CHAPTER 6

COMPILER DESIGN

Syllabus: Compiler design: Lexical analysis, Parsing, Syntax-directed translation, Runtime environments, Intermediate and target code generation, Basics of code optimization.

6.1 INTRODUCTION

Computer understands programs written in machine language. Building program in machine language is a tedious and an error-prone task for human. So the programs are written in high-level languages which are easily understood by human. A compiler is a program that converts high-level program into low-level machine language that is understood by machine. This subject deals with how a compiler is designed and organized. While writing a compiler, the compilation process is divided into various phases. These phases operate in sequence; each phase takes input from the previous phase and provides output to the next phase.

6.2 COMPILERS AND INTERPRETERS

6.2.1 Compiler

A compiler is a special program that reads statements written in a language (called source language) and then converts them into another language (called target language). In other words, a compiler is a program which translates statements written in high-level language (i.e. Java, C#, Visual Basic) into machine-level language (Fig. 6.1). In the process of translation, a compiler also checks for errors if any.

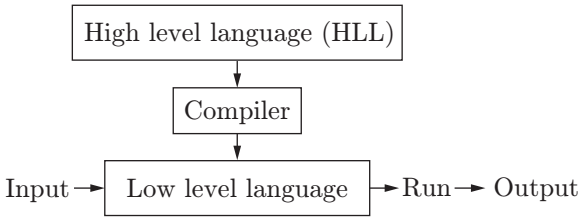


Figure 6.1 Compiler.

6.2.2 Interpreter

An interpreter takes a single instruction as input and converts it into machine-level language and shows errors in the statement if any (Fig. 6.2). It requires less memory than a compiler, and execution of conditional control statements are slower. Debugging is easier in an interpreter.

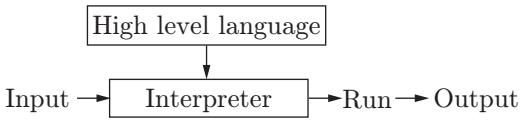


Figure 6.2 Interpreter.

6.2.3 Phases of a Compiler

A compiler takes a source program written in high-level language as input and produces an equivalent set of machine instruction as output. The compiler process is complex, so it is divided into six sub-processes which are also known as phases of a compiler. The following are different phases of a compiler (Fig. 6.3):

1. Lexical analyzer
2. Syntax analyzer
3. Semantic analyzer
4. Intermediate code generator
5. Code optimization
6. Target code

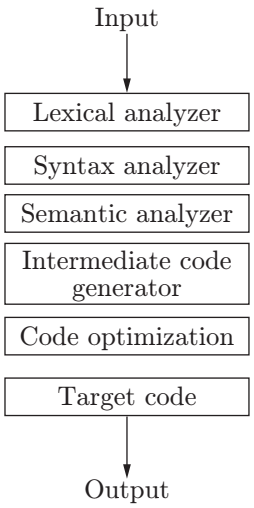


Figure 6.3 Phases of a compiler.

6.2.3.1 Lexical Analyzer

Lexical analyzer reads the source program character by character at a time and unites them into a stream of tokens. Token is a group of character which represents keywords, operators and identifiers. Character sequence formed by tokens is called "lexeme". Lexical analyzer is also known as lexer, tokenizer or scanner. If a lexical analyzer gets an invalid token, then it will generate an error. The lexical analyzer assigns an id to each token according to their occurrence. It makes entries of each identifier into the symbol table. As a lexical analyzer cannot enter all information regarding an identifier such as type and scope, the remaining information is inserted by the other phases of the compiler into the symbol table. For detail explanation about lexical analyzer, refer to Section 6.3.

1. **Symbol table:** A symbol table is a data structure which stores identifier information related to its declaration and appearance in a program such as identifier id, type, scope, value and sometimes location. Link list and hashing are the common techniques which are used to construct symbol tables.

Example 6.1

Statement written in a source program is

$$A = B + C^*25;$$

Tokens generated by lexical analyzer

$A \rightarrow$ Identifier
$= \rightarrow$ Assignment operator
$B \rightarrow$ Identifier
$+ \rightarrow$ Add operator
$C \rightarrow$ Identifier
$* \rightarrow$ Multiplication operator
$25 \rightarrow$ Constant

6.2.3.2 Syntax Analyzer or Parser

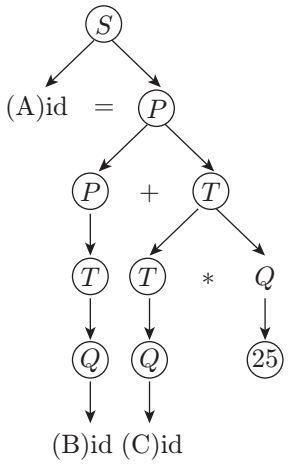
A syntax analyzer is the second and important phase of a compiler. It receives the tokens from the output of a lexical analyzer as an input. Parser performs two functions:

1. A parser checks that the input tokens from a lexical analyzer are valid or not according to the specified grammar of source language.
2. It generates a parse tree according to the given grammar to the source language. The grammar of source language is given below:

$S \rightarrow$ id $= P$
$P \rightarrow P + T/T$
$T \rightarrow T^*Q/Q$
$Q \rightarrow$ id/Integer constant

Example 6.2

Parse tree of a given problem $A = B + C$*25.



Parse tree of $A = B + C$*25

6.2.3.3 Semantic Analyzer

When the parse tree is generated by a syntax analyzer and passed as an input to the semantic analyzer, then the semantic analyzer computes the additional information related to the recognized tokens, such as operator, operand, expression or statement, and inserts that information into the symbol table. The information stored in the symbol table is frequently used by the other phases of the compiler. During the semantic analysis, the type of identifier is checked. In our example, let all identifiers be float, and 25 be treated as an integer constant. If required, the semantic analyzer will perform an implicit-type conversion, and if it is not possible, then it will throw an error. This can be easily understood by the example given in Fig. 6.4.

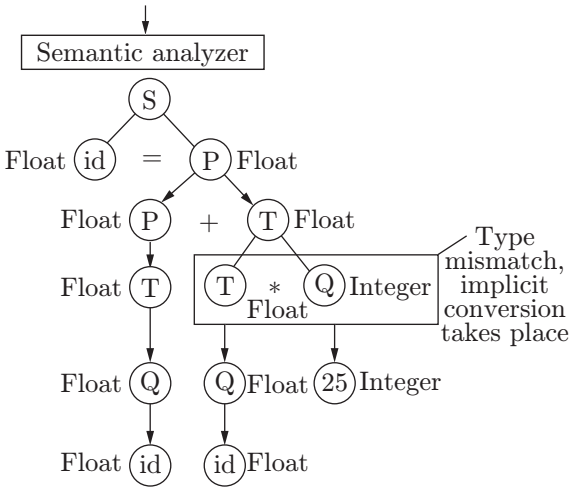


Figure 6.4 Semantic analyzer of $A = B + C$*25.

So, here in our example, implicit conversion took place. Implicit-type casting is also known as coercion.

6.2.3.4 Intermediate Code Generator

The intermediate code generator phase takes a tree as an input produced by a semantic analyzer and produces an intermediate code. The intermediate code thus generated has mainly two properties: it should be easy to produce and easy to translate into target program. An intermediate code can be represented in variety of forms. One of the forms is the three-address form, which is very similar to the assembly language in which every memory location acts like a register. The intermediate code of our example is

Source code:

$$A = B + C * 25$$

Intermediate code:

$$T_1 = C * 25$$
$$T_2 = B + T_1$$
$$A = T_2$$

6.2.3.5 Code Optimization

The code optimization phase is to reduce the size of the code and improve the performance of the code generated by an intermediate code phase. The most important part of optimized code is to minimize the amount of time taken by the code to execute and less common is to minimize the amount of memory used by the code. Optimized code of our example is

Intermediate code:	Optimized code:
$T_1 = C * 25$	$T_1 = C * 25$
$T_2 = B + T_1$	$A = B + T_1$
$A = T_2$	

6.2.3.6 Target Code

The target code is the final phase of the compiler which normally converts the input obtained from the code optimization phase into the target code (machine code or assembly code). The target code of our example is as follows:

MOV	R_1,	C
MUL	R_1,	25
MOV	R_2,	B
ADD	R_2,	R_1
STORE	A,	R_2

This is the final output of the compiler.

6.2.4 Grouping of Phases

Phases deal with the logical organization of a compiler. In an implementation, activities from more than one phase are often grouped together. Basically, phases are grouped into two parts. The first part is known as the front end, which consists of initial four phases (lexical analyzer, syntax analyzer, semantic analyzer and intermediate code) along with symbol table operations and error handling. The second part, also known as back end, consists of the last two phases (code optimization and target code). The back end also includes error handling and symbol table operation (Fig. 6.5).

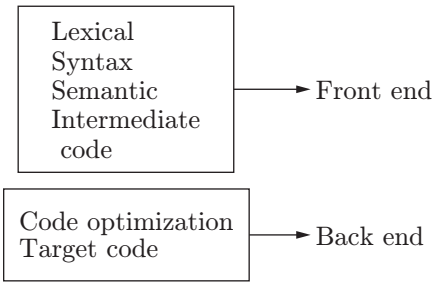


Figure 6.5 | Grouping of compiler phases.

6.2.4.1 Compiler Construction Tools

The compiler writers use software tools such as debuggers, version managers, profilers and so on. The following is a list of some useful compiler construction tools:

1. **Parser generators:** These produce syntax analyzer from context-free grammar as input.
2. **Scanner generators:** These automatically produce lexical analyzer from a specification based on regular expressions.
3. **Syntax-directed translation engines:** These produce collection of routines from parse tree, generating the intermediate code.
4. **Automatic code generators:** These take collection of rules that define the translation of each operation of the intermediate language into machine language for the target machine.
5. **Data-flow engines:** Data flow analysis is required to perform good code optimization and data flow engines facilitates the gathering of information about how values are transmitted from one part of a program to another part of that program.

6.3 LEXICAL ANALYZER

Lexical analyzer is the starting phase of a compiler. It reads the source program character by character at a time and unites them into a stream of tokens. Tokens consist of identifiers, operators and operand. A lexical analyzer also stores the name and id of identifiers in the symbol table (Fig. 6.6).

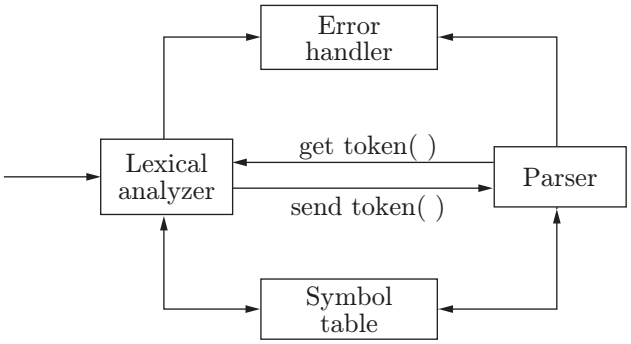


Figure 6.6 | Lexical analyzer.

6.3.1 Functions of a Lexical Analyzer

A lexical analyzer has the following functions.

1. Lexical analyzer divides the given source code or program into some meaning full words called tokens.
2. It eliminates the comment lines.
3. It finds integer and floating point constant.
4. It eliminates white-space character such as blank space and tab.
5. It helps in giving error message by providing row and column numbers.
6. It identifies identifier, keywords, operators and constants.

6.3.2 Implementation of a Lexical Analyzer

Method of implementing a lexical analyzer or scanner is regular expression and finite automaton. Some background information related to regular expression and finite automaton are given in the following sections which will help to understand how a scanner works.

6.3.2.1 Regular Expression Review

1. **Symbol:** Letters, digits and special symbols are examples of a symbol.
2. **Alphabet:** A finite set of symbols through which we build large structures. An alphabet is denoted by Σ, for example, $\Sigma = \{0, 1\}$.
3. **String:** A finite set of symbol made up of alphabets, for example, *a*, *b* are alphabets and *aaab*, *abba* are the strings.
4. **Empty string:** A string which has zero symbols, and represented by ε.

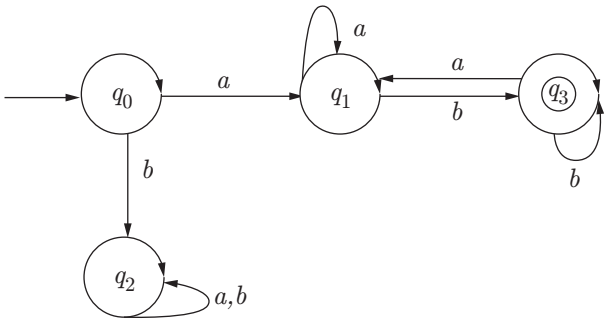
5. **Formal language:** A set of all possible strings which can be generated from given alphabets, and represented by Σ^*.
6. **Regular expression:** The rules that define the set of words that are valid tokens in a formal language. These rules are made by three operators:
   - Alternation $x|y$ (x or y)
   - Repetition x^* (x is repeated 0 or more times)
   - Concatenation xy

6.3.2.2 Finite Automata Review

Once we have all type of tokens defined by regular expression, we can create a finite automaton for recognizing them. A finite automaton has the following:

1. A finite set of states, one of which is the start state or initial state, and some (maybe none) of which are final states.
2. An alphabet Σ of possible input symbols.
3. A finite set of transitions that specifies for each state and for each symbol of the input alphabet, which defines that for an input symbol which will be the next state to go.

Example 6.3



where q_0 is the initial state and q_3 is the final state. The given finite automaton accepts a language which has a string starting with 'a' and ending with 'b'.

6.3.2.3 Recognition of Tokens

In this section, we will explain how a token is recognized by a lexical analyzer. A lexical analyzer uses finite automaton to recognize a token. Transition diagram is shown below which consists of stages and arcs. Arcs show the transition from one state to another state. Transition diagram describes the working of a lexical analyzer when called by the parser to get query for the next token. The positions are shown by the circles called states which are connected by edges. Here is a finite automaton which recognizes an integer (Fig. 6.7).

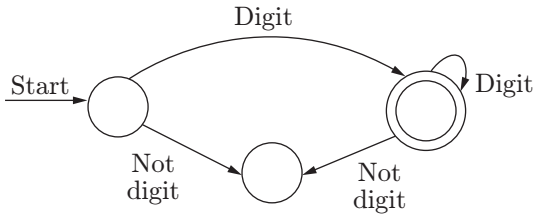


Figure 6.7 Finite automata for recognizing an integer.

Problem 6.1: Consider the following C program, find the number of tokens.

```
float average(int a, int b)
{
float c;
c = (a + b)/2;
return c;
}
```

Solution: Every token individual has been underlined. Counting the number of underlines, we have the number of tokens as 27.

```
float average ( int a , int b )
{
float c ;
c = ( a + b ) / 2 ;
return c ;
}
```

Problem 6.2: Find the number of tokens in the following C statement.

```
printf("k = %d", i);
```

Solution: The number of tokens is 7.

```
printf("k = %d", i);
```

6.4 PARSER

A parser is a part of a compiler. It takes sequence of tokens from the lexical analyzer as an input and then builds a data structure in the form of a parse tree (Fig. 6.8). A parser's main purpose is to determine if

the input data may be derived from the start symbol of the grammar. Depending upon the method how the parse tree is derived, we have two types of parsers–top-down parser and bottom-up parser (discussed shortly).

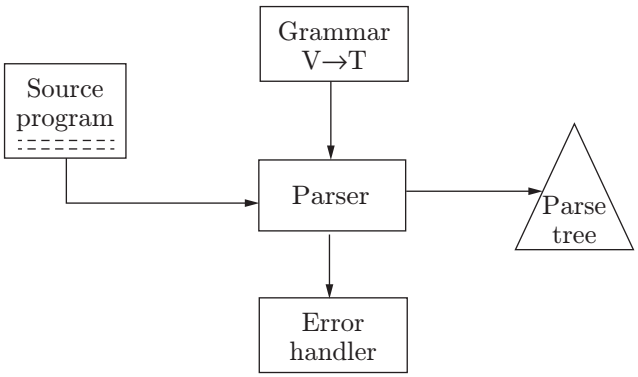


Figure 6.8 | Parser.

6.4.1 Context-Free Grammar

A grammar $G = (V_n, T, S, P)$ is said to be a context-free grammar (CFG) if the production $P = \{u \to v\}$ of G are of the form $u \to v$ and satisfy the following conditions:

1. $u \to v$, where $v \in (V \cup T)^*$, and V stands for variable and T for terminal
2. $u \to v$, where $u \in V_n$
3. $|u| \leq |v|$ (length of u is less than v)
4. Only single variable is allowed in left side (means u has single variable only)

As we know that a CFG has no context either left or right, this is the reason why it is also known as context-free.

Problem 6.3: Consider a grammar $G = (V_n, T, S, P)$ having production $S \to aSa|bSb|x$. Check the production and find the language generated.

Solution:

Let P_1: $S \to aSa$

P_2: $S \to bSb$

P_3: $S \to x$

(a, b, x) are terminals and S is a variable. As all the production are of the form $A \to \alpha$, where $\alpha \in (V_n \cup S)^*$ and $A \in V_n$, hence G is a CFG. And it will produce context-free language.

Language generated: $L(G) = \{wxw^R : w \in (a+b)^*\}$

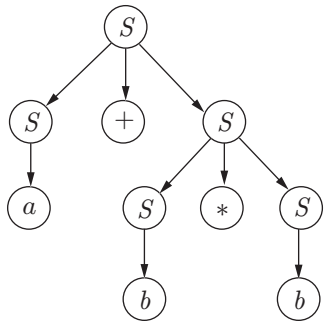
6.4.2 Derivation Tree or Parse Tree

The string generated by a CFG $G = (V_n, T, S, P)$ is represented by a hierarchical structure called tree. A derivation tree or parse tree for a CFG is a tree that satisfies the following condition:

1. If $A \to \alpha_1, \alpha_2, \alpha_3, \ldots, \alpha_n$ is a production in G, then A becomes the father of nodes, labelled $\alpha_1, \alpha_2, \alpha_3, \ldots, \alpha_n$.
2. The root has label S (starting symbol).
3. Every vertex (or node) has a label.
4. Internal nodes should be labels with variables only.
5. The leaves nodes are labelled with ε or terminal symbol.
6. The collection of leaves from left to right yields the string w.

Problem 6.4: Consider the grammar $S \to S + S|S^*S|a|b$. Construct a derivation (or parse) tree for the string $w = a + b^*b$.

Solution:



6.4.2.1 Leftmost Derivation Tree

A derivation tree is called a leftmost derivation (LMD) tree if the ordering of decomposed variable is from left to right. Thus, for generating string $w = aab$ from grammar:

$S \to AB$ (production 1)

$A \to aaA$ (production 2)

$A \to \varepsilon$ (production 3)

$B \to bB$ (production 4)

$B \to \varepsilon$ (production 5)

LMD:

$S \to \underline{A}B$ (by production 1)

$S \to aa\underline{A}B$ (by production 2)

$S \to aa\underline{B}$ (by production 3)

$S \to aab\underline{B}$ (by production 4)

$S \to aab$ (by production 5)

6.4.2.2 Rightmost Derivation Tree

A derivation tree is called rightmost derivation tree (RMD) if the ordering of decomposed variable is from right to left. Thus, for generating string $w = aab$ from the above grammar:

RMD:

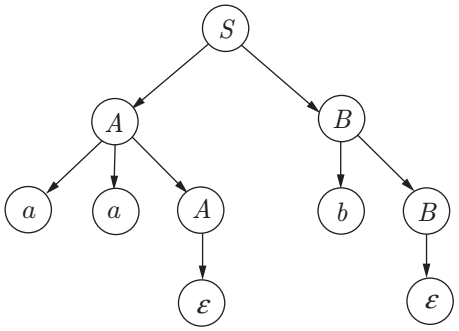
$S \rightarrow AB$ (by production 1)

$S \rightarrow AbB$ (by production 4)

$S \rightarrow Ab$ (by production 5)

$S \rightarrow aaAb$ (by production 2)

$S \rightarrow aab$ (by production 3)

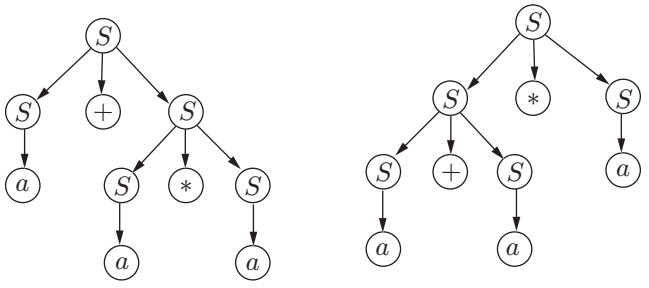


Left to right

6.4.3 Ambiguous Grammar

A grammar G is called ambiguous if for some string $w \in L(G)$, there exist two or more derivation tree (two or more LMD or two or more RMD tree). Let us consider a CFG grammar having production:

$S \rightarrow S + S|S^*S|a|b$, for string $w = a + a^*a$ have more than one LMD tree.



Note: A language (L) is called ambiguous if and only if every grammar which generates it is ambiguous. The only known ambiguous language is $\{a^n b^m c^n\} \cup \{a^n b^m c^m\}$.

Left recursion and left factoring is the major cause for a grammar to be ambiguous. But presence of these in a grammar does not mean that grammar is ambiguous; and similarly absence of these does not mean that the grammar is unambiguous.

6.4.3.1 Removal of Ambiguity

1. **Removal of left recursion:** A production of grammar $G = (V_n, T, S, P)$ is said to be left recursive grammar if it has one of the productions in the given form:

   $A \rightarrow A\alpha$, where A is a variable and $\alpha \in (V_n \cup S)^*$

   Elimination of left recursion: Let the variable A have left recursive problem as following:

   $$A \rightarrow A\alpha_1|A\alpha_2|A\alpha_3|\ldots|A\alpha_n|\beta_1|\beta_2|\beta_3|\ldots|\beta_m$$

   where $\beta_1, \beta_2, \ldots, \beta_m$ do not begin with A. Then we replace A production by:

   $$\{A \rightarrow \beta_1 A'|\beta_2 A'|\beta_3 A'|\ldots|\beta_n A'|\}$$

   where $A' \rightarrow \alpha_1 A'|\alpha_2 A'|\alpha_3 A'|\ldots|\alpha_n A'|\varepsilon$

Example 6.4

Let grammar $S \rightarrow S + S|S^*S|a|b|c$

To eliminate left recursion, the grammar S is replaced by

$$S' \rightarrow +SS'|^*SS'|\varepsilon$$
$$S \rightarrow aS'|bS'|cS'$$

2. **Removal of left factoring:** In grammar G, two or more productions of variable A are said to have left factoring if the productions are in the form:

   $$A \rightarrow \alpha\beta_1|\alpha\beta_2|\alpha\beta_3\ldots|\alpha\beta_m$$

   where $\{\beta_1|\beta_2|\beta_3\ldots|\beta_m\} \in (V_n \cup S)^*$ and does not start with α. All these production have common left factor α.

   Elimination of left factoring: Let variable A have (left factoring) production as follows:

   $$A \rightarrow \alpha\beta_1|\alpha\beta_2|\alpha\beta_3\ldots|\alpha\beta_m|\gamma_1|\gamma_2|\gamma_3|\ldots|\gamma_m$$

   where $\gamma_1, \gamma_2, \gamma_3, \ldots, \gamma_m$ and $\{\beta_1, \beta_2, \beta_3\ldots\beta_m\}$ do not contain α as a prefix, then we replace this production by

   $$A \rightarrow \alpha A'|\gamma_1|\gamma_2|\gamma_3|\ldots|\gamma_m$$
   $$A' \rightarrow \beta_1|\beta_2|\beta_3\ldots|\beta_m$$

Example 6.5

Let grammar $A \rightarrow abc|abd|abe$

To remove left factoring, we have

$$A \rightarrow abA'$$
$$A' \rightarrow c|d|e$$

6.4.4 Top-Down Parser

In top-down parser, parse tree construction starts from the root and proceeds to the leaf. Top-down parser uses

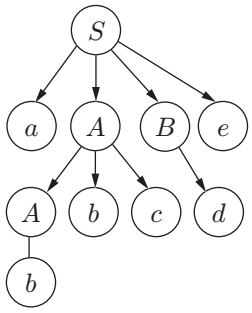
LMD for constructing the parse tree. When a variable contains more than one choice, choosing the correct production is always going to be difficult. In top-down parsing, no left recursion and no left factoring exist.

Problem 6.5: String $w = abbcde$, grammar G is given below:

$$S \rightarrow aABe$$
$$A \rightarrow Abc/b$$
$$B \rightarrow D$$

Draw a parse tree for the production of a string w with the help of grammar G.

Solution:



Parse tree for string $w = abbcde$.

Top-down parsing can be performed by the following two methods:

1. Top-down parsing with backtracking
2. Top-down parsing without backtracking

6.4.4.1 Top-Down Parsing with Backtracking

One of the most straightforward forms of parsing is recursive descent parsing (RDP). This is a top-down process in which the parser attempts to verify that the syntax of the input stream is correct as it is read from left to right. The pseudocode of recursive descent parser is given as follows:

```
RDP(S)
{
Choose a production S→x1,x2,x3,x4
for(i=1 to n)
{
if(xi is variable)
RDP(i)
else if(xi==lookahead)
increment i/p pointer
else
error(choose another production)
}
}
```

Problem with Recursive Descent Parser

1. Recursion is used to generate parse tree.
2. More time is wasted in backtracking.
3. Recursive descent parser can enter into an infinite loop if the given grammar contains left recursion.
4. Time complexity of RDP is $O(2^n)$.

6.4.4.2 Top-Down Parser without Backtracking

A predictive parser is a special class of recursive decent parser. The goal of predictive parsing is to construct a top-down parser that does not require backtracking. Predictive parsing technique can only be used for class of LL(k) grammars, where k is some integer. In LL(k) grammar, by seeing k tokens a recursive decent parser decides which production should be examined so that the LL(k) grammars are able to exclude all grammars that are ambiguous and having left recursion (Fig. 6.9).

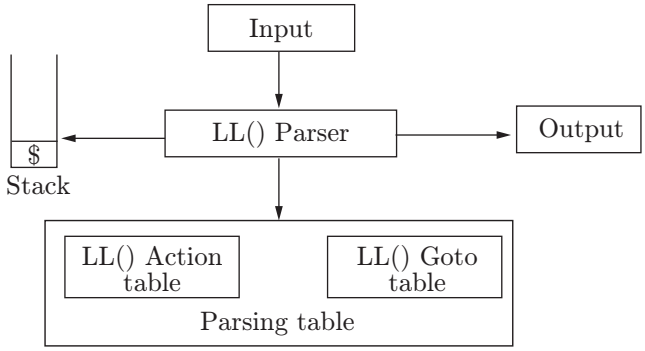


Figure 6.9 Predictive parser.

6.4.4.3 Algorithm of Predictive Parser

Let x be the top of stack and 'a' be the look-ahead symbol. Then

1. If ($x == a == \$$), then successful complete.
2. If ($x == a == \$$), then pop the stack element and increment input pointer.
3. If (x is a variable), then see the LL(1) action parsing table M.
   If $M[x_1, a] = x \rightarrow uvw$, then replace x by uvw in the reverse order.
4. If $M[x_1, a]$ = blank, then parsing error.

Problem 6.6: Draw a top-down predictive parser using the following parser table.

Grammar of given language

$$S \rightarrow (L)/a$$
$$L \rightarrow SL'$$
$$L' \rightarrow \varepsilon/,SL'$$

String $w = (a, a, a)$

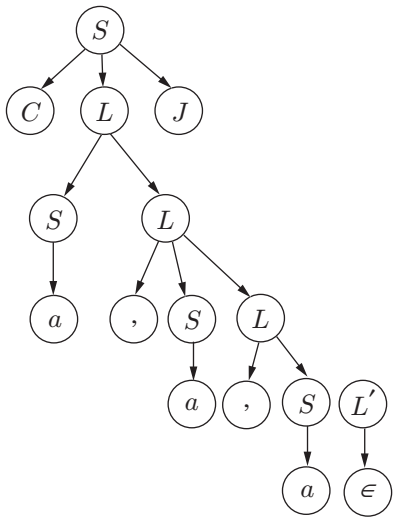
Parsing Table

	A	,	(	)	$
S	$S \rightarrow a$		S → (L)		
L	$L \rightarrow SL'$		$L \rightarrow SL'$		
L′		$L' \rightarrow, SL'$		$L' \rightarrow \varepsilon$	

Solution:

S. No.	Input	Stack	Production
1.	$(\underline{a},\ a,\ a)\$$	$\$S$	$S \rightarrow (L)$
		$\$)L\underline{(}$	
2.	$\underline{a},\ a,\ a)\$$	$\$)L$	$L \rightarrow SL'$
		$\$)L'S$	$S \rightarrow a$
		$\$)L'a$	
3.	$\underline{,}\ a,\ a)\$$	$\$)L'$	$L' \rightarrow, SL'$
		$\$)L'S\underline{,}$	
4.	$\underline{a},\ a)\$$	$\$)L'S$	$S \rightarrow a$
		$\$)L'\underline{a}$	
5.	$\underline{,}\ a)\$$	$\$)L'S\underline{,}$	$L' \rightarrow,SL'$
6.	$\underline{a})\$$	$\$)L'S$	$S \rightarrow a$
		$\$)L'\underline{a}$	
7.	$\underline{)}\ \$$	$\$)L'$	$L' \rightarrow \varepsilon$
		$\$\ \underline{)}$	
8.	$\underline{\$}$	$\$$	

By following these steps we can generate a parse tree for the string $w = (a,\ a,\ a)$.



Parse tree for the string $w = (a,\ a,\ a)$.

6.4.4.4 LL(1) Parsing Table Construction

1. **First set:** First(A) gives a set of all terminals that may begin in a string derived from A. To get first of A, start finding all strings generated by that variable in a language and then pick the first element of every string.

   Rules:
   - If $S \rightarrow ab/cd/ef$ then
     First(S) = {$a,\ c,\ e$}
   - If $S \rightarrow ab$ then
     First(S) = {a}
     First(ε) = (ε)
     First(a) = {a}, where 'a' is terminal
   - If $S \rightarrow AB$
     $A \rightarrow a$
     $B \rightarrow b$
     then First(s) = {a}
   - If $S \rightarrow aS/b$ then
     First(S) = {$a,\ b$}
   - If $S \rightarrow AB$
     $A \rightarrow a/c/d/\varepsilon$
     $B \rightarrow b$
     then First(S) = {$a,\ c,\ d,\ b$}

2. **Follow set:** Follow(A) gives a set of all terminals that may follow immediately to the right of A. ε can never be a part of any follow set.

   Rules:
   - If S is the starting symbol, then $ is also one of the element of Follow(S).
   - If $S \rightarrow aBCDE$, then
     $$\text{Follow}(C) = \text{First}(DE) = \text{First}(D)$$
   - If $S \rightarrow AB$ or $S \rightarrow ABC$ and $C \rightarrow \varepsilon$, then
     $$\text{Follow}(B) = \text{Follow}(S)$$

Problem 6.7: Find the first and the follow sets for the given grammar:

$E \rightarrow TE'$
$E' \rightarrow \varepsilon/ + TE'$
$T \rightarrow FT'$
$T' \rightarrow \varepsilon/*FT'$
$F \rightarrow \text{id}/(E)$

Solution:

Variable	First( )	Follow( )
E	id, (	$,)
E'	+, ε	$,)
T	id, (	+, $,)
T'	*, ε	+, $,)
F	id, (	*, +, $,)

Problem 6.8: Find the first and the follow sets for the given grammar:

$$S \rightarrow (L)/a$$
$$L \rightarrow SL'$$
$$L' \rightarrow \varepsilon/,SL'$$

Solution:

Variable	First( )	Follow( )
S	(, a	,,), \$
L	(, a	)
L'	,, ε	)

6.4.4.5 Algorithm for Constructing a Parsing Table

If a grammar G is given, the steps to construct a parsing table is as follows:

For each production $A \rightarrow \alpha$, do the following

1. Add $A \rightarrow \alpha$ under $M[A, b]$, where $b \in$ First(α).
2. If First(α) contains ε, then add $A \rightarrow \alpha$ under $M[A, c]$, where $c \in$ Follow(A).

Number of parsing table entries = $\{V^*(T + 1)\}$, where V is the number of variables in the given grammar and T is the number of terminals in the given grammar.

Problem 6.9: Construct the parsing table from the following grammar:

$$E \rightarrow TE'$$
$$E' \rightarrow \varepsilon/ + TE'$$
$$T \rightarrow FT'$$
$$T' \rightarrow \varepsilon/*FT'$$
$$F \rightarrow \text{id}/(E)$$

Solution:

The first and the follow sets for the given grammar are as follows:

Variable	First( )	Follow( )
E	id, (	\$,)
E'	+, ε	\$,)
T	id, (	+, \$,)
T'	*, ε	+, \$,)
F	id, (	*, +, \$,)

Number of cells in parsing table = $\{V^*(T + 1)\}$
So, table entries = $\{5^*(5 + 1)\} = 30$

Parsing table constructed with the help of the first and follow sets of given grammar:

	Id	+	*	(	)	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow$ id			$F \rightarrow (E)$		

How to Check a Given Grammar (G) Is LL(1) or Not?

The following are two ways to identify that a given grammar is LL(1) or not:

1. By constructing parsing table, if parsing table does not have more than one production entry in any of the cell then only it is LL(1).
2. By checking the following two conditions:
   - If a given grammar G does not contain null(ε) production:
     Let grammar G be
     $$A \rightarrow \alpha_1/\alpha_2/\alpha_3$$
     Then α_1, α_2, α_3 should be pair wise disjoint. That is,
     $$\text{First}(\alpha_1) \cap \text{First}(\alpha_2) = \varnothing$$
     $$\text{First}(\alpha_2) \cap \text{First}(\alpha_3) = \varnothing$$
     $$\text{First}(\alpha_3) \cap \text{First}(\alpha_1) = \varnothing$$
   - If grammar G contains null(ε) production:
     Let grammar G be
     $$A \rightarrow \alpha_1/\alpha_2/\varepsilon$$
     Find First(α_1), First(α_2) and Follow(A), they should be pair wise disjoint.
     $$\text{First}(\alpha_1) \cap \text{Follow}(A) = \varnothing$$
     $$\text{First}(\alpha_2) \cap \text{Follow}(A) = \varnothing$$

Problem 6.10: Check whether the following grammar is LL(1) or not.

$$S \rightarrow E/a$$
$$E \rightarrow a$$

Solution: In the given grammar there is no null production, so

$$\text{First}(S) = a$$

$$\text{First}(E) = a$$

According to the given condition 1, $\text{First}(S) \cap \text{First}(A) \neq \varnothing$. So the given grammar is not LL(1).

Problem 6.11: Check the following grammar is LL(1) or not.

$$S \rightarrow aABb$$

$$A \rightarrow a/\varepsilon$$

$$B \rightarrow d/\varepsilon$$

Solution: In the given grammar there is null production, so find first and follow of those productions which have null production.

$$\left.\begin{matrix}\text{First}(A) = a \\ \text{Follow}(A) = d, b\end{matrix}\right\} \text{No common element}$$

$$\left.\begin{matrix}\text{First}(B) = d \\ \text{Follow}(B) = b\end{matrix}\right\} \text{No common element}$$

Important Points

1. Every regular grammar need not be LL(1) because that grammar may contain left factoring.
2. Any ambiguous grammar cannot be LL(1).
3. If any grammar contains left factoring, then it cannot be LL(1) grammar.
4. If a given grammar contains left recursion, then it cannot be LL(1) grammar.

6.4.5 Bottom-Up Parser

In bottom-up parser, parse tree construction starts from children and proceeds to root. Bottom-up parser uses RMD for constructing the parse tree. A substring which will give reduction of string is called handle.

The difficulty with bottom-up parser is identifying the right handle which will give one required variable so that we will go to start symbol.

Problem 6.12: String $w = abbcde$, grammar G is given below:

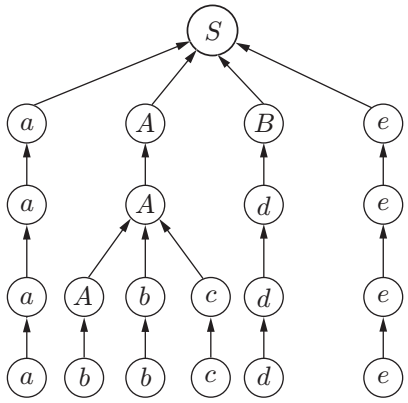
$$S \rightarrow aABe$$

$$A \rightarrow Abc/b$$

$$B \rightarrow D$$

Draw a parse tree for reduction of a string w with the help of the grammar G.

Solution: The parse tree for reduction of the string $w = abbcde$.



6.4.5.1 Classification of Bottom-Up Parser

The bottom up parser is classified as shown in Fig. 6.10.

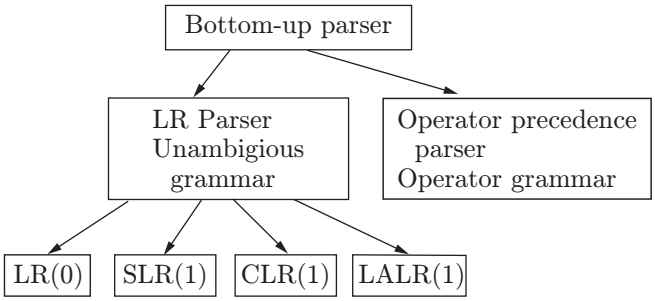


Figure 6.10 Classification of bottom-up parser.

LR Parser

In computer science, an LR parser is a type of bottom-up parser that efficiently handles context-free languages in guaranteed linear time. An LR parser reads input from left to right and produces an RMD. The canonical LR (CLR), look-ahead LR (LALR) and simple LR (SLR) parsers are common variants of LR parsers. For all types of LR parsers, parsing algorithm is same but parsing tables are different.

LR Parsing Algorithm

Let S be the state on top of the stack and 'a' be the look-ahead symbol, then

1. If action $[S, a] = S_i$, then shift 'a' and 'i', and also increment the i/p pointer.
2. If action $[S, a] = r_j$ and r_j is $\alpha \rightarrow \beta$, then pop $2|\beta|$ symbols and replace by α. If S_{m-1} is the state below α, then push Goto $[S_{m-1}, \alpha]$.

3. If action $[S, a]$ = accepted, then successful parsing.
4. If action $[S, a]$ = blank, then parsing error.

Stack Operations

1. **Shift:** Shifts the next input symbol onto the top of the stack.
2. **Reduce:** Replaces a set of grammar symbol or handle on the top of the stack with the LHS of a production rule.
3. **Accept:** Declares successful completion of the parsing.

LR(0) Parser

1. **LR(0) parsing table construction:** For constructing LR(0) parsing table we have to follow the given steps:
   - Construct augmented grammar.
     Let given grammar G be
     $$S \to AA$$
     $$A \to aA/b$$
     Augmented grammar G' will be
     $$S' \to S$$
     $$S \to AA$$
     $$A \to aA/b$$
     $S' \to S$ is called the augmented production.
   - Find closure $\{I_0 = \text{closure } (S' \to \cdot S)\}$.
     To find closure of a production I, follow these steps:
     - In closure of I, add I also to closure (I).
     - If I is $A \to B \cdot CD$ and $C \to EF$ is in the given grammar G, then add $C \to \cdot EF$ to closure (I).
     - Repeat these steps for every newly added LR(0) item.

     This can be understood clearly by the given example; it uses the above grammar to find closure.
     Closure $(S' \to \cdot S) = \{S' \to \cdot S | S \to \cdot AA | A \to \cdot aA | A \to \cdot b\}$
     Closure $(S \to \cdot AA) = \{S \to \cdot AA | A \to \cdot aA | A \to \cdot b\}$
     Closure $(A \to \cdot aA) = \{A \to \cdot aA\}$
   - Using I_0 construct deterministic finite automata (DFA).
   - Reduce DFA into LR(0) parsing table.
   - Find Goto (I, X).
     Goto (I, X) function is used to fill the second part of the parsing table, which have entries related to move after X.

Example 6.6

Consider the above grammar for the Goto function.

- Goto $(S' \to \cdot S, S) = S' \to S.$
- Goto $(S' \to \cdot AA, A) = S \to S \cdot | A \to \cdot aA | A \to \cdot b$
- Goto $(A \to \cdot aA, a) = A \to a \cdot A | A \to \cdot aA | A \to \cdot b$

Problem 6.13: Consider the following grammar and construct the LR parsing table.

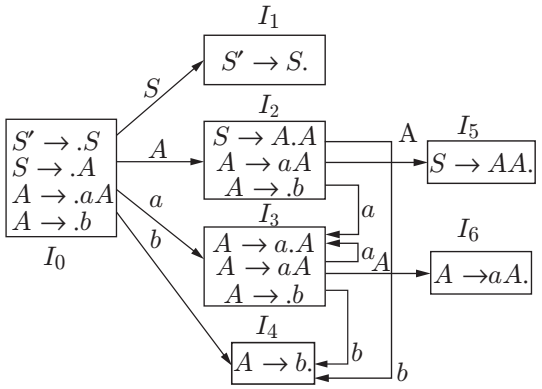
$S \to AA$ (production number 1)
$A \to aA$ (production number 2)
$A \to b$ (production number 3)

Solution:

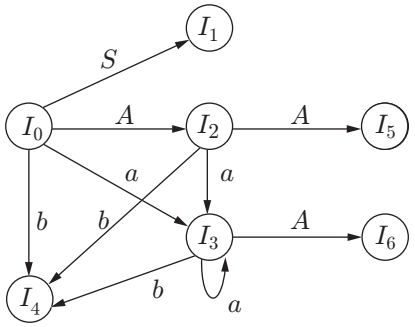
Step 1: Construct the augmented grammar.

$S' \to S$
$S \to AA$
$A \to aA$
$A \to b$

Step 2: Find closure.



Step 3: Construct DFA.



Step 4: Construct LR(0) parsing table.

- S_i denotes the shift move in the above DFA, where 'i' denotes the state number.
- r_i represents reduction move in the above DFA, where 'i' denotes the production number.

State	Action			Goto	
	A	***b***	**$**	***S***	***A***
I_0	S_3	S_4		1	2
I_1			Accepted		
I_2	S_3	S_4			5
I_3	S_3	S_4			6
I_4	r_3	r_3	r_3		
I_5	r_1	r_1	r_1		
I_6	r_2	r_2	r_2		

Problem 6.14: Check whether the following grammar is LR(0) or not.

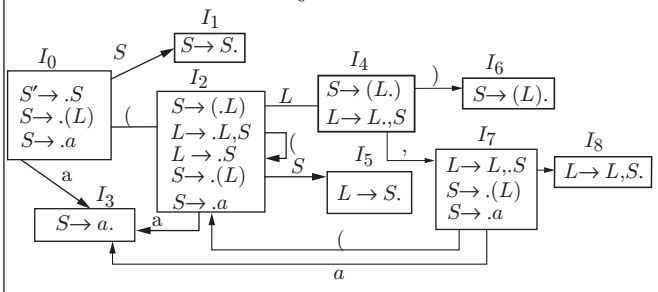
$S \rightarrow (L)$ (production number 1)
$S \rightarrow a$ (production number 2)
$L \rightarrow L, S$ (production number 3)
$L \rightarrow S$ (production number 4)

Solution:

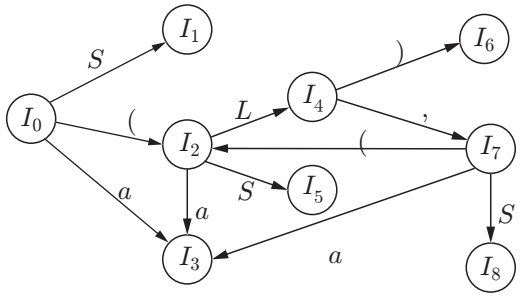
Step 1: Construct the augmented grammar.

$S' \rightarrow S$
$S \rightarrow (L)$
$S \rightarrow a$
$L \rightarrow L, S$
$L \rightarrow S$

Step 2: Find closure ($I_0 = S' \rightarrow S$).



Step 3: Construct DFA.



Step 4: Construct LR parsing table.

	Action					Goto	
State	***A***	**(**	**)**	**,**	**$**	***S***	***L***
I_0	S_3	S_2				1	
I_1					Accepted		
I_2	S_3	S_2				5	4
I_3	r_2	r_2	r_2	r_2	r_2		
I_4			S_6	S_7			
I_5	r_4	r_4	r_4	r_4	r_4		
I_6	r_1	r_1	r_1	r_1	r_1		
I_7	S_3	S_2				8	
I_8	r_3	r_3	r_3	r_3	r_3		

The given grammar is LR(0) because there are no multiple entries in the same cell.

Important Points

1. If there are two reductions on any state of DFA, then in table entry we have to put both in that cell. In that situation parser will not be able to decide for which reduction it has to parse, so this problem is called reduce-reduce (R-R) problem.
2. If in any state one production is reduced and another is shifted then also parser is not able to parse it. This problem is called shift-reduce (S-R) problem.
3. Due to DFA no shift-shift (S-S) problem occurred.
4. For conflict, there should be two productions and at least one of them should be reduced.
5. Goto section does not participate in conflict.

Problem 6.15: Check if the following grammar is LR(0) or not.

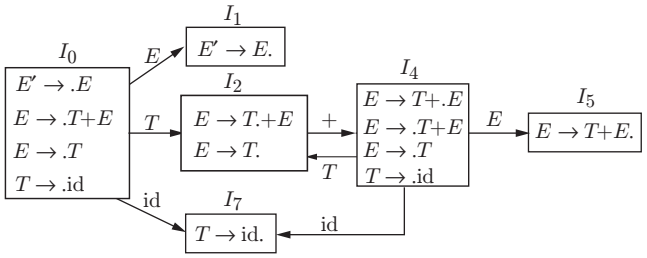
$E \rightarrow T + E$ .....(production number 1)
$E \rightarrow T$ .....(production number 2)
$T \rightarrow$ id .....(production number 3)

Solution:

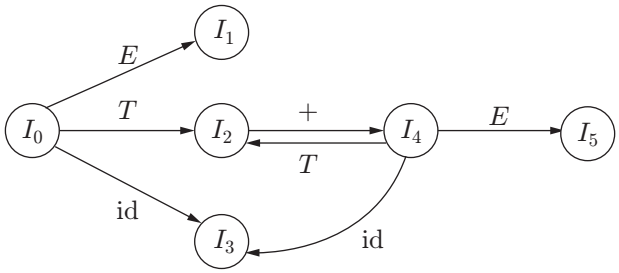
Step 1: Construct an augmented grammar.

$E' \rightarrow E$
$E \rightarrow T + E$
$E \rightarrow T$
$T \rightarrow$ id

Step 2: Find closure.



Step 3: Construct DFA.



Step 4: Construct LR() parsing table.

	Action			Goto	
	+	Id	$	E	T
I_0		S_3		1	2
I_1			Accepted		
I_2	S_4/r_2	r_2	r_2		
I_3	r_3	r_3	r_3		
I_4	S_3			5	2
I_5	r_1	r_1	r_1		

The given grammar is not LR(0) because there are multiple entries (S_4/r_2) in the same cell. This conflict is called as **SR (shift-reduce) conflict**.

SLR(1) Parser

SLR(1) parser is one of the variants of LR parser. An SLR parser is efficient at finding the single correct bottom-up parser in a single scan without backtracking. SLR(1) parser table has only one difference from LR(0) parsing table, reduction entries are made only in the specific location. Let production $E \rightarrow T$ be reduced, then entry of the reduced production will be in those column which comes in Follow(T). If there are multiple entries in the same cell, then the given grammar will not be SLR(1).

From Problem 6.15, we will construct parse table only for SLR(1) (Table 6.1). State 2 has shift and reduce entry, but we have to find that where we have to put the reduce entry. The production which is reduced is $E \rightarrow T$, so we will put r_2 at places which come in Follow(T).

$$\text{Follow}\ (T) = \{+, \$\}$$

Table 6.1 Parsing table for SLR(1)

	Action			Goto	
	+	Id	$	E	T
I_0		S_3		1	2
I_1			Accepted		
I_2	S_4/r_2		r_2		
I_3	r_3	r_3	r_3		
I_4	S_3			5	2
I_5	r_1	r_1	r_1		

There are two entries in the same cell, so this is not SLR(1) grammar.

Canonical LR Parser

Canonical LR parser is a simplified version of an LR parser and is also known as CLR parser. CLR parser is the most powerful parser of the LR parser family. Closure and Goto function are processed differently than in SLR parser.

1. **Closure function of CLR:**
   Closure(I)
   - If I is $A \rightarrow B{\cdot}CD$, \$ and $C \rightarrow {\cdot}EF$ is in G, then add $C \rightarrow {\cdot}EF$, {First(D), \$}.
   - Also add I to closure(I).
   - Repeat above steps for every newly added items.
2. **Goto function of CLR:**
   Goto(I, X)
   - Add I to Goto(I, X) and also move dot($\cdot$) after X.
   - Apply closure to the result obtained in the previous step.

Problem 6.16: Construct parsing table for the following grammar and also check that the given grammar is CLR(1) or not?

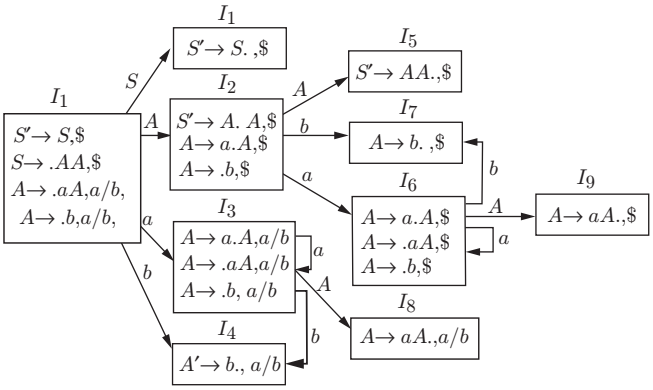
$$S \rightarrow AA$$
$$A \rightarrow aA$$
$$A \rightarrow b$$

Solution:

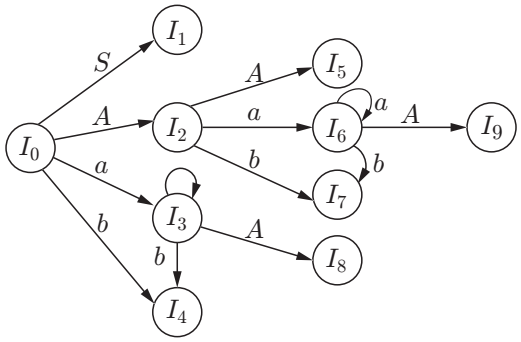
Step 1: Construct an augmented grammar.

$$S' \rightarrow S$$
$$S \rightarrow AA$$
$$A \rightarrow aA$$
$$A \rightarrow b$$

Step 2: Find closure.



Step 3: Construct DFA.



Step 4: Construct parsing table.

	Action			Goto	
State	A	B	$	S	A
I_0	S_3	S_4		1	2
I_1			Accepted		

(*Continued*)

Continued

State	Action A	Action B	Action $	Goto S	Goto A
I_2	S_6	S_7			5
I_3	S_3	S_4			8
I_4	r_3	r_3			
I_5			r_1		
I_6	S_6	S_7			9
I_7			r_3		
I_8	r_2	r_2			
I_9			r_2		

Here, in our example we have three productions which are reduced.

$S \rightarrow AA$ production 1 (represented by r_1)
$A \rightarrow aA$ production 2 (represented by r_2)
$A \rightarrow b$ production 3 (represented by r_3)

To know in which place r_3 will get entry, check what the look-ahead terminals in the closure are of the given production of the grammar. Production $A \rightarrow b$ has $-a$, b as look-ahead terminals. So reduction r_3 will get entry under a and b column. Given grammar is CLR(1) grammar because there are two entries in the same cell.

Note: CLR(1) parser is more powerful than other variants of LR parser, but it is costlier than the other due to more number of states in the DFA of CLR(1), even though two states are exactly the same other than look-ahead symbol. So by merging those two states together, CLR(1) can be minimized and the minimized CLR(1) is called LALR(1).

Operator Precedence Parser: It is based upon bottom-up parsing technique that follow shift-reduce parsing method. Operator precedence parser interprets an operator precedence grammar. Operator precedence parser is capable of parsing all LR(1) grammars.

Operator grammar: A grammar G is said to be operator grammar iff

1. G does not have null production.
2. G does not have two adjacent variables on the right-hand side of the production.

Problem 6.17: Which is/are operator grammar in the given set?

(a) $E \rightarrow E + E|E^*E|\text{id}$
(b) $E \rightarrow AB$, $A \rightarrow a$, $B \rightarrow b$
(c) $E \rightarrow E + E|E^*E|\text{id}|\varepsilon$

Solution:

(a) Operator grammar
(b) Not operator grammar
(c) Not operator grammar

Problem 6.18: Convert the following grammar into operator grammar.

$P \rightarrow SR|S$
$R \rightarrow bSR|bS$
$S \rightarrow WbS|W$
$W \rightarrow L^*W|L$
$L \rightarrow \text{id}$

Solution:

The operator grammars for the given grammars are:

$P \rightarrow SbP|SbS|S$
$R \rightarrow bP|bS$
$S \rightarrow WbS|W$
$W \rightarrow L^*W|L$
$L \rightarrow \text{id}$

Important Points

1. The relation between CLR(1), LALR(1), SLR(1), LR(0) and LL(1) is shown in Fig. 6.11.

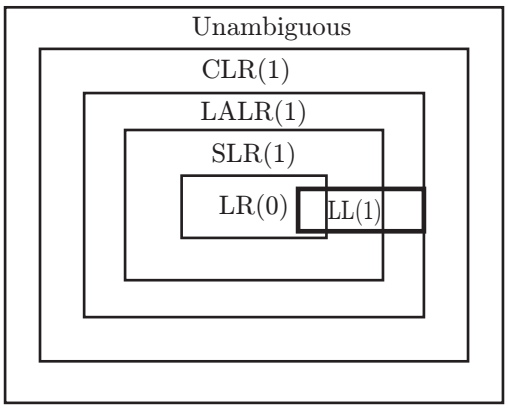


Figure 6.11 Relation between CLR(1), LALR(1), SLR(1), LR(0) and LL(1).

2. The relation between CLR(1), LALR(1), SLR(1), LR(0) and operator precedence parser is shown in Fig. 6.12.

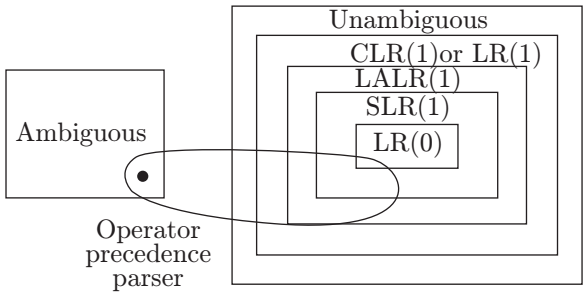


Figure 6.12 Relation between CLR(1), LALR(1), SLR(1), LR(0) and operator precedence parser.

3. If grammar G is LL(1), then
   - It may be LR(0).
   - It may be SLR(1).
   - It will surely be LALR(1).
4. Every LL(1) grammar is surely LALR(1), but if any grammar is LALR(1) then it may or may not be LL(1).
5. If any grammar is LALR(1), then it will surely be CLR(1).
6. CLR(1) is also known as LR(1).
7. Every LL(1) grammar will surely be CLR(1).
8. All LL(k) parsers are subset of LR(k) parser.
9. If the number of states in LR(0), SLR(1), CLR(1) and LALR(1) is n_1, n_2, n_3, n_4, respectively, then the relation between them is

$$n_1 = n_2 = n_4 \leq n_3$$

6.5 SYNTAX-DIRECTED TRANSLATION

Syntax-directed translation (SDT) is a method of compiler implementation which attaches semantic rules with every production of a CFG while translating a string into a sequence of actions. An SDT can be implemented by first constructing a parse tree and then performing the actions in a pre-order traversal.

Example 6.7

Let a production be $E \rightarrow E_1 + E_2$, then SDT will be

$$E \rightarrow E_1 + E_2 \left\{ \begin{array}{c} E \cdot \text{val} = E_1 \cdot \text{val} + E_2 \cdot \text{val} \\ \text{Print}\,(E \cdot \text{val}); \end{array} \right\}$$

6.5.1 Attributes of Syntax-Directed Translation

The following are two types of attributes supported by the SDT:

1. **Synthesized attribute:** An attribute is said to be synthesized only if its value is calculated in terms of its children in the parse tree.
2. **Inherited attribute:** An attribute is said to be inherited attribute only if its value is calculated in terms of its parent or children or both in the parse tree.

6.5.2 Types of Syntax-Directed Translation

An SDT is basically divided into two types: S-attribute definition and L-attribute definition (Table 6.2).

Table 6.2 Characteristics of S-attribute and L-attribute definitions

S Attribute Definition	L Attribute Definition
S-attribute definition uses only synthesized attributes.	L-attribute definition uses both synthesized and inherited attributes.
Semantic rules should be placed at the rightmost place of the right-hand side.	Semantic rules can be placed anywhere on the right-hand side of its production.
Evaluated by a bottom-up or post-order traversal of a parse tree.	Evaluated by a bottom-up or pre-order traversal of a parse tree.

6.5.3 Applications of SDT

1. Evaluating arithmetic expression
2. Creating syntax tree
3. Converting infix to postfix
4. Converting infix to prefix
5. Generating intermediate code
6. Converting binary to decimal
7. Storing type information into symbol table

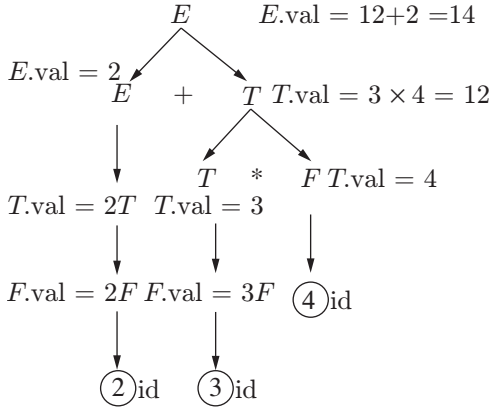
Problem 6.19: Construct SDT to evaluate the given arithmetic expression:

Input: 2 + 3*4
Output: 14

Solution:

+Grammar	Semantic Rules
$E \rightarrow E + T$	E.val = E.val + T.val Print (E.val);
$E \rightarrow T$	E.val = T.val
$T \rightarrow T*F$	E.val = T.val * F.val
$T \rightarrow F$	T.val = F.val
$F \rightarrow$ id	F.val = id

Parse Tree:



Parse tree for given arithmetic expression 2 + 3*4.

Problem 6.20: Consider the following SDT:

$S \rightarrow TR$

$R \rightarrow +T$ {print (+);} R

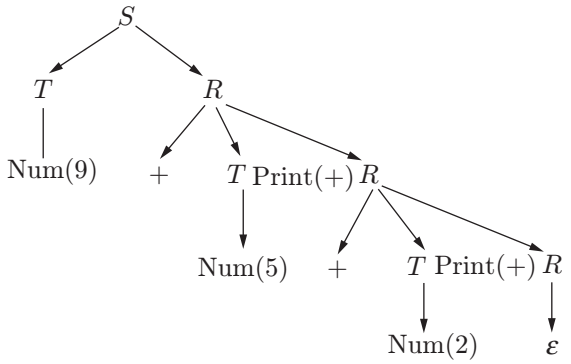
$R \rightarrow \varepsilon$

$T \rightarrow$ Num {print (num);}

For input string $(9 + 5 + 2)$, what will be the output?

Solution:

We first draw the parse tree and then determine the output.



Parse tree for given arithmetic expression $(9 + 5 + 2)$.

Output = 95 + 2+

Problem 6.21: Construct an SDT to convert infix expression to postfix expression.

Input: $a + b^*c$

Output: $abc^* +$

Solution:

The SDT is constructed as follows:

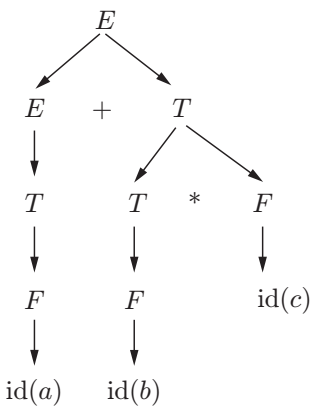
$E \rightarrow E + T$ {print (+)}

$E \rightarrow T$

$T \rightarrow T^*F$ {print (*)}

$T \rightarrow F$

$F \rightarrow$ id {print (id)}



Parse tree for given infix expression $a + b^*c$.

Problem 6.22: Construct an SDT to convert infix expression to prefix expression.

Input: $a + b^*c$

Output: $+ a^*b\ c$

Solution:

The SDT is constructed as follows:

$E \rightarrow$ {print (+)} $E + T$

$E \rightarrow T$

$T \rightarrow$ {print (*)} T^*F

$T \rightarrow F$

$F \rightarrow$ id {print (id)}

6.6 RUNTIME ENVIRONMENT

To generate the target code, there is a need to know about the environment where the source program will execute. It consists mapping of names and objects in the memory, procedure activation, storage allocation, library routines and exception handling, scopes and extents of declarations and symbol table organization.

6.6.1 Storage Organization

Runtime memory needs to be subdivided as follows to hold the different components of an executing program (Fig. 6.13):

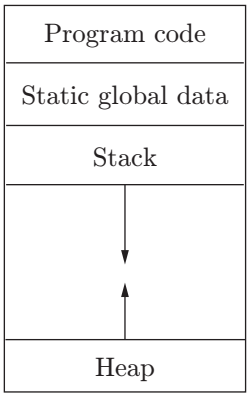


Figure 6.13 | Storage organization.

1. **Program code:** Refers to static area used by the generated target code which is fixed at compile time.
2. **Static global data:** Refers to storage space for data, which does not change during the execution of the program.
3. **Stack:** Manages activation of procedures at runtime. The area usually grows towards lower address.
4. **Heap:** Holds variable created at runtime.

There are two different approaches for runtime storage allocation, as given in Table 6.3.

Table 6.3 | Static and dynamic allocation

Static Allocation	Dynamic Allocation
Allocates all needed space when program starts.	Allocates space when it is needed.
Deallocates all space when program terminates.	Deallocates space when it is no longer needed.

6.6.2 Activation Record and Activation Trees

Activation is the function in execution mode. The function code is the static part whereas execution is the counterpart. The storage associated with an activation of a procedure is called **activation record.**

1. **Activation record content:**
   - *Temporary values*: Values generated as a result of the expression evaluations which cannot be put in registers.
   - *Local data*: Local data to belong to the procedure.
   - *Saved machine*: Keeps the context or machine status (register, PC, etc.).
   - *Access link*: Points to non-local data in other AR.
   - *Control link*: Points to the caller's activation record the return value space of the called function, if any.
   - *Actual parameters*: Used by the calling procedure to pass parameters to called procedures; registers are used to pass these information.

For a recursive procedure or function, several activations may be alive simultaneously. Activation tree shows the path through which control enters and leaves activations for single run of a program. Each node represents an activation of a function, if an arrow is facing a child node from the parent node that means the child function is called by the parent function. Sibling 'a' is left to 'b' means that function 'a' is called before function b.

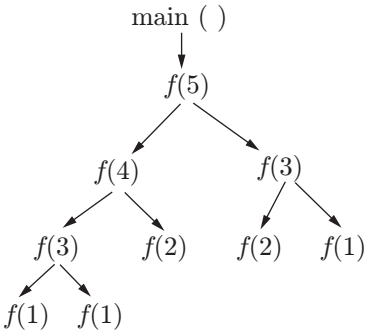
Problem 6.23: Draw an activation tree of the given example.

```
int a[10];
int main(){
int i;
for (i=0; i<10; i++){
a[i] = f(i);
     }
}
int f (int n) {
if (n<3)  return 1;
  return f(n-1)+f(n-2);
          main ends
       }
```

Solution: The activation takes place as follows:

```
System starts main
        enter f(5)
            enter f(4)
                enter f(3)
            enter f(2)
                exit f(2)
                enter f(1)
                exit f(1)
               exit f(3)
               enter f(2)
               exit f(2)
            exit f(4)
            enter f(3)
              enter f(2)
             exit f(2)
             enter f(1)
             exit f(1)
            exit f(3)
        exit f(5)
```

The activation tree for $f(5)$ is as follows:



Activation tree of $f(5)$.

The following observations are made based upon the activation tree:

1. Order of activation corresponds to the pre-order traversal of the tree.
2. Order of deactivation corresponds to the post-order traversal of the tree.
3. If an activation of p calls q, then p will not terminate before q.

6.6.3 Procedure Call Return Model

Every machine's architecture and every language are slightly different from each other. The basic steps followed by a function call are as follows:

1. Before a function call, the calling routine:
   - Saves any necessary registers
   - Pushes the arguments onto the stack for the target call
   - Sets up the static link (if appropriate)
   - Pushes the return address onto the stack
   - Jumps to the target

2. During a function call, the target routine:
   - Saves any necessary registers
   - Sets up the new frame pointer
   - Makes space for any local variables
   - Does its work
   - Tears down frame pointer and static link
   - Restores any saved registers
   - Jumps to saved return address
3. After a function call, the calling routine:
   - Removes return address and parameters from the stack
   - Restores any saved registers
   - Continues executing

6.6.4 Lexical Versus Dynamic Scoping

Lexical (or Static Scoping)	Dynamic Scoping
Binding of variable to declarations is done at compile time.	Binding of variable to declarations is done at compile time.
Lexical scope rules specify the association of variables with declaration based on the textual order of the source code.	Dynamic scoping can be achieved by copying a function verbatim at the place of call.
Innermost enclosing block rule.	Most recent occurrence rule.
Name resolution is independent of caller.	Name resolution depends on caller.
Used by most modern languages: C/C++, Pascal, etc.	Formerly used in some interpreted languages (older versions of LISP).

6.6.5 Symbol Table

It is a data structure used by compiler to store all the information of tokens generated by lexical analyzer. After the lexical analysis phase, semantic analyzer performs type checking on the input code. During type checking, semantic analyzer checks whether the use of names (such as type names, variables, functions) is consistent with their definition in the source code. Consequently, if there are any inconsistencies or misuses found during type checking then it is shown as an error. This is the task of a symbol table.

Operations that can be performed on a symbol table are as follows:

1. Insert
2. Delete
3. Search

Symbol table can be implemented by:

1. Ordered list
2. Unordered list
3. Hash table
4. Tree

Possible entries in a symbol table are as follows:

1. Name
2. Data type
3. Size
4. ID
5. Scope information
6. Storage allocation

6.7 INTERMEDIATE CODE GENERATION

The source program can be directly converted into the target language. But there are many benefits of having an intermediate code or machine-independent code such as increased abstraction, clear separation between front end and back end, and easily retarget and code optimization technique can also be applied (Fig. 6.14).

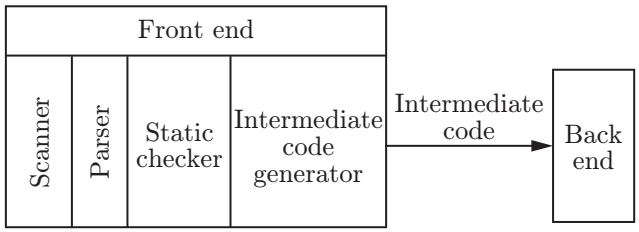


Figure 6.14 Intermediate code generation.

6.7.1 Intermediate Representations

The commonly used representations of intermediate code are as follows:

1. Syntax tree
2. Postfix notation
3. Three-address code

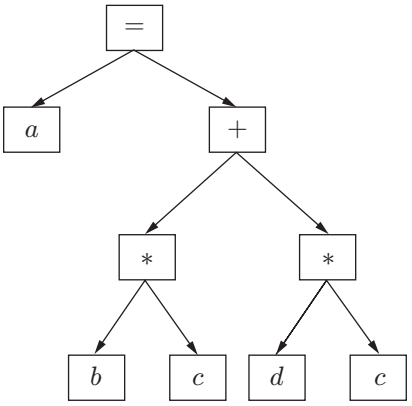
6.7.1.1 Syntax Tree

A syntax tree is the tree representation of the abstract syntactic structure of source code. Due to its abstract nature it is also called as abstract syntax tree. Each of the tree node denotes a construct occurring in source code.

Example 6.8

Statement written in source language:

$a = b^*c + d^*c$



Syntax tree of $a = b^*c + d^*c$.

6.7.1.2 Postfix Notation

Postfix notation is also known as 'reverse polish' notation. Any expression can be written unambiguously. Interpreters can be built easily for postfix notation by using the stack data structure. In the postfix notation, an operator follows the operand.

Example 6.9

The source language statement:

$$a = b^*c + d^*c$$

Can be rewritten in postfix notation as:

$$abc^*dc^* + =$$

6.7.1.3 Three-Address Statement

Three-address statement is a linearized representation of an abstract syntax, in which names of the temporaries correspond to the nodes. The intermediate values name allows three-address code to be easily rearranged, which is convenient for optimization technique. The reason to call 'three-address code' is that each statement generally contains three addresses, two for the operands and one for the result. Representations of the three-address statement are quadruples, triples and indirect triples.

1. **Quadruples:** A quadruple is a record structure having four fields, namely op, arg1, arg2 and result. The op field contains an internal code for operator. To translate binary expression 'x operator y' into three-address code, operator is placed in 'op' column; x in 'arg1' column, y in 'arg2' column and a new temporary variable stores their calculated result in the result column.

Example 6.10

The quadruples for the assignment $a = b^*c + d^*e$ generates the following three-address code:

	Op	arg1	arg2	Result
(0)	=	C		t_1
(1)	*	B	t_1	t_2
(2)	=	D		t_3
(3)	*	E	t_3	t_4
(4)	+	t_2	t_4	t_5
(5)	=	t_5		A

2. **Triples:** Temporary variables or memory names in the symbol table can be avoided by the position number of the statement that computes it. If we used this method then three address statements can be represented by records with three columns operation, arg1 and arg2. The column arg1 and arg2, for the arguments of operation, are either pointers to the symbol table or pointers into the triple structure. As numbers of used fields are three, so this format is known as triples.

Example 6.11

	Operation	Arg1	Arg2
(0)	=	c	
(1)	*	b	(0)
(2)	=	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	=	a	(4)

3. **Indirect triples:** Triples are very difficult to optimize because for optimization it required moving of intermediate code and other triples connected to it also have to be updated. So, a new variant of

triples is called indirect triples is being used which is easier to optimized. Indirect triples perform listing pointers to triples, rather than listing the triples themselves.

Example 6.12

#	Stmt		#	Op	Arg1	Arg2
(0)	(14)	→	(14)		*C*	
(1)	(15)	→	(15)	*	*B*	(14)
(2)	(16)	→	(16)		*C*	
(3)	(17)	→	(17)	*	*B*	(16)
(4)	(18)	→	(18)	+	(15)	(17)
(5)	(19)	→	(19)	=	*A*	(18)

Program Triple container

6.8 CODE OPTIMIZATION

Code optimization phase is to reduce the size of the code and improve the performance of the code generated by intermediate code phase. The most important part of optimized code is to minimize the amount of time taken by the code to execute and less common is to minimize the amount of memory used by the code. The basic requirement optimization methods should comply with is that an optimized program must have the same output and side effects as its non-optimized version.

6.8.1 Types and Levels of Optimization

Optimization can be performed by automatic optimizers or programmers. An optimizer is a built-in unit of a compiler. Modern processors have the capability to optimize the execution order of code instructions. Optimizations are classified mainly into two types, one is high-level and the other is low-level optimizations. High-level optimizations are generally performed by the programmer who handles source code of the programs such as classes, functions, control statements, and procedures. Low-level optimizations are performed at the stage when source code is compiled into a set of machine instructions, and it is at this stage that automated optimization is usually employed.

6.8.2 Primary Source of Optimization

Optimization can be done by using different ways. Some of the primary source of optimization is discussed below:

6.8.2.1 Dead-Code Elimination

Dead-code is a section of program code which is executed but the result produced by that section is never used. So, it can be removed from the program code because it does not have any effect on the functionality of program.

Let we have temp1 = temp2 – temp3, and temp1 is never used further in the program then we can eliminate this whole instruction.

6.8.2.2 Constant Propagation

Constant propagation is the process of substituting a constant value by the subsequent uses of a variable while there is no intervening changes in the value of that variable. Consider the given example below:

```
int a = 10;
int b = 12 - a / 2;
return b = b * (35/a +2);
```

Variable 'a' propagating the constant value 10 in the given code.

6.8.2.3 Constant Folding

Constant folding is the process of evaluating the constant expression at compile time. Constant folding process improves run-time performance and also reduces code size by evaluating constant at compile time. In program code below, the expression (4–2) can be evaluated at compile time and replaced with the constant 2.

```
int sub( )
{
Return (4-2);
}
```

6.8.2.4 Elimination of Common Sub-Expression

If two or more operations are similar and produce same results then it is efficient way that to compute at once and refer that results further rather than re-evaluate it. In the given program code below, temp 3 and temp 6 produce similar results and temp 4 and temp 5 also

produce similar results, so we can compute them once and refer results further below in the code.

```
main()
{
int x, y, z;

x = (1+20)* -x;
y = x*x+(x/y);
y = z = (x/y)/(x*x);
}

straight translation:

tmp1 = 1 + 20;
tmp2 = -x;
x = tmp1 * tmp2;
tmp3 = x * x;
tmp4 = x/y;
y = tmp3 + tmp4;
tmp5 = x/y;
tmp6 = x * x;
z = tmp5/tmp6;
y = z;
```

What sub-expressions can be eliminated? How can valid common sub-expressions (live ones) be determined? Here is an optimized version, after constant folding and propagation and elimination of common sub-expressions:

```
tmp2 = -x;
x = 21 * tmp2;
tmp3 = x * x;
tmp4 = x/y;
y = tmp3 + tmp4;
tmp5 = x/y;
z = tmp5/tmp3;
y = z;
```

6.8.2.5 Copy Propagation

Copy propagation is the different way of optimization, in which assignment a=b for some variable 'a' and 'b', we can replace later uses of 'a' with use of 'b' (it is to be assumed that there is no change to either variable in-between). The code on the left side makes a copy of tmp1 in tmp2 and a copy of tmp3 in tmp4. When we do optimization then on the right, we eliminated those unnecessary copies and propagated the original variable into later uses.

```
tmp2 = tmp1;
tmp3 = tmp2 * tmp1;
tmp4 = tmp3;
tmp5 = tmp3 * tmp2;
c = tmp5 + tmp4;
tmp3 = tmp1 * tmp1;
tmp5 = tmp3 * tmp1;
c = tmp5 + tmp3;
```

IMPORTANT FORMULAS

1. Time complexity of RDP is $O(2^n)$.
2. Every regular grammar need not be LL(1), because that grammar may contain left factoring.
3. Any ambiguous grammar cannot be LL(1).
4. If any grammar contains left factoring, then it can't be LL(1) grammar.
5. If given grammar contains left recursion, then it can't be LL(1) grammar.
6. If grammar G is LL(1), then
   - It may be LR(0).
   - It may be SLR(1).
   - It will surely be LALR(1).
7. Every LL(1) grammar will surely be LALR(1), but if any grammar is LALR(1) then it may or may not be LL(1).
8. If any grammar is LALR(1), then it will surely be CLR(1).
9. CLR(1) is also known as LR(1).
10. Every LL(1) grammar will surely be CLR(1).
11. All LL(k) parsers are subset of LR(k) parser.
12. If the number of states in LR(0), SLR(1), CLR(1) and LALR(1) is n_1, n_2, n_3, n_4, respectively, then relation between them is

$$n_1 = n_2 = n_4 \leq n_3$$

SOLVED EXAMPLES

1. Which of the following derivations does a top-down parser use while parsing an input string? The input is assumed to be scanned in left to right order.

(a) Leftmost derivation
(b) Topmost derivation
(c) Rightmost derivation
(d) Leftmost derivation traced out in reverse

Solution: Top–Down parser uses leftmost derivation for constructing parse tree.

Ans. (a)

2. The process of assigning load addresses to various parts of the program and adjusting the code and data in the program to reflect the assigned addresses is called

(a) address assembly (b) parsing
(c) relocation (d) indexing

Solution: The process adjusting the code and data in the program to reflect the assigned addresses is called relocation.

Ans. (c)

3. Which of the following statements is true?

(a) An unambiguous grammar need not have same leftmost and rightmost derivation.
(b) An LL(1) parser is not a top-down parser.
(c) LALR is less powerful than SLR.
(d) An ambiguous grammar can be LR(k) for any k.

Solution: Only option (a) is true. LL(1) parser is a top–down parser; LALR is more powerful.

Ans. (a)

4. Given the following expression grammar:

$E \rightarrow E*F|F + E|F$

$F \rightarrow F - F|\text{id}$

Which of the following is true?

(a) * has higher precedence than +
(b) − has higher precedence than *
(c) + and − have same precedence
(d) + has higher precedence than *

Solution: id has higher precedence than any other symbol.

Ans. (b)

5. The number of tokens in the following C statement is

```
printf("i=%d, &i =%x",i&i);
```

(a) 13 (b) 6
(c) 10 (d) 11

Solution: The tokens in C tokens include identifiers, keywords, constants, operators, string literals and other separators. Thus, there are 10 tokens in the given printf statement.

Ans. (c)

6. The identifications of common sub-expression and replacement of run-time computations by compile-time computations is

(a) local optimization
(b) global optimization
(c) constant folding
(d) common fielding

Solution: Constant folding is the process of evaluating the constant expression at compile time.

Ans. (c)

7. In a compiler the module that checks every character of the source text is called

(a) the machine dependant optimizer
(b) the code checker
(c) the lexical analyzer
(d) the syntax analyzer

Solution: Lexical analyzer reads the source program character by character at a time and unites them into a stream of tokens.

Ans. (c)

8. Match the following.

Group 1	**Group 2**
A. Lexical analysis	P. Directed acyclic graphs
B. Code optimization	Q. Syntax trees
C. Code generation	R. PDA
D. Abelian groups	S. Finite automaton

(a) A-S, B-P, C-R, D-Q
(b) A-Q, B-R, C-S, D-P
(c) A-S, B-Q, C-R, D-P
(d) A-P, B-S, C-R, D-Q

Solution: A-S, B-P, C-R, D-Q

Ans. (a)

9. Which of the following strings can definitely be said to be token without looking at the next input character while compiling a Pascal program?

1. Begin
2. Goto
3. <>

(a) 1 (b) 2
(c) 3 (d) All of the above