure. This design work can never be automated.

Bibliography

Abramson, H. (1973): Theory and Application of a Bottom-up Syntax Directed Translator, ACM Monograph Series, Academic Process, New York.

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

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

Aho, A. V. and Ullman, J.D. (1977): Principles of Compiler Design, Addison-Wesley, Reading, MA.

Backhouse, R. C.(1979): Syntax of Programming Languages, Theory and Practice, Prentice-Hall, Englewood Cliffs, N. J.

Barret, W. A. and Couch, J. D. (1979): Compiler Construction, Theory and Practice, Science Research Associates, Chicago, ILL.

Bauer, F, L. and Eickel, J. (1977): Compiler Construction-an Advanced Course, Springer-Verlag, New York.

Bornat, R. (1980): Understanding and Writing Compilers, The MacMillan Press, New York.

Callingart P, Assemblers, Compilers and Program Translation, Computer Science Press, Rockville, MD.

Cleaveland, C. and Uzgalis, R.C. (1977): Grammars for Programming Languages , Elsevier, New York.

Elson, M. (1973): Concepts of Programming Languages, Science Research Associates, Chicago, ILL.

Ershov, A. and Koster, C. H. A. (1977): Methods of Algorithmic Language Implementation, Springer-Verlag, New York.

Foster, J. M. (1970): Automatic Syntax Analysis, Elsevier, New York.

Friedman, J. (1971): Computer Model of Transformation Grammar, Elsevier, New York.

Grau, A. A., Hill, U. and Langmaack, H. (1971): Translation of ALGOL 60, Springer-Verlag, New York.

Gries, D. (1975): Compiler Construction for Digital Computers, Wiley, New York.

Hecht, M.S.(1977): Flow Analysis of Computer Programs, Elsevier, New York.

Heindel, L.E. and Roberto, J. T. (1975): Lang-Pak, an Interactive Language Design System, Elsevier, New York.

Lee, J. A. N. (1974): Anatomy of a Compiler, Van Nostrand, New York.

Lewi, P.M., Rosenkrantz, D. J. and Stearns, R. E. (1976): Compiler Design Theory, Addison-Wesley, Reading, MA.

Magnines, J. R. (1972): Elements of Compiler Construction, Prentice-Hall, Englewood Cliffs, N. J.

Ollongren, A. (1975): Definition of Programming Languages by Interpreting Automata, Academic Press, New York.

Pollack, B. W. (1972): Compiler techniques, Van Nostrand Reinhold, New York.

Rohl, J.S. (1975): An Introduction ot Compiler Writing, Elsevier, New York.

Rustin, R. (1972): Design and Optimization of Compilers, Prentice-Hall, Englewood Cliffs, N. J.

Steele, D. R. (1978): An Introduction to Elementary Computer and Compiler Design, Elsevier, New York.

Weingarten, F. W. (1973): Translation of Computer Languages, Holden-Day, San Francisco.

Williams, J. H. and Fisher, D. A. (1976): Design and Optimization of Programming Languages, Proceedings of a DoD Sponsored Workshop, Ithaca, N.Y. 1976, Springer-Verlag, New York.

Wulf, W., Johnson, R.K., Weihstock, C. B. , Hobbs, S. O. and Geschke, C. M. (1975): The Design of an Optimizing Compiler, Elsevier, New York.

Appendix

The Syntactical Rules of FRANCIS

<program>:=<main program><subroutine deck>

<main program>:=<program heading><block>ENP;

<program heading>:=PROGRAM<identifier>;

<identifier>:=<letter>{<letter>|<digit>}

<block>:=<array declaration part>
<variable declaration part>
<label declaration part>
<statement part>

<array declaration part>:={DIMENSION<array declaration>;}

<array declaration>:=<type>:<subscripted variable>,
<subscripted variable>}

<subscripted variable>:=<identifier>(<unsigned integer>)
{;<unsigned integer>}

<unsinged integer>:=<digit>{<digit>}

<type>:=INTEGER|REAL|BOOLEAN

<variable declaration part>:={VARIABLE<variable declaration>;}

<variable declaration>:=<type>:<identifier>{,<identifier>}

<label declaration part>:={LABEL<label>{,<label>};

<label>:=<identifier>

<statement part>:=<statement>{<statement>}

<statement>:=<unlabelled statement>;|
<label><unlabelled statement>;

<unlabelled statement>:=<statement I>|<if statement>

<statement I>:=<empty statement>|<assign statement>
<call statement>|<IO statement>|
<go to statement>
<empty statement>:=
<assign statement>:=<variable>=<expression>
<variable>:=<identifier>|<idnetifier>(<unsigned integer>|
<identifier>)
<expression>:=<simple expression>|<simple expression>
<relational operator><simple expression>
<relational operator> := EQ|NE|GT|GE|LT|LE
<simple expression>:=<term>|<sign><term>|
<simple expression><adding operator><term>
<adding operator>:= +|-|OR
<term>:=<factor>|<term><multiplying operator><factor>
<multiplying operator>:=*|/|AND| ↑
<factor>:=<variable>|<unsigned constant>|(<expression>)
<unsigned constant>:=<unsigned number>|<constant identifier>
<unsigned number>:=<unsigned integer>|<unsigned real>
<unsigned real>:=<unsigned integer>.{<digit>}
<sign>:= +|-
<constant identifier>:=<identifier>
<call statement>:=CALL<subroutine identifier>(<argument>
{,<argument>})
<subroutine identifier>:=<idnetifier>
<argument>:= <identifier>|<constant>

<constant>:=<unsigned constant>|<sign><unsigned constant>

<IO statement>:=INPUT<variable>

OUTPUT<variable>

<number size>:=<unsigned integer>

<go to statement>:=GTO<label>

<if statement>:=IF <condition>THEN<statement I>|

IF<condition>THEN<statement I>

ELSE<statement I>

<condition>:=<condition variable><relations><condition variable>

<condition variable>:=<variable>|<constant>

<relations>:=<relational operator>|OR|AND

<subroutine deck>:={<subroutine declaration>}

<subroutine declaration>:=<subroutine heading><block>ENS;

<subroutine heading>:=SUBROUTINE<identifier>(<parameter group>

{<parameter group>});

<parameter group>:=<type>:<parameter>{,<parameter>}

<parameter>:=<identifier>|<array>

<array>:=<array identifier>( )

<array identifier>:=<identifier>

Table of Contents