RGPVNOTES.IN

Program : **B.Tech**

Subject Name: **Compiler Design**

Subject Code: **IT-603**

Semester: **6th**

Department of Information Technology
Subject Notes
IT603 (A) – Compiler Design
B.Tech, IT-6th Semester

Unit II

Syllabus: Syntax Analysis: working of Parser, Top down parsing, Bottom-up parsing, Operator precedence parsing, predictive parsers, LR parsers (SLR, Canonical LR, LALR), constructing SLR parsing tables, constructing Canonical LR parsing tables, Constructing LALR parsing tables, using ambiguous grammars, an automatic parser generator.

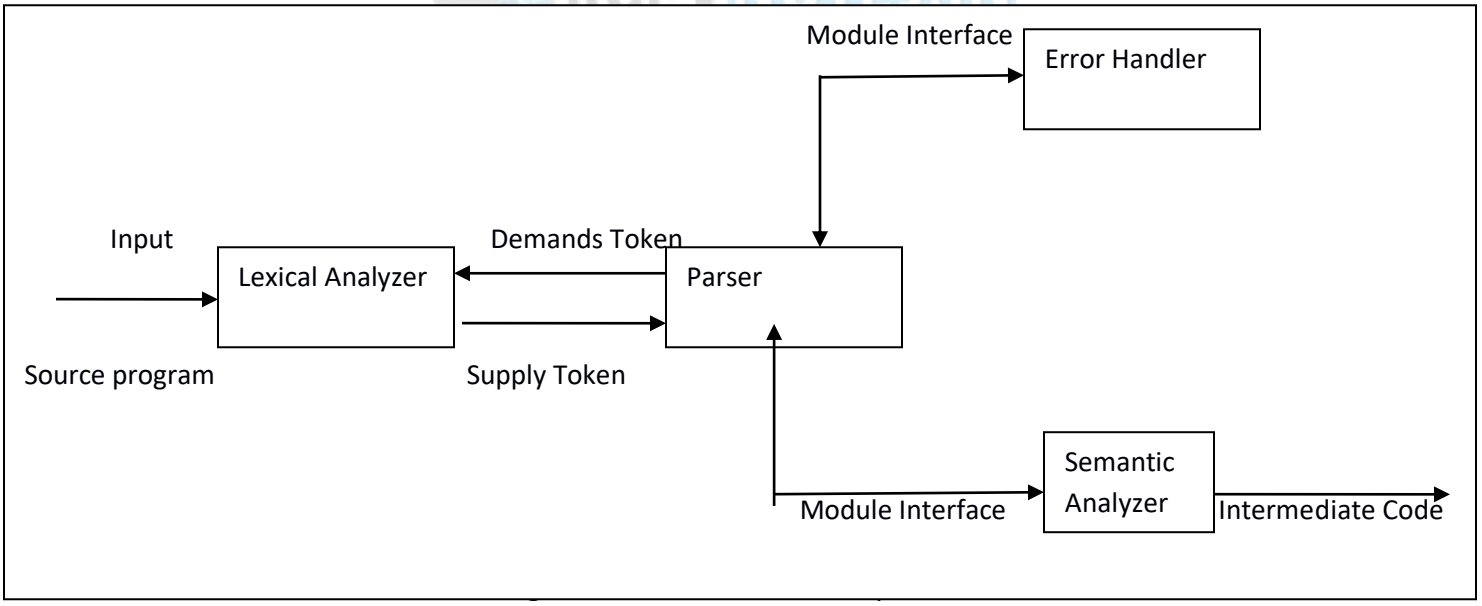
Unit Outcome: Specify and analyze the lexical, syntactic and semantic structures of advanced language features.

UNIT- 2: Syntax Analysis

The syntax analysis is the second phase of compiler which basically checks for the syntax of the language. The syntax analyzer takes the tokens from the lexical analyzer and groups them in such a way that some programming syntax can be recognized.

Parser:

A parsing or syntax analysis is a process which takes the input string w and produces either a parse tree (syntactic structure) or generates the syntactic errors.



Working of Parser:

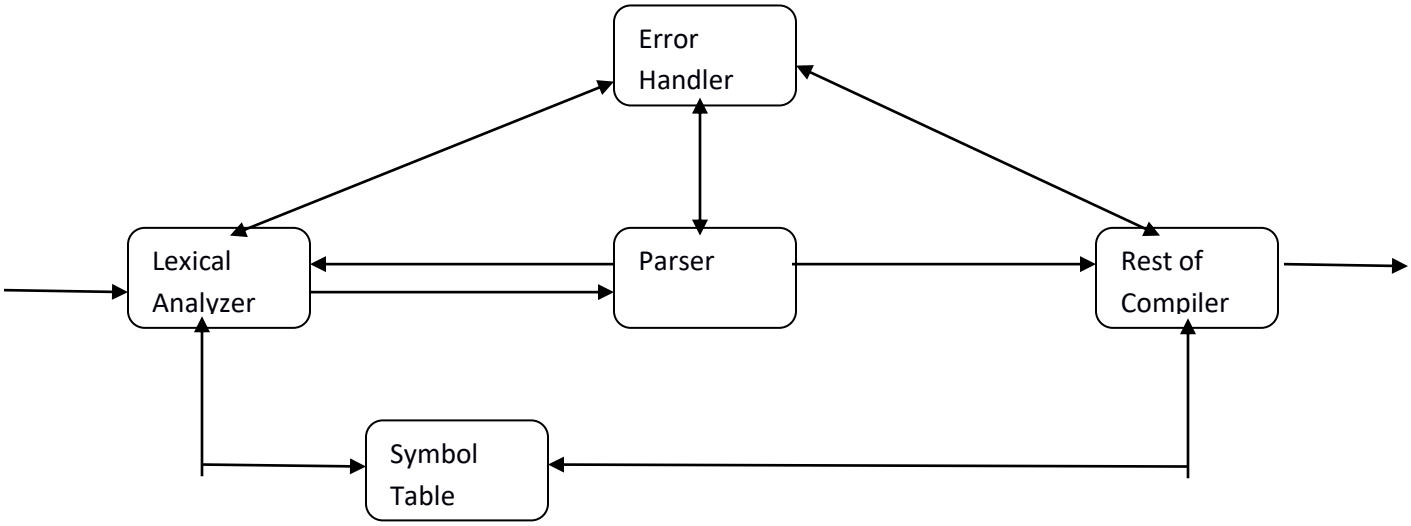


Figure 2.2: Working of Parser

The parser collects sufficient number of tokens and builds a parse tree. By building the parse tree, parser smartly finds the syntactical errors of any.

Parsing Techniques:

Based on the working principle, there are majorly two parsing techniques:

1. The parser scans the input string from left to right and identifies that the derivation is leftmost or rightmost.
2. The parser makes use of production rule for choosing the appropriate derivation.

The different parsing techniques use different approaches in selecting the appropriate rules for derivation.

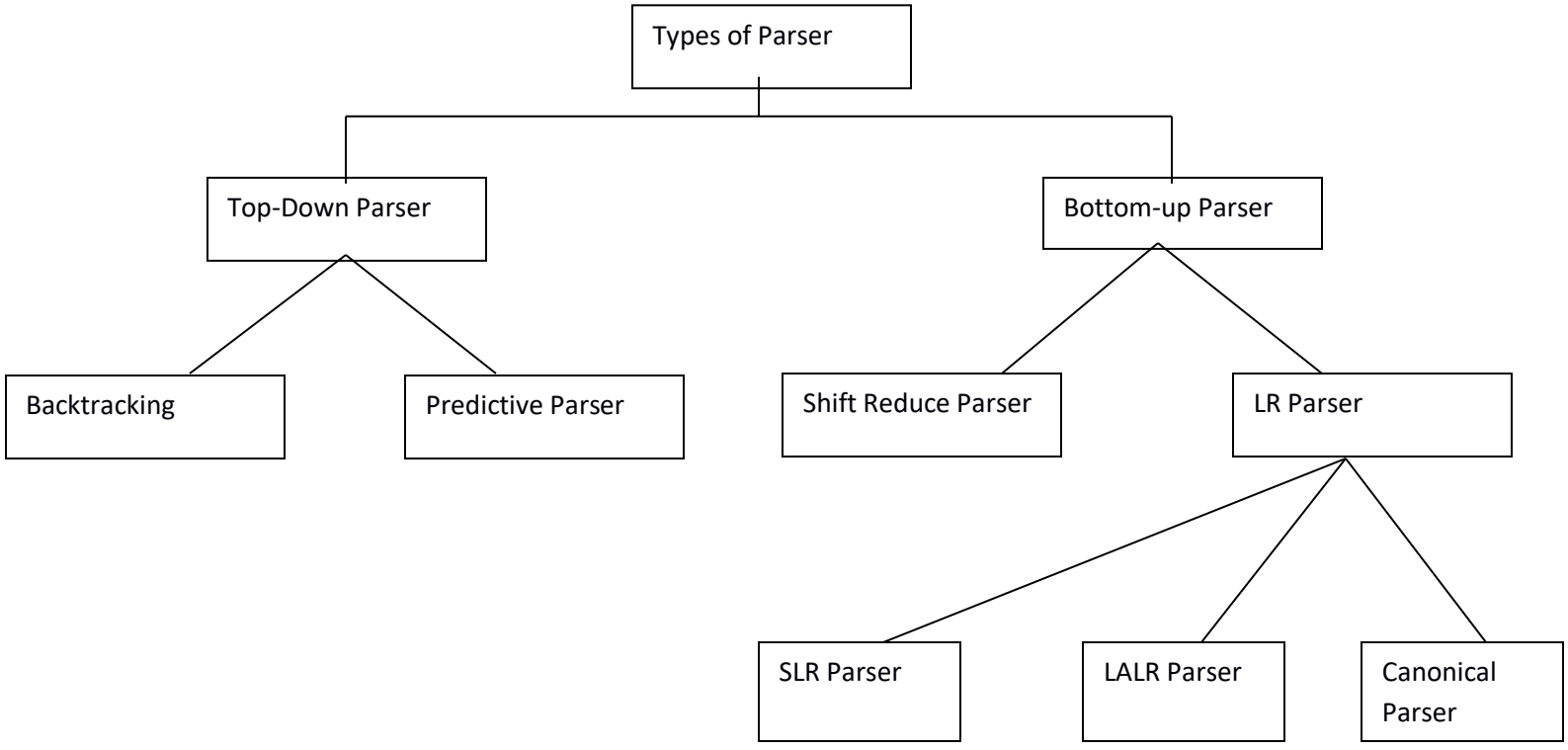


Figure 2.3: Classification of Parsing

Top-Down Parsing:

In top-down parsing the parse tree is generated from top to bottom (from root to leaves). The derivation terminates when the required input string terminates. The prime objective of top-down parsing is to find the appropriate production rule in order to produce the correct input string.

Example:

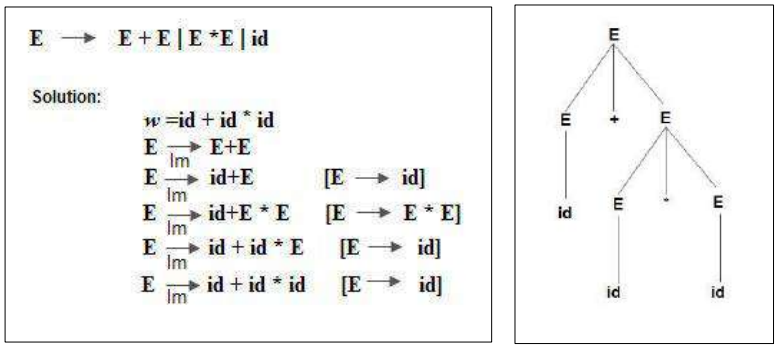


Figure 2.4: Top-Down Parsing

Bottom-up Parsing:

In bottom-up parsing, the input string is taken first and we try to reduce this string with the help of grammar and try to obtain the start symbol. The process of parsing halts successfully as soon as we reach to start symbol.

Example:

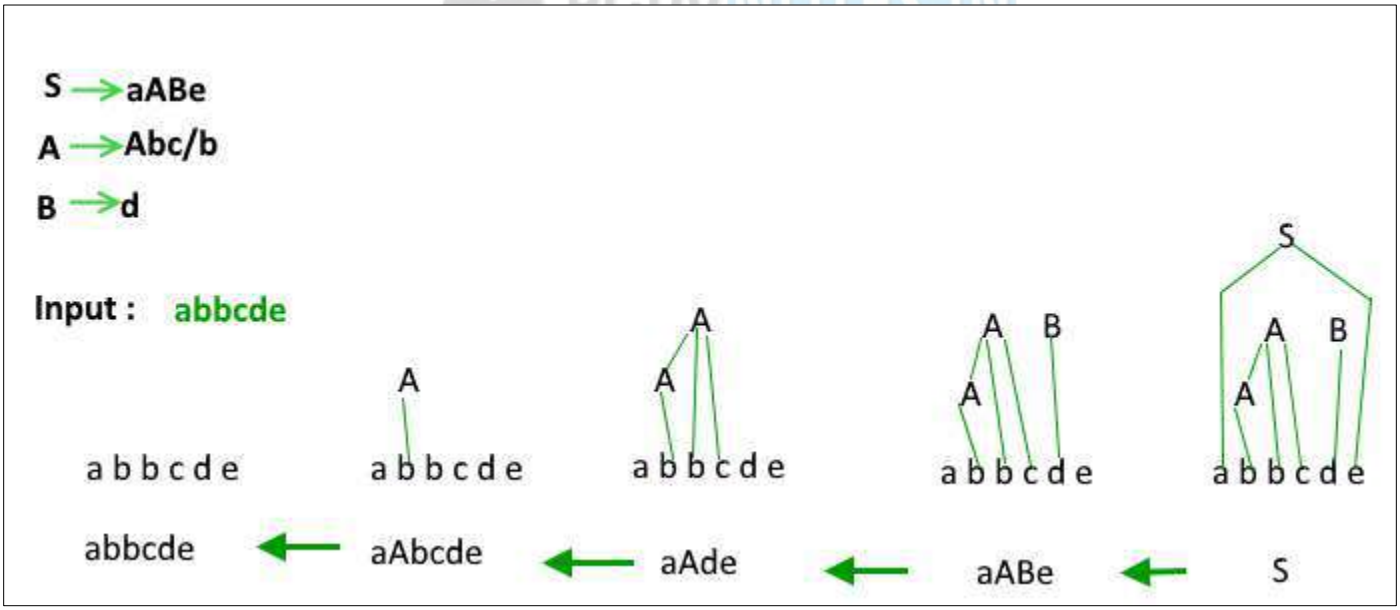


Figure 2.5: Bottom-up Parsing

Predictive Parser:

The predictive parser tries to predict the next construction using one or more lookahead symbols from input string. There are two types of parsers:

1). Recursive Descent

2). LL(1) Parser

Recursive Descent Parser:

A parser that uses collection of recursive procedures for parsing the given input string is called Recursive Descent Parser. The CFG is used to build the recursive routines. The R.H.S. of the production rule is directly converted to a program. For each non-terminal a separate procedure is written and body of the procedure is R.H.S. of the corresponding non-terminal.

LL(1) Parser:

This is of non-recursive type. In this type of parsing a table is built. For LL(1) the first L means the input is scanned from left to right. The second L means it uses leftmost derivation for input string; and the number 1 in the input symbol means it uses only one input symbol (lookahead) to predict the parsing process.

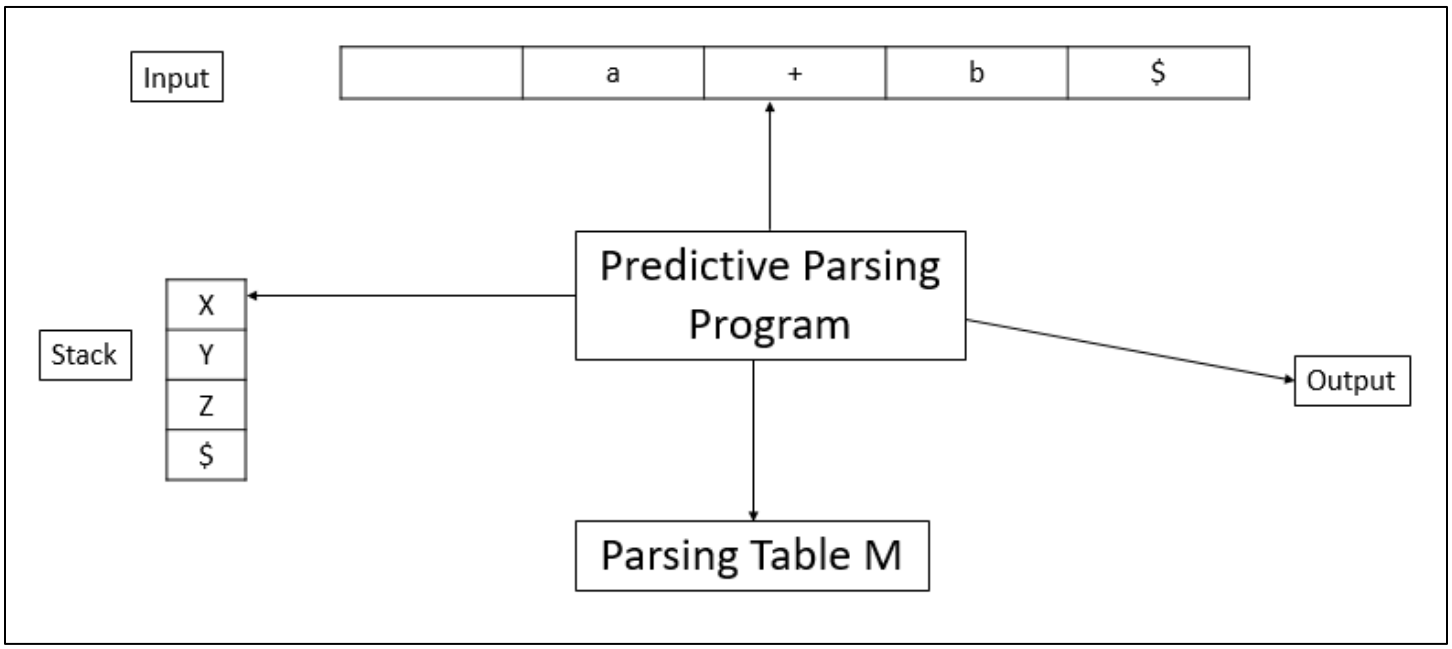


Figure 2.6: Model of LL(1) Parser

The data structure used by LL(1) are input buffer, stack and parsing table. The input buffer is used to store the input tokens. The stack is used to hold the left sentential form. The symbols used in R.H.S. of rule are pushed into the stack in reverse order i.e. from right to left. The table is basically two dimensional array. The table has row for non-terminal and column for terminals.

Operator Precedence Parser:

A grammar G is said to be operator precedence if it process following properties:

1. No production on the right side is €
2. There should not be any production rule possessing two adjacent non-terminals at the right hand side.

In operator precedence parsing we will first define precedence relations <.= and .> between pair of terminals. The meaning of these relations are

p <. q p gives more precedence than q

p = q p has same precedence as q

p. > q p takes precedence over q

-	Id	+	*	$
Id		.>	.>	.>
+	<.	.>	.>	.>
*	<.	.>	.>	.>
$	<.	<.	<.	

Table 2.1: Precedence Relation Table

LR Parser:

This is the most efficient method of bottom-up parsing which can be used to parse the large class of context free grammar. This method is also called LR(K) parsing. Here

L stands for Left to Right scanning

R stands for Rightmost Derivation in reverse.

K is number of input symbols. When k is omitted k is assumed to be 1.

Properties of LR parser

1. LR parsers can be constructed to recognize most of the programming language for which context free grammar can be written.
2. The class of grammar that can be parsed by LR parser is a super of class of grammars that can be parsed using predictive parsers.
3. LR parser works using non backtracking shift reduce technique yet it is efficient one.

LR parser detect syntactical errors very efficiently.

Structure of LR Parser

LR parser structure consist of input buffer for storing the input string, a stack for storing the grammar symbols, output and a parsing table comprised of two parts namely action and goto. There is one parsing program which is actually a driving program and reads the input symbol one at a time from the input buffer.

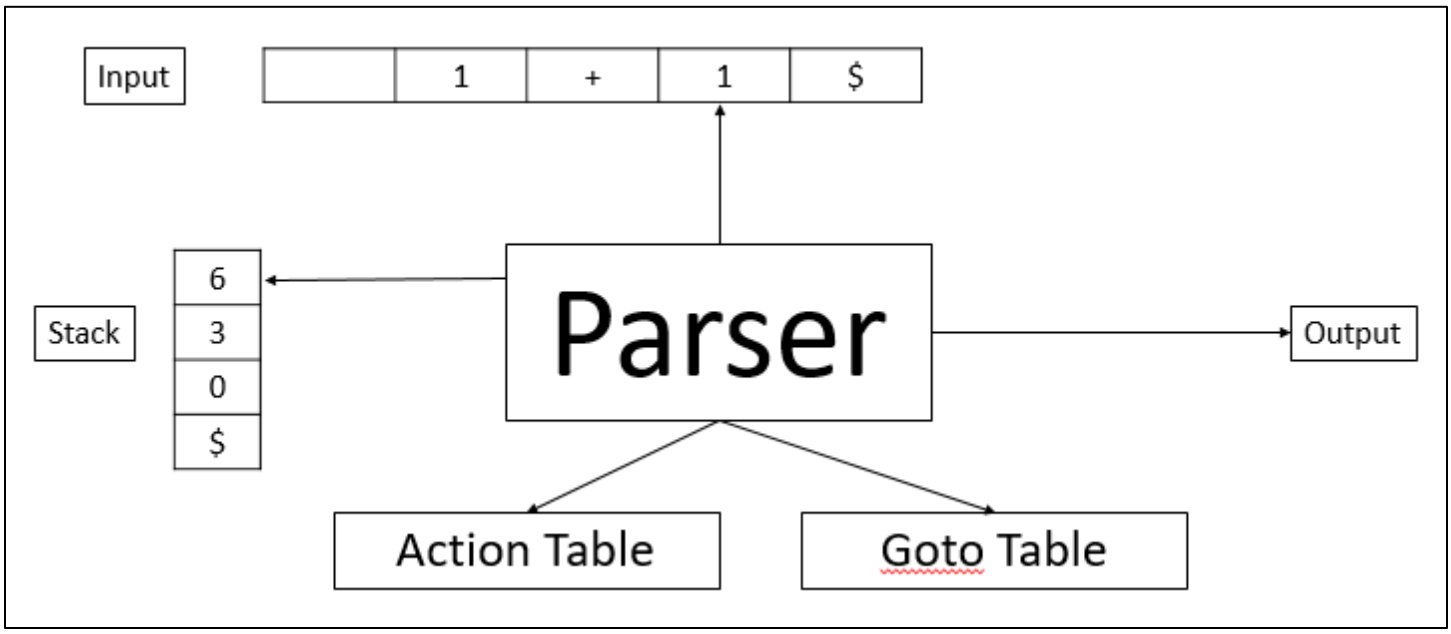


Figure 2.7: Structure of LR Parser

Example:

Consider the following grammar

E → E+T | T; T → TF | F; F → F* | a | b

Solution:

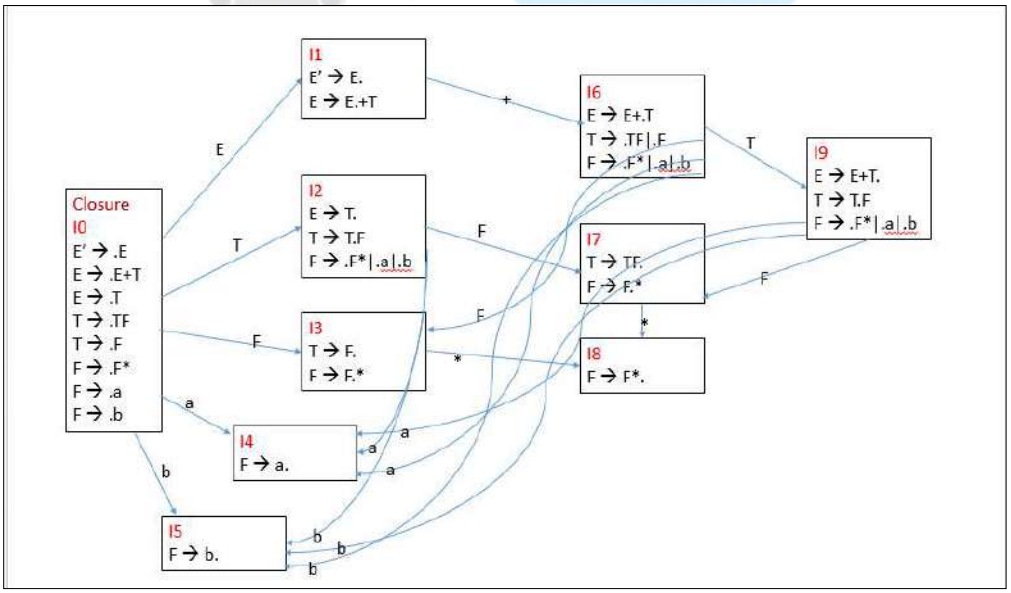


Figure 2.8: LR Parsing Example

	Action Part					Goto Part		
	a	b	+	*	$	E	T	F
0	S4	S5				1	2	3
1			S6		Accept			
2	R2/S4	R2/S5	R2	R2	R2			7

3	R4	R4	R4	R4/S8	R4			
4	R6	R6	R6	R6	R6			
5	R7	R7	R7	R7	R7			
6	S4	S5					9	3
7	R3	R3	R3	R3/S8	R3			
8	R5	R5	R5	R5	R5			
9	R1/S4	R1/S5	R1	R1	R1			7

Table 2.2: LR Parsing table

Simple LR Parsing (SLR):

It is the weakest of the three methods but it is easiest to implement. The parsing can be done as follows:

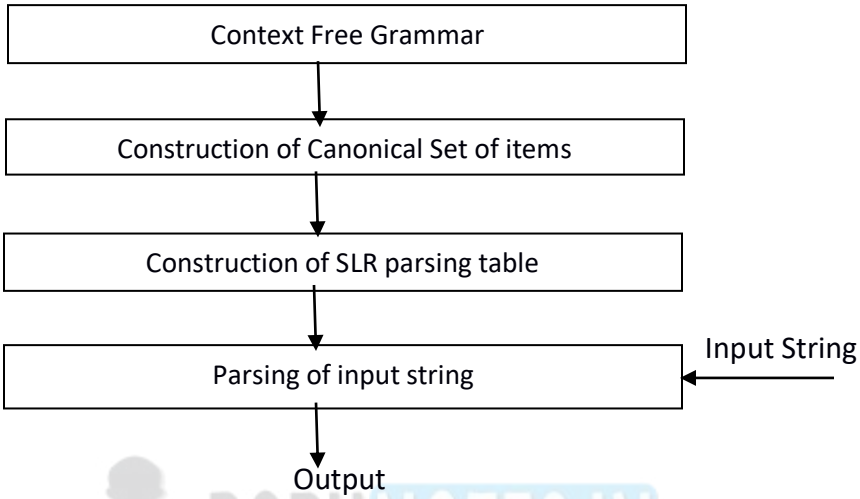


Figure 2.9: SLR

Construction of SLR parsing table:

According to the structure of SLR parser there are two parts of SLR parsing table- action and goto. By considering basic parsing actions such as shift, reduce, accept and error we will fill up the action table. The goto table can be filled up using goto function.

Input: An Augmented Grammar G'

Output: SLR parsing table

Algorithm:

1. Initially construct set of items C = {I_0, I_1, I_2, … I_n} where C is a collection of set of LR(0) items for the input grammar G'
2. The parsing actions are based on each item I_i. The actions are as given below.
   a. If A->α.aβ is in I_i and goto(I_i, a) = I_j then set action[I,a] as "shift j". Note that a must be a terminal symbol.
   b. If there is a rule A-> α. Is in I_i then set action[I,a] to "reduce A->α" for all symbols a, where a € FOLLOW (A). Note that A must not be an augmented grammar S'.
   c. If S' -> S is in I_i then the entry in the action table action[I,$] = "accept".
3. The goto part of the SLR table can be filled as : The goto transitions for state I is considered for non-terminal only. If goto(I_i, A) = I_j then goto[I_i, A]=j
4. All the entries not defined by rule 2 and 3 are considered to be "error".

Example:

Consider the grammar:

S → AaAb | BbBa; A → €; B → €

Solution:

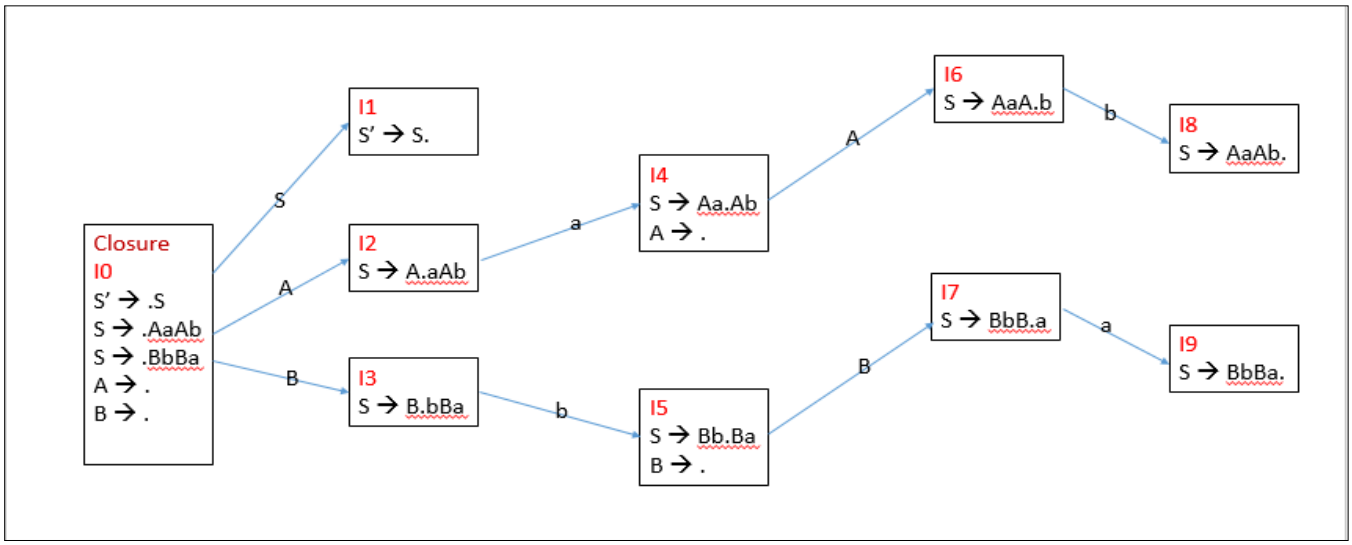


Figure 2.10: SLR Example

	Action Part			Goto Part		
	A	b	$	S	A	B
0	R3/R4	R3/R4		1	2	3
1			Accept			
2	S4					
3		S5				
4	R3	R3			6	
5	R4	R4				7
6		S8				
7	S9					
8			R1			
9			R2			

Table 2.3: SLR Parsing Table

LR(K) Parser: Canonical LR Parser:

The canonical set of items is the parsing technique in which a lookahead symbol is generated while constructing set of items. Hence the collection of set of items is referred as LR(1). The value 1 in the bracket indicates that there is one lookahead symbol in the set of items.

Steps:

1. Construction of canonical set of items along with the lookahead.
2. Building canonical LR parsing table.
3. Parsing the input string using canonical LR parsing table.

Construction of Canonical LR parsing table:

Input: An Augmented Grammar G'

Output: The canonical LR parsing table

Algorithm:

1. Initially construct set of items C = {I_0, I_1, I_2, … I_n} where C is a collection of set of LR(1) items for the input grammar G'
2. The parsing actions are based on each item I_i. The actions are as given below.
   a. If [A->α.aβ, b] is in I_i and goto(I_i, a) = I_j then create an entry in the action table action[I_i,a] = shift j..
   b. If there is a production [A-> α., a] in I_i then in the action table action[I_i,a] = reduce by A->α. Here A should not be S'.
   c. If there is a production S' -> S., $ in I_i then action[i,$] = accept.
3. The goto part of the LR table can be filled as : The goto transitions for state i is considered for non-terminal only. If goto(I_i, A) = I_j then goto[I_i, A]=j
4. All the entries not defined by rule 2 and 3 are considered to be "error".

Example:

Consider the grammar:

S → AA; A → aA; A → b

Solution:

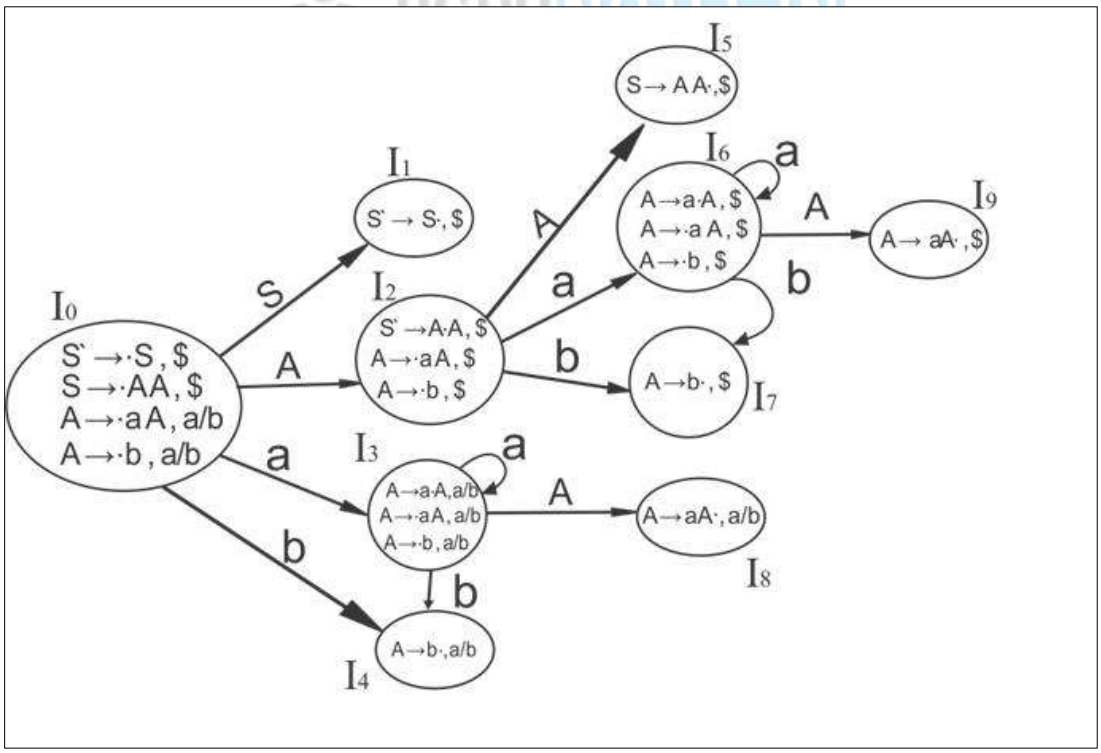


Figure 2.11: Canonical Parsing Example

States	a	b	$	S	A
I_0	S_3	S_4			2
I_1			Accept		
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		

Table 2.4: Canonical Parsing Table

LALR Parser:

In this type of parser the lookahead symbol is generated for each set if item. The table obtained by this method is smaller in size than LR(k) parser. In fact the states of SLR and LALR parsing are always same. Most programming languages uses LALR parsers.

Steps:

1. Construction of canonical set of items along with the lookahead.
2. Building LALR parsing table.
3. Parsing the input string using canonical LR parsing table.

Construction of LALR parsing table:

Step 1: Construct the LR(1) Set of items.

Step 2: Merge the two states I_i and I_j if the first component (i.e. the production rules with dots) are matching and create a new state replacing one of the older state such as $I_{ij} = I_i \cup I_j$

Step 3: The parsing actions are based on each item Ii. The action are as given below

a. If [A->α.aβ, b] is in I_i and goto(I_i, a) = I_j then create an entry in the action table action[I,a] = shift j.
b. If there is a production [A->α., a] in I_i then in the action table action [I_i, a] = reduce by A->α. Here A should not be S'
c. If there is a production S' -> S., $ in I_i then action[i,$]=accept.

Step 4: The goto part of the LR table can be filled as : The goto transitions for state i is considered for non-terminals only. If goto(I_i, A) = I_j then goto[I_i, A] = j.

Step 5: If the parsing action conflict then the algorithm fails to produce LALR parser and grammar is not LALR(1). All the entries not defined by rule 3 and 4 are considered to be "error".

Example:

Consider the grammar:

S → CC; C → aC | d

Solution:

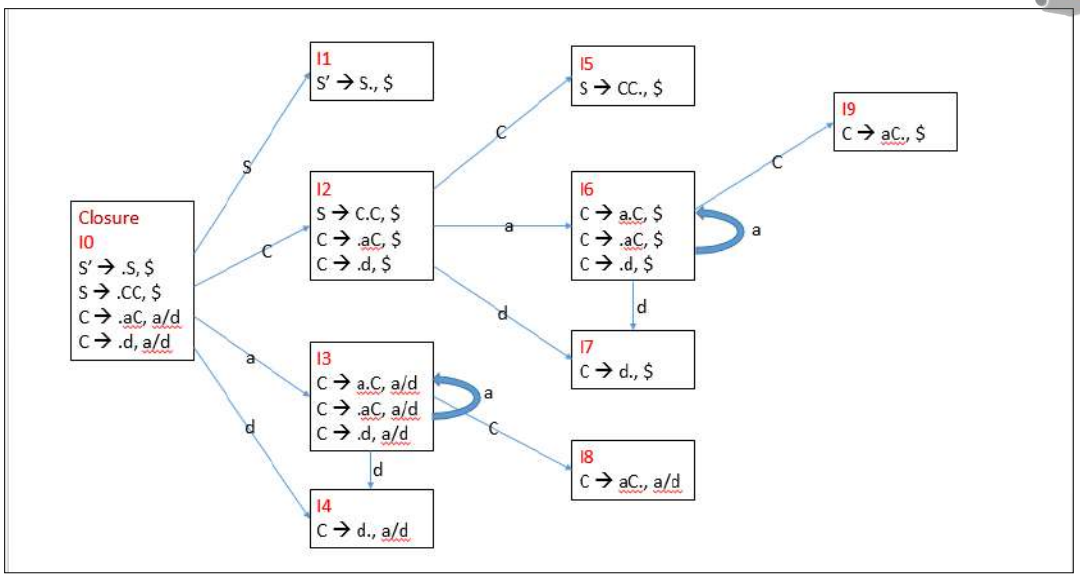


Figure 2.12: LALR Example

	Action Part			Goto Part	
	A	b	$	S	C
0	S36	S47		1	2
1			Accept		
2	S36	S47			5
36	S36	S47			89
47	R3	R3			
5			R1		
89	R2	R2	R2		

Table 2.5: LALR Parsing Table

Using Ambiguous Grammar:

There are various parsing methods in which if at all the grammar is ambiguous then it creates the conflicts and we cannot parse the input string with such ambiguous grammar. But for some languages in which arithmetic expressions are given ambiguous grammar are most compact and provide more natural specification as compared to equivalent unambiguous grammar. Secondly using ambiguous grammar we can add any new productions for special constructs.

While using ambiguous grammar for parsing the input string we will use all the disambiguating rules so that each time only one parse tree will be generated for that specific input. Thus ambiguous grammar can be used in controlled manner for parsing the input.

Automatic Parser Generator:

A parser generator is a program that takes as input a specification of a syntax, and produces as output a procedure for recognizing that language. Historically, they are also called compiler-compilers.

YACC (yet another compiler-compiler) is an LALR(1) (LookAhead, Left-to-right, Rightmost derivation producer with 1 lookahead token) parser generator. YACC was originally designed for being complemented by Lex.

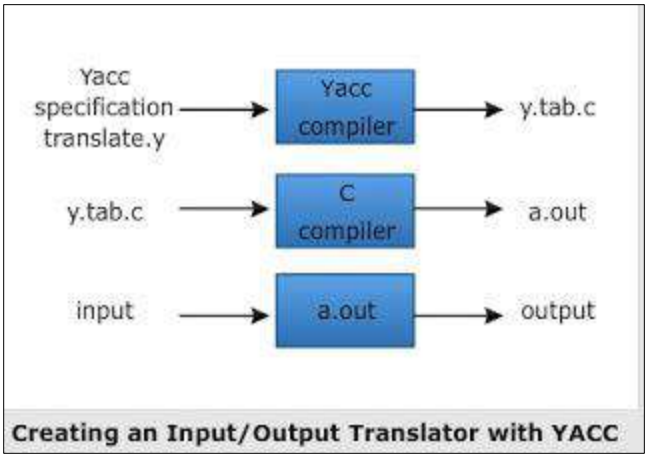


Figure 2.13: YAAC-An Automatic Parser Generator

YACC is a automatic tool that generates the parser program

YACC stands for Yet Another Compiler Compiler. This program is available in UNIX OS

The construction of LR parser requires lot of work for parsing the input string. Hence, the process must involve automation to achieve efficiency in parsing an input

Basically YACC is a LALR parser generator that reports conflicts or uncertainties (if at all present) in the form of error messages.

RGPVNOTES.IN

We hope you find these notes useful.

You can get previous year question papers at https://qp.rgpvnotes.in .

If you have any queries or you want to submit your study notes please write us at rgpvnotes.in@gmail.com

LIKE & FOLLOW US ON FACEBOOK
facebook.com/rgpvnotes.in