Q.8. Explain ambiguity in context-free grammars.

Or

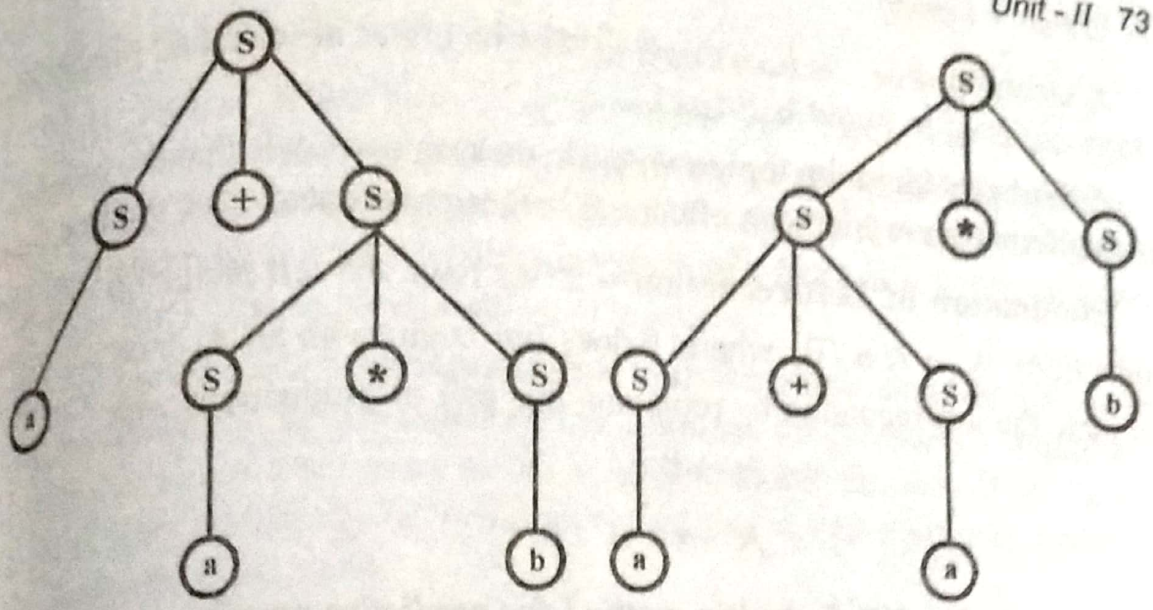
What do you mean by ambiguous and unambiguous grammar ? Explain with example. ***(R.G.P.V., June 2009)***

Or

What do you mean by ambiguity of grammar ? (R.G.P.V., June 2003)

Ans. A grammar that produces more than one parse tree for some sentences is said to be *ambiguous.* Put another way,an *ambiguous* grammar is one that produces more than one leftmost or more than one rightmost derivation for some sentence. For certain types of parsers, it is desirable that the grammar be made *unambiguous,* for if it is not, we cannot uniquely detrmine which parse tree to select for a sntence. For some application we shall also consider methods whereby we can use certain ambiguous grammars, together with *disambiguating* rules that "throw away" undesirable parse trees, leaving us with only one tree for each sentence.

Sometimes there may occur some ambiguous sentences in the language We are using. Let us take an example of the following English sentence – "In books selected information is given". The word 'selected' may refer to books or information. So the sentence may be parsed in two different ways. The same situation may arise in context-free languages.



*(a) First Derivation Tree for String a + a * b*

*(b) Second Derivation Tree for String a + a * b*

Fig. 2.3 Two Different Derivation Trees for One String a + a * b

The same terminal string may be the yield of two different derivation trees. So there may be two different leftmost derivations for a string. This leads to the definition of ambiguous sentences in a context-free language, as follows –

A terminal string $u \in L(G)$ is ambiguous, if there exist two or more derivation trees for u (or there exist two or more leftmost derivations of u).

For example, $G = (\{S\}, \{a, b, +, *\}, P, S)$, where P consists of

$$S \rightarrow S + S \mid S * S \mid a \mid b$$

We have two derivation trees for string a + a * b, given in fig. 2.3.

The leftmost derivations of a + a * b induced by the two derivation trees are as follows –

(i) $S \Rightarrow S + S \Rightarrow a + S \Rightarrow a + S * S \Rightarrow a + a * S \Rightarrow a + a * b$

(ii) $S \Rightarrow S * S \Rightarrow S + S * S \Rightarrow a + S * S \Rightarrow a + a * S \Rightarrow a + a * b$

Therefore, string a + a * b is ambiguous.

So, we can define that a context-free grammer G is ambiguous if there exists some $u \in L(G)$, which is ambiguous.

Q.9. Write down the difficulties with top-down parsing. How can we eliminate the left-recursion in top-down parsing ?

Ans. There are several difficulties with top-down parsing as follows –

The first difficulty is *left-recursion*. A grammar G is said to be left-recursive, if it has a nonterminal A such that there is a derivation $A \overset{+}{\Rightarrow} A\alpha$ some α. A left-recursive grammar can cause a top-down parses to go into an infinite loop.

A second problem is *backtracking*. The recursive descent and predictive parsers are used to avoid backtracking.

A third problem with top-down backcracking parsers is that the order in which alternates are tried can affect the language accepted.

Elimination of Left-recursion – If we have the left-recursion pair of productions $A \to A\alpha / \beta$, where β does not begin with an A, then we can eliminate the left-recursion by replacing this pair of productions with

$$A \to \beta A'$$

$$A' \to \alpha A' / \varepsilon$$

Q.10. Define left-factoring method for predictive parsing.

Ans. Left-factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parsing. The basic idea is that when it is not clear which of two alternative productions to use to expand a nonterminal A, we may be able to rewrite the A-productions to defer the decision until we have seen enough of the input to make the right choice.

In general, if $A \to \alpha\beta_1 / \alpha\beta_2$ are two productions, and the input begins with a nonempty string derived from α, we do not know whether to expand A to $\alpha\beta_1$ or to $\alpha\beta_2$. However, we may defer the decision by expanding A to $\alpha A'$ and A' to β, or to β_2. That is, left-factored, the original productions become,

$$A \to \alpha A'$$

$$A' \to \beta_1 / \beta_2$$

The following grammar abstracts the danging-else problem –

$$S \to iEtS / iEtSeS / a$$

$$E \to b$$

Here i, t, and e stand for if, then and else, E and S for "expression" and "statement".

Left-factored, this grammar becomes –

$$S \to iEtSS' / a$$

$$S' \to eS / \in$$

$$E \to b$$

Thus, we may expand S to iEtSS' on input P, and wait until iEtS has been seen to decide whether to expand S' to eS or to $\in$.

Q.11. Explain the term "parsing".

Ans. Parsing is the process or technique used to determine whether a string of tokens can be generated by a grammar. A parser must be capable of constructing the tree or the translation is incorrect. A parser can be constructed for any grammer. For any context-free-grammar, there is a power that takes at most $O(n^3)$ time to parse a string of n tokens.

The parser obtains a string of tokens from the lexical analyzer and verifies that the string can be generated by the grammar for the source language. A parser also report any syntax errors in an intelligible fashion. It should also recover from commonly occuring errors so that it can continue processing the remainder of its input.

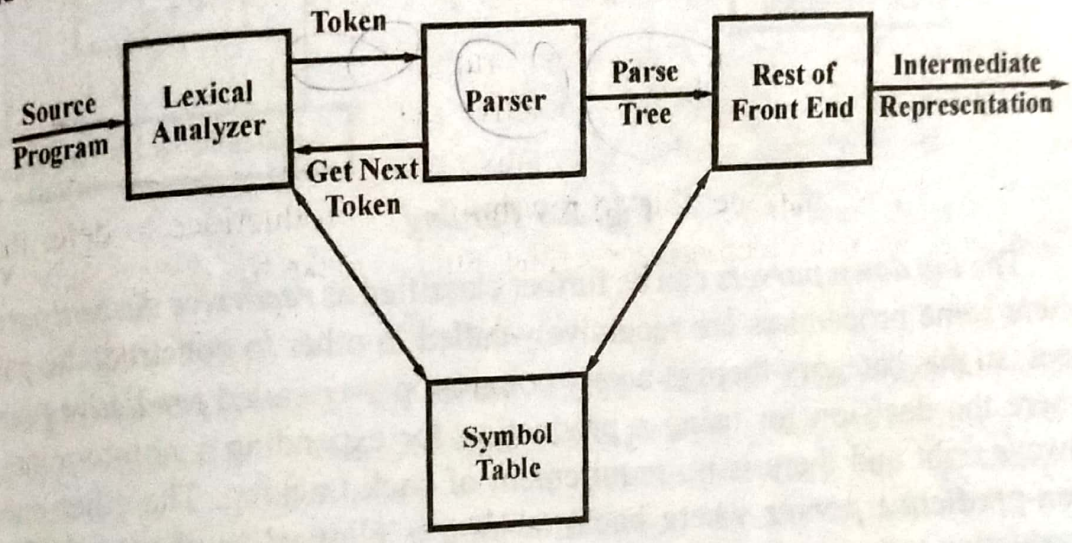


Fig. 2.4 Position of Parser in Compiler Model

This is the function of the syntax analyser. It uses the concept of derivation and reduction applied on the context free grammar. There are number of tasks that are performed during parsing such as collecting information about various tokens into the symbol table, performing type checking and semantic analysis and intermediate code.

Parsing technique can be classified into two classes. These are as follows -

(i) Top down approach

(ii) Bottom up approach.

These terms refer to the order in which nodes in the parse tree are constructed. In ***top down parsing***, construction of parse tree starts at the root and proceed towards the leaves using the leftmost derivation. While in case of ***bottom up parsing***, construction starts at the leaves and proceeds towards the root using reduction process. The top down parser can be constructed more easily by hand using top down methods, bottom up parsing, can handle a larger class of grammars and translation schemes.

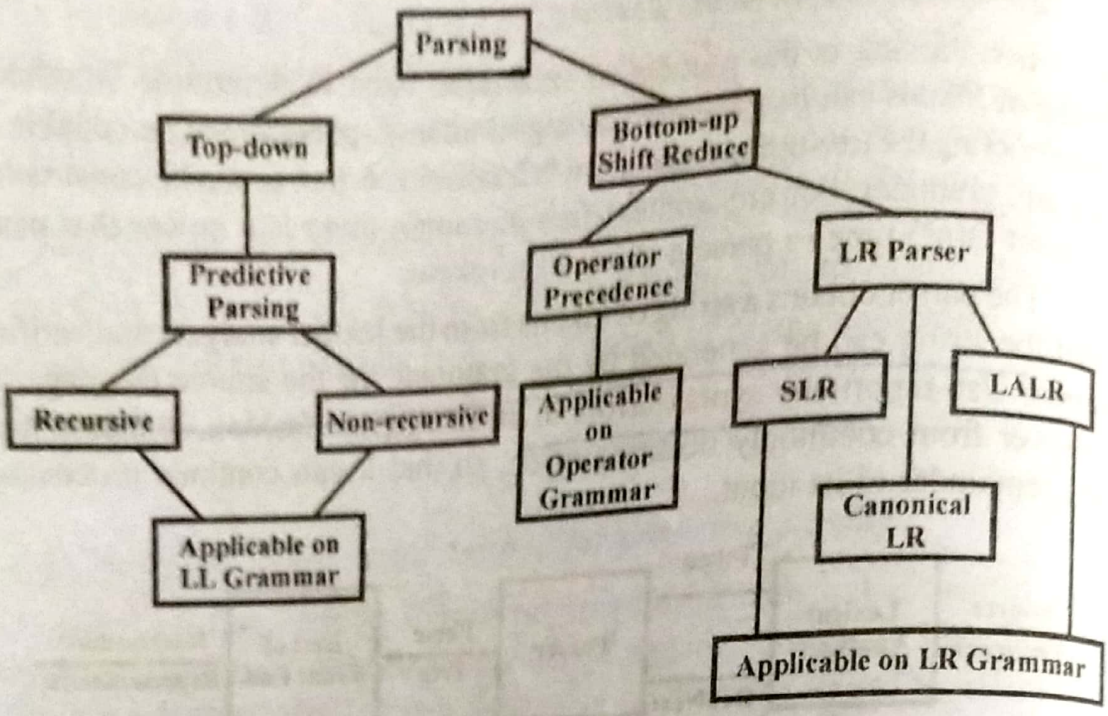


Fig. 2.5 Parsing

The ***top down parsers*** can be further classified as ***recursive descent parser*** where some procedures are recursively called in other to construct the parse trees. In this category there is a very common parser called ***predictive parser*** where the decision for using a production for expanding a non-terminal is always right and there is no requirement of back tracking. The other one is ***non-predictive parser*** where back-tracking is allowed to choose the right production trying each one.

The bottom up parser uses a technique called ***shift reduce.*** This technique is useful for many class of grammar such as operator grammar where production do not have adjacent non-terminals on right hand side neither there are ϵ production. Another large class of grammar include the LR grammar where the L stands for reading input left to right and R stands for right most reduction. These grammar again have sub classes such simple (SLR) or Lookahead LR (LALR) and canonical LR all of them have their own parser with slight difference.

Q.12. What is top-down parsing ? What are the difficulties encounter in this and how are they overcome ? ***(R.G.P.V., Dec. 2007)***

Ans. **Top-down Parsing** – Top-down parsing attempts to find the left-most derivations for an input string w, which is equivalent to constructing parse tree for the input string w that starts from the root and creates the nodes of the parse tree in a preorder. For example, consider the grammar –

$$S \rightarrow cAd$$

$$A \rightarrow ab \mid a$$

Let the input string be w = cad. To construct a parse tree for this sentence top-down, we initially create a tree consisting of a single node labeled S. An input pointer points to c, the first symbol of w. We then use the first production for S to expand the tree and obtain fig. 2.6.

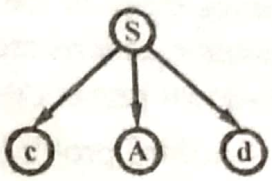


Fig. 2.6 S-production to Expand the Parse Tree

The leftmost leaf, labeled c, matches the first symbol of w, so we now advance the input pointer to a, the second symbol of w, and consider the next leaf, labeled A. We can then expand A using the first alternate for A to obtain the tree shown in fig. 2.7 (a).

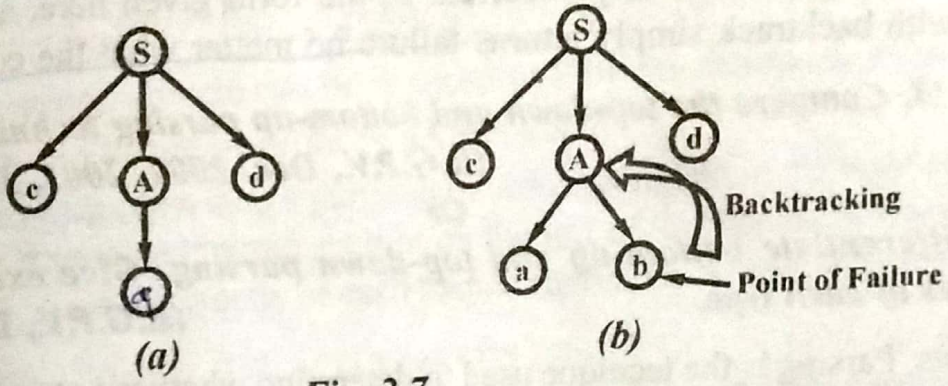


Fig. 2.7

The parser now has the match for the second input symbol. We now consider d, the third input symbol, and the next leaf, labeled b. Since b does not match d, we report failure and go back (backtracks) to A, as shown in fig. 2.7 (b). The parser will also reset the input pointer to the second input symbol and it will try a second alternative for A in order to obtain the tree.

The leaf a matches the second symbol of w and the leaf d matches the third symbol. Since we have now produced a parse tree for w, we halt and announce successful completion of parsing.

Difficulties with Top-down Parsing – There are several difficulties with top-down parsing as just presented. The first concerns left-recursion. A grammar G is said to be left-recursive if it has a nonterminal A such that there is a derivation $A \overset{+}{\Rightarrow} A\alpha$ for some α. A left-recursive grammar can cause a top-down parser to go into an infinite loop. That is, when we try to expand A, we may eventually find ourselves again trying to expand A without having consumed any input. This cycling will surely occur on an erroneous input string, and it may also occur on legal inputs, depending on the order in which the alternates for A are tried.

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,

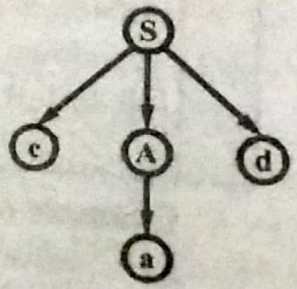


Fig. 2.8

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.

A third problem with top-down backtracking parsers is that the order in which alternates are tried can affect the language accepted. For example, if; we used a and then ab as the order of the alternates for A, we could fail to accept cabd. That is, with parse tree and ca already matched, the failure of the next input symbol, b, to match, would imply that the alternate cAd for S was wrong, leading to rejection of cabd.

Yet another problem is that when failure is reported, we have very little idea where the error actually occurred. In the form given here, a top-down parser with backtrack simply returns failure no matter what the error is.

Q.13. Compare the top-down and bottom-up parsing techniques. ***(R.G.P.V., Dec. 2002, 2005, June 2007)***

Or

Differentiate bottom-up and top-down parsing. Give examples of methods of each type. ***(R.G.P.V., Dec. 2006)***

Ans. Parsing is the tecnique used to determine whether a string of tokens can be generated by a grammar. Parsing technique can be classified into two classes i.e., top-down and bottom-up. The difference/comparison between top-down and bottom-up parsing is given below –

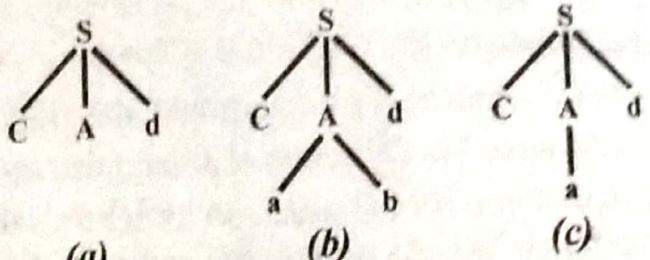
S. No.	Top-down Parsing	Bottom-up Parsing
(i)	Top-down parser build parse trees from the top (root) to the bottom (leaves).	Bottom-up parser starts from the leaves and work up to the root.
(ii)	Construction of parse tree starts using the leftmost derivation.	Construction starts using the reduction process.
(iii)	The top-down parser can be constructed more easily by hand using top-down approach.	Bottom-up parsing, can handle a larger class of grammars and translation schemes.
(iv)	Some procedures of top-down parsing are recursively.	There are no any recursive procedure in bottom-up parsing.
(v)	Top-down parsing is known as left-to-left (LL) parsing.	Bottom-up parsers are known a LR parsers and process of parsing known a shift reduce parsing.

Example of Top-down Parsing – Consider the grammar,

$S \rightarrow CAd$

$A \rightarrow ab/a$

and the input string w = cad. To construct a parse tree for this string top-down, we initially create a tree consisting of a single node labeled S. We then use the productions for S to expand the tree and obtain the tree of fig. 2.9



(a) (b) (c)

Fig. 2.9

Example of Bottom-up Parsing – Consider the grammar,

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

The sentence abbcde can be reduced to S by the following steps –

$$abbcde \rightarrow aAbcde \rightarrow aAde \rightarrow aABe \rightarrow S$$

At each ***reduction*** step a particular substring matching the right side of a production is replaced by the symbol on the left of that production and if the substring is chosen correctly at each step, a rightmost derivation is traced out in reverse.

Q.14. What is predictive parser ? How a parser is controlled by a program ? ***(R.G.P.V., Dec. 2003, 2007)***

Ans. A predictive parser is an efficient way of implementing recursive-descent parsing by handling the stack of activation records explicitly. We can picture a predictive parser as in fig. 2.10.

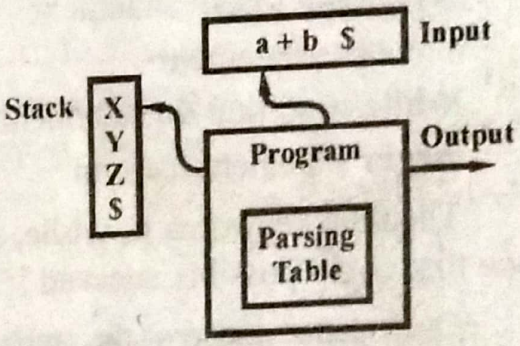


Fig. 2.10 Model of a Predictive Parser

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 $,the 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 follow. The program determines X, the symbol on top of the stack, and a, the current input symbol. These two symbols determine the action of the parser. There are three possibilities –

(i) If X = a = $, the parser halts and announces successful completion of parsing.

(ii) If X = a ≠ $, the parser pops X off the stack and advances the input pointer to the next input symbol.

(iii) 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. As output, the grammar does the semantic action associated with this production, which, for the time being, we shall assume is just printing the production used. If M [X, a] = **error**, the parser calls an error recovery routine.

Q.15. Write short note on recursive descent parser.

(R.G.P.V., Dec. 2002)

Ans. Recursive descent parsing is a top-down method of syntax analysis in which we execute a set of procedures to process the input. In many practical cases a top-town parser needs no backtrack. In order that no backtracking be required, we must know, given the current input symbol a and the nonterminal A to be expanded, which one of the alternates of production A $\alpha_1 |\alpha_2|...|\alpha_n$ is the unique alternate that derives a string begining with a. That is, the proper alternate is detectable by looking at only the first symbol it derives. For example, control constructs with their distinguishing keywords, are detectable in this way. Suppose we have productions –

Statement –
if condition **than** statement
else statement
|**while** condition **do** statement
|**begin** statement-list **end**

Then the keywords **if, while,** and **begin** tell us which altrnate is the only one that could possibly succeed if we are to find a statement.

One nuance concerns the empty string. If one alternate for A is ∈, and none of the other alternates derives a string begining with a, then on input a we may expand A by A → ∈, that is, we suceed without further ado in recongnizing an A.

A parser that uses a set of recursive procedures to recognize its input with no backtracking is called a recursive-descent parser. The recursive procedures can be quite easy to write and fairly efficient if written in a language that implements procedure calls efficiently. To avoid the necessity of a recursive language, we shall also consider a tabular implementation 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.

Q.16. Describe nonrecursive predictive parsing method.

Ans. A nonrecursive predictive parser make use of stack to find out the suitable production to applied for a nonterminal to construct a top-down parsing.

A table-driven predictive parser has an input buffer, a stack, a parsing table and an output stream. The input buffer contains the string to be parsed, followed by $ an end of input string right end marker. The stack contains a sequence of grammar symbols with $ at the bottom, indicating bottom of stack. Initially the stack contains the start symbol of the grammar on top of $. The parsing table is a two dimensional array M[A,a], where A is non-terminal and 'a' is terminal or $.

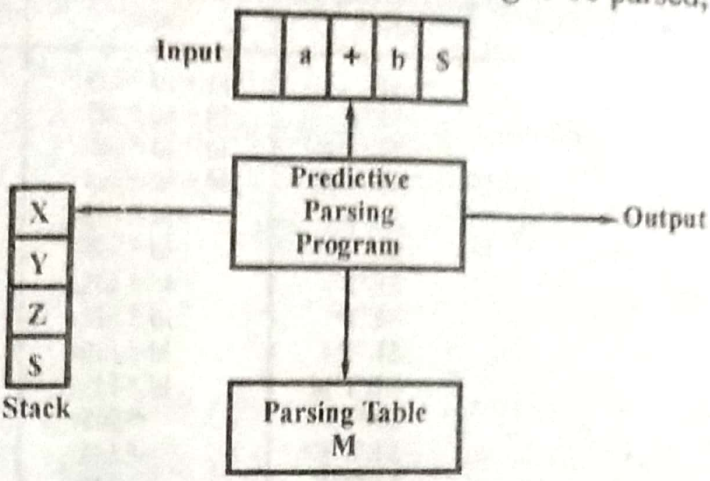


Fig. 2.11 Model of a Non-recursive Predictive Parser

The parsing program proceeds as follows. If top of stack is X and current input is 'a' then

(i) IF X = a = $, successful parsing is declared by parser.

(ii) If X = a ≠ $, parser pops X off and advance the input pointer to next symbol.

(iii) If X is a non-terminal, the program consults entry M[X, a] of parsing table M. This entry will be either an X-production or an error entry. If it carries the entry X → uvw then parser replaces X on top of stack by wvu. If M[X, a]= error the parser calls an error recovery routine.

Consider the grammer

$$E \rightarrow TE'$$
$$E' \rightarrow + TE'/\in$$
$$T \rightarrow FT'$$
$$T' \rightarrow * FT'/\in$$
$$F \rightarrow (E)/id$$

with input id + id * id the parser works in the following way by making use of given parsing table.

Table 2.1 Parsing Table M for Grammar

Non-terminal	Input Symbol					
	id	+	*	(	)	$
E	E → TE'			E → TE'		
E'		E' → + TE'			E' → ∈	E' → ∈
T	T → FT'			T → FT'		
T'		T' → ∈	T' → *FT'		T' → ∈	T' → ∈
F	F → id			F → (E)		

*Table 2.2 Moves Made by Predictive Parser on Input id + id * id*

Stack	Input	Output
$E	id + id * id$	
$E'T	id + id * id$	E → TE'
$E'T'F	id + id * id$	T → FT'
$E'T'id	id + id * id$	F → id
$E'T'	+id * id$	
$E'	+ id * id$	T' → ∈
$E'T +	+ id * id$	E' → +TE'
$E'T	id * id$	
$E'T'F	id * id$	T → FT'
$E'T'id	id * id$	F → id
$E'T'	* id$	
$E'T'F*	* id$	T' → *FT'
$E'T'F	id$	
$E'T'id	id$	F → id
$E'T'	$	
$E'	$	T' → ∈
$	$	E' → ∈

Construction of Predictive Parsing Table – To construct a parsing table for predictive parsing two functions are used which are called **FIRST** and **FOLLOW.** These functions allow to fill in the entries of predictive parsing table.

FIRST (α) is the function applied on a string of variable and terminal of any size is defined as the set of the terminals that begin the strings derived from α. If $\alpha \Rightarrow \in$ Then ∈ is also in FIRST (α).

To compute FIRST (x) the following rules are applied –

(i) If X is terminal the FIRST (X) is X

(ii) If $X \rightarrow \in$ is a production then add ∈ to FIRST (x).

(iii) If X is a nonterminal and $X \rightarrow y_1\ y_2 \ldots\ldots y_k$ is any production place 'a' in FIRST (x) if 'a' is in FIRST (y_i) and ∈ is in all FIRST (y_1), FIRST (y_2) ..... FIRST (y_{i-1}). If ∈ is in all FIRST (y_j) where j = 1, 2, ... k, then add ∈ to FIRST (x).

FOLLOW (A) is the function applied on a nonterminal and defined as the set of terminals that can appear immediately to the right of A.

To compute FOLLOW (A) for all nonterminals A, apply the following rules –

(i) Place $ in FOLLOW (S), where S is the start symbol.

(ii) If there is a production $A \rightarrow \alpha B\beta$ then everything in FIRST (β) except for ∈ is placed in FOLLOW (B).

(iii) If there is a production $A \rightarrow \alpha B$ or a production $A \rightarrow \alpha B\beta$ where FIRST (β) contains $\in$ (i.e., $\beta \overset{*}{\Rightarrow} \in$) , then everything in FOLLOW (A) is in FOLLOW (B).

By making use of above dicussed grammar which is unambiguous, left recursion free, FIRST and FOLLOW function are determined.

FIRST (E) = FIRST (T) = FIRST (F) = {(, id}
FIRST (E') = {+, $\in$}
FIRST (T') = {*, $\in$}
FOLLOW (E) = FOLLOW (E') = {), $}
FOLLOW (T) = FOLLOW (T') = {+,), $}
FOLLOW (F) = {+, *,), $}

Construction of Parser Table – The idea behind the algorithm is the following. If the FIRST and FOLLOW functions are evaluated then this algorithm is applied. If $A \rightarrow \alpha$ is a production with a in FIRST (α), then the parser will expand A by α when current input symbol is a when $\alpha = \in$ or $\alpha \overset{*}{\Rightarrow} \in$ then we should expand A by α if the current input symbol is in FOLLOW (A) or if θ is the next input with θ in FOLLOW (A).

ALGORITHM

(i) For each production $A \rightarrow \alpha$ of the grammar do steps 2 and 3.

(ii) For each terminal a in FIRST (α) add $A \rightarrow \alpha$ to M[A, a].

(iii) If $\in$ is in FIRST (α), add $A \rightarrow \alpha$ to M[A, b] for each terminal b in FOLLOW (A). If $\in$ is in FIRST (α) and $ in FOLLOW (A) add $A \rightarrow \alpha$ to M[A, $].

(iv) Make each undefined entry of M be error.

Q.17. Explain bottom up parsing method.

Ans. A bottom up parser attempts to construct a parse tree for an input string beginning at the leaves and working up towards the root. This is called the reduction process. At each reduction step a particular substring matching the right side of a production is replaced by the symbol on the left of that production, and if the steps are correct then a right most derivation is traced out in reverse.

For example – Consider the grammar

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

The sentence abbcde can be reduced to S by the following steps –

abbcde
aAbcde
aAde
aABe
S

A "handle" of a string is a substring that matches the right side of a production, and whose reduction to the non-terminal on the left side of the production represents one step along the reverse of a rightmost derivation. A handle of a right sentential form γ is a production $A \rightarrow \beta$ and a position of γ where the string β may be found and then replaced by A to produce the previous right sentential form in a rightmost derivation of γ. In the above example abbcde is a right sentential form, $A \rightarrow b$ is a handle at position 2. Like wise, aAbcde is a right sentential form whose handle is $A \rightarrow Abc$ at position 2 A point to be noted that the position where the handle is applied do have only terminals in the right of the handle marked. Consider the following grammar –

(i) $E \rightarrow E + E$
(ii) $E \rightarrow E * E$
(iii) $E \rightarrow (E)$
(iv) $E \rightarrow id$

and the right most derivations

$$E \underset{rm}{\Rightarrow} E + E \qquad E \underset{rm}{\Rightarrow} E * E$$

$$\underset{rm}{\Rightarrow} E + E * E \qquad \underset{rm}{\Rightarrow} E * id_3$$

$$\underset{rm}{\Rightarrow} E + E * id_3 \qquad \underset{rm}{\Rightarrow} E + E * id_3$$

$$\underset{rm}{\Rightarrow} E + id_2 * id_3 \qquad \underset{rm}{\Rightarrow} E + id_2 * id_3$$

$$\underset{rm}{\Rightarrow} id_1 + id_2 * id_3 \qquad \underset{rm}{\Rightarrow} id_1 + id_2 * id_3$$

A rightmost derivation in reverse, called a ***canonical reduction sequence*** is obtained by "handle pruning". In this method, we start with a string of terminals w, to be parsed. If w is a sentence of the grammar at hand then $w = \gamma_n$, where γ_n is the n^{th} right sentential form

$$S = \gamma_0 \underset{rm}{\Rightarrow} \gamma_1 \underset{rm}{\Rightarrow} \gamma_2 \underset{rm}{\Rightarrow} \ldots\ldots\ldots\ldots \underset{rm}{\Rightarrow} \gamma_{n-1} \underset{rm}{\Rightarrow} \gamma_n = w$$

To reconstruct this derivation in reverse order, the handle β_n in γ_n is replaced by left side of some production $A_n \rightarrow \beta_n$ to obtain the (n – 1) st right sentential form γ_{n-1}. This process is repeated until start symbol is left.

A shift reduce parser is implemented by using a stack to hold grammar symbols and an input buffer to hold the string w to be parsed. $ is used to mark the bottom of the stack and also the right end of the input. Initially, the stack is empty and string w is on the input. The parser operates by shifting zero or more input symbols onto the stack until a handle β is on top of the stack. The parser then reduces β to the left side of appropriate production. The parser repeat this process until it has detected an error or the stack contains the start symbol and the input is empty.

Table 2.3 Configurations of Shift-reduce Parser on Input $id_1 + id_2 * id_3$

	Stack	Input	Action
(1)	$	$id_1 + id_2 * id_3$$	shift
(2)	$id_1	$+ id_2 * id_3$$	reduce by E → id
(3)	$E	$+ id_2 * id_3$$	shift
(4)	$E +	$id_2 * id_3$$	shift
(5)	$E + id_2	$* id_3$$	reduce by E → id
(6)	$E + E	$* id_3$$	shift
(7)	$E + E *	$id_3$$	shift
(8)	$E + E * id_3	$$	reduce by E → id
(9)	$E + E * E	$$	reduce by E → E * E
(10)	$E + E	$$	reduce by E → E + E
(11)	$E	$$	accept

There are four possible actions performed by a shift-reduce parser.

(i) Shift Action – The next input symbol is shifted onto the top of stack.

(ii) Reduce Action – The right end of the handle is at the top of stack. The parser locates the left end of the handle within the stack and replace the L.H.S. of handle.

(iii) Accept Action – The parser announces successful completion of parsing.

(iv) Error Action – A syntax error is detected and calls an error recovery routine.

Q.18. What is the role of handle pruning in parsing ?
(R.G.P.V., June 2008)

Ans. A handle of a string S is a substring a such that –

(i) 'a' matches the RHS of a production A → a; and

(ii) Replacing 'a' by the LHS 'A' represents a step in the reverse of 'a' rightmost derivation of S.

Example –

Consider a grammar

$S \rightarrow aABe$

$A \rightarrow Abc|b$

$B \rightarrow d$

The rightmost derivation for the input abbcde is

$S \Rightarrow aABe$

$\Rightarrow aAde$

$\Rightarrow aAbcde$

$\Rightarrow abbcde$

The string aAbcde can be reduced in two ways –

(i) $aAbcde \Rightarrow aAde$; and

(ii) $aAbcde \Rightarrow aAbcBe$

But (2) is not a right most derivation, so Abc is the only handle.

The strings to the right of a handle will only contain non-terminals.

The parsing is required to construct a derivation. A bottom-up parser builds a derivation by working from the input sentence back towards the start symbol S.

$S \Rightarrow y_0 \Rightarrow y_1 \Rightarrow y_2 \ldots\ldots \Rightarrow y_{n-1} \Rightarrow y_n \Rightarrow$ sentence

The parser must find a substring β of the tree's frontier that – matches some production $A \rightarrow \beta$ that occurs as one step in the rightmost derivation.

Informally, we call this substring β a handle.

Formally, a handle of a right-sentential form γ is a pair $< A \rightarrow \beta, k >$ where $A \rightarrow \beta \in P$ and k is the position in γ of β's rightmost symbol.

The process of discovering a handle & reducing it to the appropriate left-hand side is called handle pruning.

Handle pruning forms the basis for a bottom-up parsing method.

To construct a right most derivation

$S \Rightarrow y_0 \Rightarrow y_1 \Rightarrow y_2 \Rightarrow \ldots\ldots \Rightarrow y_{n-1} \Rightarrow y_n \Rightarrow w$

Apply the following simple algorithm

for $i \leftarrow n$ to 1 by – 1

Find the handle $< A_i \rightarrow \beta_i, k_i >$ in y_i

Replace β_i with A_i to generate y_{i-1}

This takes 2n steps

One implementation technique is shift-reduce parser. Shift reduce parsers are easily built and easily understood.

Parsing can be thought of as handle pruning.

Q.27. Explain the various steps for constructing SLR parsing tables.

Ans. SLR is known as "simple LR", is the easiest method to implement. An LR (0) item of a grammar G is a production of G with a dot at some position of the right side. Thus, production $A \rightarrow XYZ$ yields the following items

$$A \rightarrow \cdot XYZ$$
$$A \rightarrow X \cdot YZ$$
$$A \rightarrow XY \cdot Z$$
$$A \rightarrow XYZ \cdot$$

The production $A \rightarrow \in$ generates $A \rightarrow \cdot$. An item can be represented by a pair of integers, the first giving the number of the production and second the position of the dot.

The SLR methods is used to construct from the grammar a deterministic finite automation to recognize viable prefixes items are grouped into sets which give rise to the states of the SLR parser. Canonical LR(0) collection is used for constructing SLR parser, for this we define augmented grammar and two function, **CLOSURE** and **GOTO.**

(i) Augmented Grammar – If G is a grammar with start symbol S, then G' is the augmented grammar for G, the G with a new start symbol S' and production $S' \rightarrow S$. This indicates the completion of parsing and acceptance of the input. The acceptance occurs when parser is about to reduce by $S' \rightarrow S$.

(ii) Closure Operation – If I is a set of items for a grammar G, the CLOSURE(I) is constructed from I by two rules –

(a) Every item in I is added to CLOSURE(I)

(b) If $A \rightarrow \alpha.B\beta$ is in CLOSURE(I) and $B \rightarrow \gamma$ is a production, then add $B \rightarrow \cdot \gamma$ to I. Apply this until no more new items can be added to closure (I). For example –

$$E' \rightarrow E$$
$$E \rightarrow E + T|T$$
$$T \rightarrow T * F|F$$
$$F \rightarrow (E)|id$$

then closure (I) contains the item,

$$E' \rightarrow \cdot E$$
$$E \rightarrow \cdot E + T$$
$$E \rightarrow \cdot T$$
$$T \rightarrow \cdot T * F$$
$$T \rightarrow \cdot F$$
$$F \rightarrow \cdot (E)$$

(iii) GoTo Operation – GOTO (I, x) is defined to be the closure of the set of all items $[A \to \alpha X . \beta]$ such that $[A \to \alpha .X\beta]$ is in I, where I is a set of items and X is a grammar symbol. If I is the set of valid items for some viable prefix γ, then goto (I, x) is the set of items that are valid for the viable prefix γx. The goto function can be applied on the above grammar the following

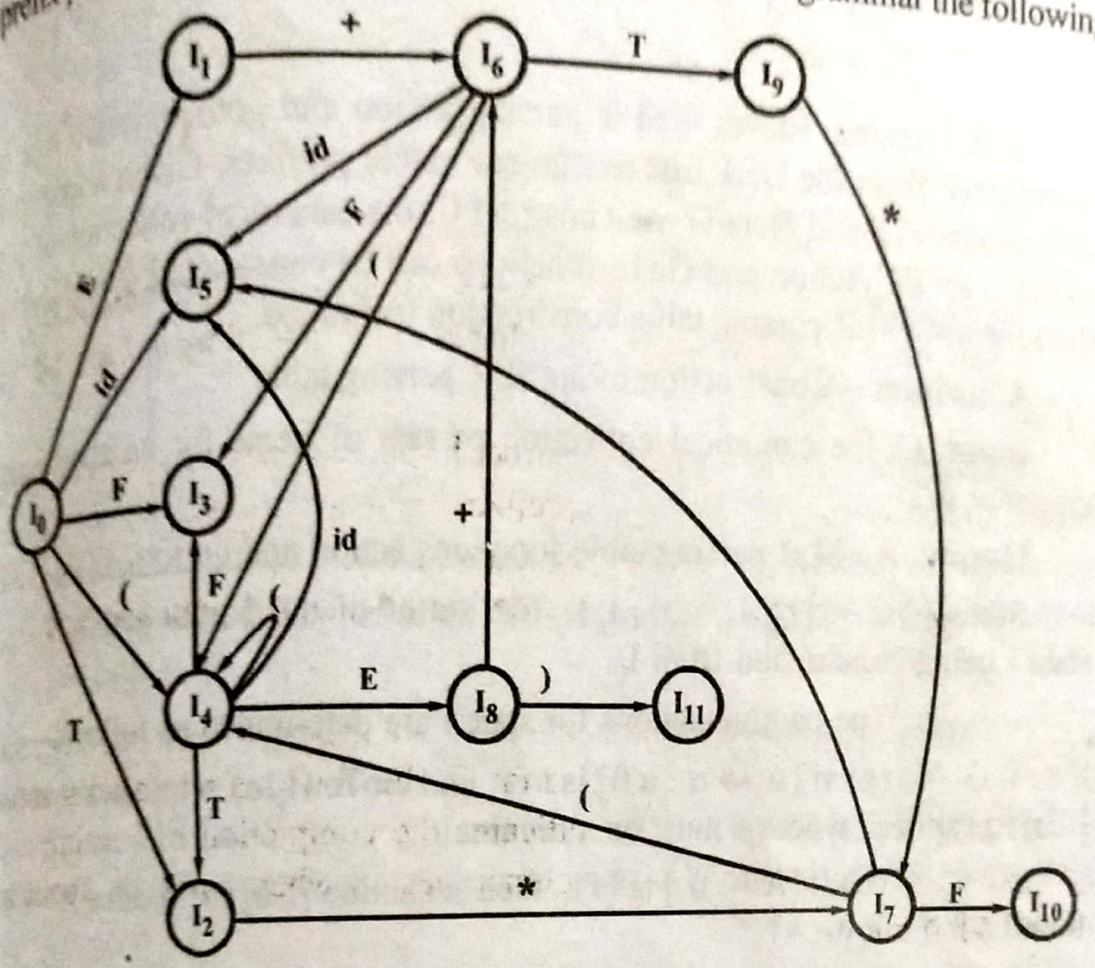


Fig. 2.13 Deterministic Finite Automation D

canonical LR(0) collection for grammar are

I_0 : $E' \to \cdot E$
$E \to \cdot E + T$
$E \to \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot id$

I_1 : $E' \to E\cdot$
$E \to E \cdot + T$

I_2 : $E \to T\cdot$
$T \to T\cdot * F$

I_3 : $T \to F\cdot$

I_4 : $F \to (\cdot E)$

I_5 : $F \to id\cdot$

I_6 : $E \to E + \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot id$

I_7 : $T \to T * \cdot F$
$F \to \cdot (E)$
$F \to \cdot id$

I_8 : $F \to (E\cdot)$
$E \to E\cdot + T$

I_9 : $E \to E + T\cdot$
$T \to T\cdot * F$

$E \rightarrow \cdot E + T$
$E \rightarrow \cdot T$
$T \rightarrow \cdot T * F$
$T \rightarrow \cdot F$
$F \rightarrow \cdot (E)$
$F \rightarrow \cdot id$

$I_{10} : T \rightarrow T * F\cdot$

$I_{11} : F \rightarrow (E)\cdot$

SLR Parsing Tables – SLR parsing action and goto function can be constructed from the DFA that recognizes viable prefixes. Given a grammar G to produce G' and from G' we construct C, the canonical collection of sets of items for G'. Action and GoTo functions can be constructed from C using the following SLR parsing table construction technique.

Alogrithm – Construction of an SLR parsing table.

Input. C, the canonical collection of sets of items for an augmented grammar G'.

Output. An SLR parsing table functions action and go to for G'.

Steps (i) $C = \{I_0, I_1, ...I_n\}$. The states of the parser are 0, 1 .....n state i being constructed from I_i.

(ii) The parsing actions for state i are determined as follows –

(a) If $[A \rightarrow \alpha . a \beta]$ is in I_1 and GoTo $(I_i, a) = I_j$ then set action [i, a] to "shift j." where a must be a terminal.

(b) If $[A \rightarrow \alpha.]$ is in I_i, then set action [i, a] to "reduce $A \rightarrow \alpha$" for all a in Follow (A) .

(c) If $[S' \rightarrow S.]$ is in I_i, then set action {i, $] to "accept".

The goto transitions for state i are constructed using the following rules –

(iii) If goto $(I_i, A) = I_j$, then goto [i, A] = j

(iv) All entries not defined by rules (ii) and (iii) are made "error".

(v) The initial state of the parser is the one constructed from the set of items containing $[S' \rightarrow .S]$.

The parsing table consisting of the parsing action and goto functions is called the SLR(1) table for G. The SLR parsing table for the given example is shown in table 2.6.

For constructing the SLR passing table, the canonical collection of sets of LR(0) items.

The item $F \rightarrow \cdot (E)$ gives rise to the entry action [0, ( ] = shift 4, the item $F \rightarrow id$ to the entry action [0, id] = shift s. Other items in I_0 yield no actions. Now consider I_1 –

$E^i \rightarrow E$
$E \rightarrow E\cdot + T$

Table 2.6 Parsing Table for Expressing Grammar

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			

The first item yields action [1, $] = accept, the second item yield action [1, +] = shift 6. For I_2 –

$$E \rightarrow T\cdot$$
$$\Gamma \rightarrow T\cdot * F$$

Since, FOLLOW (E) = {$, +,)}, the first item makes action [2, $] = action [2, +] = action [2,)] = reduce E → T·. The second item makes acton [2, *] = shift 7.

Continuing in this fashion we obtain the parsing action and goto tables. The number of productions in reduce actions are the same as the order in which they appear in the original grammar. That is, E → E + T is number 1, E → T is 2, and so on.

Q.30. How canonical LR parsing table and LALR parsing table constructed ? Explain.

Ans. **Canonical LR Parser** – There are certain grammar for which if SLR parsing tables are constructed some position have multiple entries one to shift and other to reduce. This can be avoided in canonical LR parsing. To understand this two terms must be defined –

Capable of working/working successfully

(i) ***Viable Prefix*** – The viable prefixes of a grammar is the prefixes of right sentential form that do not contain any symbols to the right of the handle. For SLR parser we built the stack till we get a viable prefix on top of stack and there we apply a reduction process.

(ii) ***Valid Item*** – An item $A \to \beta_1 \cdot \beta_2$ is valid for a viable prefix $\alpha\beta_1$ if there is a derivation $S \overset{*}{\Rightarrow} \alpha Aw \Rightarrow \alpha\, \beta_1\, \beta_2\, w$ if it is so then finding $\alpha\beta_1$ in stack shift or reduce is done unless $\beta_2 = \epsilon$.

The general form for such kind of item now becomes $[A \to \alpha.\beta, a]$ where $A \to \alpha\beta$ is a production, 'a' is a terminal or right endmarker \$. This is called an LR(1) item. The number (1) refers to the length of second component. If it is taken after looking next K then this is called LR(K) grammar. The

second component is called the lookahead component. LR(1) item $[A \to \alpha.\beta,a]$ is valid for a viable prefix γ if there is a derivation $S \overset{*}{\Rightarrow} \delta Aw \Rightarrow \delta\alpha\beta w$, where

(a) $\gamma = \delta\alpha$ and

(b) either a is first symbol in w or w is $\in$ i.e., a = is \$.

The method for constructing the collection of sets of valid LR(1) items is same as that in SLR parser using closure and goto function. Here closure of item $[A \to \alpha.B\beta, a]$ is found which is valid for some viable prefix γ, leads to inclusion of all $[B \to \eta, b]$ item where $B \to \eta$ for any η are the B production and b is all symbol from FIRST $[\beta a]$ consider an example.

$S' \to S$
$S \to CC$
$C \to cC/d$

The canonical collection of LR(1) item are –

I_0 : $S' \to \cdot S, \$$
$S \to \cdot CC, \$$
$C \to \cdot cC, c/d$
$C \to \cdot d, c/d$

I_1 : $S' \to S\cdot, \$$

I_2 : $S \to C\cdot C, \$$
$C \to \cdot cC, \$$
$C \to \cdot d, \$$

I_3 : $C \to c\cdot C, c/d$
$C \to \cdot cC, c/d$
$C \to \cdot d, c/d$

I_4 : $C \to d\cdot, c/d$

I_5 : $S \to CC\cdot, \$$

I_6 : $C \to c\cdot C, \$$
$C \to \cdot cC, \$$
$C \to \cdot d, \$$

I_7 : $C \to d\cdot, \$$

I_8 : $C \to cC\cdot, c/d$

I_9 : $C \to cC\cdot, \$$

Algorithm – Construction of canonical LR parsing table.

Input. An augmented grammar G'

Output. The canonical LR parsing table functions action and goto for G'.

Method (i) contruct $C = \{I_0, I_1, \ldots\ldots I_n\}$, the collection of sets of LR(1) items for G.

(ii) State i of the parser is constructed from I_i'. The parsing actions for state i are determined as follows –

(a) If $[A \to \alpha.a\beta,b]$ is in I_i and Goto $(I_i, a) = I_j$ then set action $[i, a]$ to "Shift j".

Table 2.9 Canonical Parsing Table

State	action			goto	
	c	d	$	S	C
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

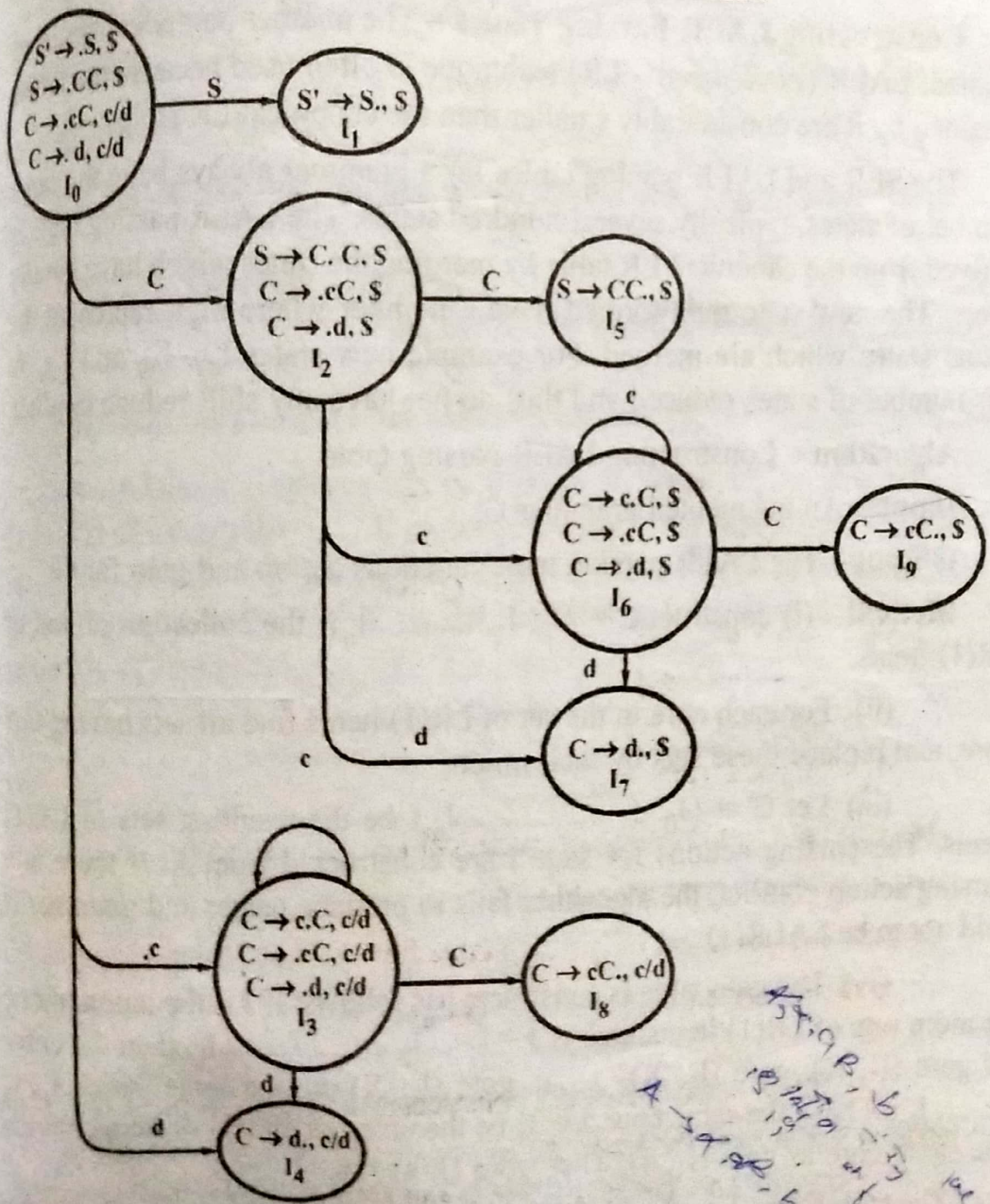


Fig. 2.14 The Goto Graph

(b) If $[A \rightarrow \alpha .. a]$ is in I_i, then set action [i, a] to "reduce $A \rightarrow \alpha$."

(c) If $[S' \rightarrow S .. \$]$ is in I_i, then set action [i, $] to "accept".

(iii) The goto transition for state i are determined as follows. If Goto $(I_i, A) = I_j$, then Goto [i, A] = j.

(iv) All entries not defined by rule (ii) through (iii) are made "error".

(v) The initial state of the parser is the one constructed from the set containing item $[S' \rightarrow . S, \$]$.

The table formed is called canonical LR(1) parsing table, An LR parser using this table is called a canonical LR parser. Table is shown in table 2.9 and graph is shown in fig. 2.14.

Constructing LALR Parsing Tables – The another parser construction method, LALR (Lookahead – LR) technique is often used because the tables obtained by it are considerably smaller than the canonical LR tables.

The SLR and LALR parsing tables for a grammar always have the same number of states, typically several hundred states. The LALR parsing table is derived from the canonical LR table by merging the states which have similar cores. The new state thus formed have a number where digit represent the actual states which are merged. For example new states I_{47}, I_{89} and I_{36}. In this number of states reduced and thus do not have any shift reduce conflict.

Algorithm – Constructing LALR parsing table.

Input – An augmented grammar G'

Output – The LALR parsing table functions action and goto for G'

Method – (i) construct $C = \{I_0, I_1, ... I_n\}$, the collection of sets of LR(1) items.

(ii) For each core in the set of LR(1) items find all sets having that core, and replace these sets by their union.

(iii) Let $C' = \{J_0, J_1 ..... J_m\}$ be the resulting sets of LR(1) items. The parsing actions for state i are constructed from J_i. If there is a parsing action conflict, the algorithm fails to produce parser and grammar is said not to be LALR(1)

(iv) The goto table is constructed as follows. If J is the union of one or more sets of LR(1) items, that is $J = I_1 \cup I_2 \cup ..... \cup I_k$, then the cores of goto (I_1, X), goto (I_2, X), ... goto (I_k, X) are the same, since I_1, I_2 ..... I_k all have the same core. Let K be the union of all sets of items having the same core as goto (I_1, x). Then goto $(J, x) = K$.

The table formed is called LALR parsing table for G. If there are no parsing action conflicts, then the given grammar is said to be an LALR(1) grammar.

Table 2.10 LALR Parsing Table

State	action			goto	
	c	d	$	S	C
0	s36	s47		1	2
1			acc		
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

Q.31. Write the Algo/Procedure for construction of the canonical LR parsing table. ***(R.G.P.V., June 2004)***

Ans. Refer to Q.30.

Q.32. Describe the function of the LALR parser generator "Yacc".

Or

Write short note on "YACC". ***(R.G.P.V., June 2003, Dec. 2003, 2004, 2005, June 2007, Dec. 2007)***

Or

Write short note on automatic parser generator. (R.G.P.V., June 2005, 2006)

Ans. A parser generator can be used to facilitate the construction of the front end of a compiler. LALR parser generator 'yacc' stands for "yet another compiler-compiler", created by S.C. Johnson in early 1970's. Yacc is available as a command on the UNIX system, and has been used to help implement hundreds of compilers.

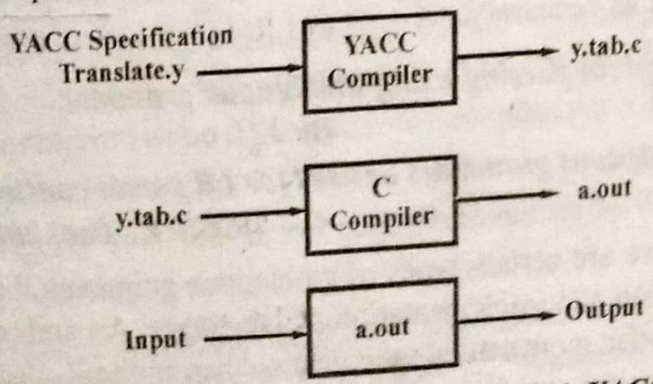


Fig. 2.15 Creating an I/O Translator with YACC

A translator can be constructed using 'yacc' by first, a file, translate. y, containing a 'yacc' specification of the translator is prepared. The UNIX system command.

yacc translate.y

transforms the file translate.y into a C program called y.tab.c using the LALR method. The program y.tab.c is a represen-tation of an LALR parser written in C. By compiling y.tab.c along with the Ly library that contains LR parsing program using the command.

cc y.tab.c – ly.

We obtain the desired object program 'a'.out that performs the translation specified by the original 'yacc' program.

A 'yacc' source program has three parts –

declarations

% %

translation rules

% %

supporting c – routines.

(i) The Declarations Part – There are two optional sections in the declarations part of a 'yacc' program. In the first section, ordinary c declarations like delimited by % { and % } are put. Here we place declarations of any temporaries used by the translation rules or procedures of the second and third sections. Declarations part also contain declarations of grammar tokens.

(ii) The Translation Rules Part – The translation rules are put after the first % % pair in the part of 'yacc' specification. Each rule consists of a grammar production and the associated semantic action. A 'yacc' semantic action is a sequence of c statements.

(iii) The Supporting C-routines Part – The third part of a 'yacc' specification consists of supporting c-routines. A lexical analyser by the name yylex() must be provided. Other procedures such as error recovery routines may be added as necessary.

Prob.12. Consider the following grammar –

$S \rightarrow Aa / b$

$A \rightarrow Ac / Sd / \in$

remove left recursion.

Sol. Above problem, there is only immediate left recursion low in this problem we can't easily remove left recursion so a special algo required to remove left recursion completly.

Algo for eliminating left recursion –

(1) Arrange the nonterminals in some order $A_1, A_2, ..., A_n$.

(2) for i = 1 to n do begin

for j = 1 to i – 1 do begin

replace each production of the form $A_i \rightarrow A_j\gamma$ by the production

$A_i \rightarrow \delta_1\gamma / \delta_2\gamma / .... / \delta_k\gamma$

where $A_j \rightarrow \delta_1/\delta_2 ..../\delta_k$ are all current A_j-productions;

end

eliminate the immediate left recursion among the A_i-production

end

Apply the above algo for given grammar

$S \rightarrow Aa / b$

$A \rightarrow bdA' / A'$

$A' \rightarrow cA' / adA' / \in$

Prob.13. Do left factoring in the following grammar –

$A \rightarrow aAB / aA / a$

$B \rightarrow bB / b$

(R.G.P.V., June 2003)

Sol. Apply left factoring procedure in the given grammar

$A \rightarrow \alpha\beta_1/\alpha\beta_2$

$A \rightarrow \alpha A'$

$A' \rightarrow \beta_1/\beta_2$

Compare the production with left factoring production. Then, we get new production A and A'. On comparing,

$$\alpha = a; \beta_1 = AB$$
$$\beta_2 = A, \beta_3 = \epsilon$$

Then, new production are –

$$A \rightarrow aA'$$
$$A' \rightarrow AB/A/\epsilon$$

Apply the same procedure for another production.

$$B \rightarrow bB'$$
$$B' \rightarrow B / \epsilon$$

Prob.14. For the following grammar find FIRST and FOLLOW sets for each of nonterminals –

$$S \rightarrow aAB \mid bA \mid \epsilon$$
$$A \rightarrow aAb \mid \epsilon$$
$$B \rightarrow bB \mid \epsilon$$

(R.G.P.V., Dec. 2005)

Sol. FIRST(S) = FIRST(aAB) ∪ FIRST(bA) ∪ FIRST(∈)
= {a, b, ∈}

FIRST(A) = FIRST (aAb) ∪ FIRST(∈)
= {a} ∪ {∈}
= {a, ∈}

FIRST(B) = FIRST (bB) ∪ FIRST(∈)
= {b} ∪ {∈}
= {b, ∈}

FOLLOW(S) = {$}

Using the production S → aAB, we get,

FOLLOW(A) = FIRST(B) – {∈} ∪ FOLLOW(S)
= {b, ∈} – {∈} ∪ {$}
= {b, $}

FOLLOW(B) = FOLLOW(S) = {$}

Prob.15. Consider the grammar

$$S \rightarrow ACB/CbB/Ba$$
$$A \rightarrow da/BC$$
$$B \rightarrow g/\epsilon$$
$$C \rightarrow h/\epsilon$$

Calculate FIRST and follow.

(R.G.P.V., June 2009)

Sol. First (s) = First (ACB) U first (CbB) U First (Ba)
= {(d, g, ∈) U (h, ∈) (g, ∈)}
= {d, g, h, ∈}

First (A) = First (da) U first (BC)
= {(d) U (g, ∈)}
= {d, g, ∈}

First (B) = First (g) U first (∈)
= {g, ∈}

First (C) = First (h) U first (∈)
= {h, ∈}

FOLLOW (s) = {$}

Using the production S → ACB, we get

Follow (A) = {h, ∈}

Using the productions

S → ACB

S → CbB

S → Ba

A → BC, we get

Follows (B) = {$, a, h}

Using the productions

S → ACB

S → CbB

A → BC, we get

Follows (c) {$, a, h, b, g}

Prob.16. Check whether the given grammar is LL(1) ? Remove left recursion and then again verify whether it is LL(1) –

$$S \rightarrow Aa \mid b$$
$$A \rightarrow Ac \mid Sd \mid \varepsilon$$

(R.G.P.V., Dec. 2008)

Sol. Since, there is a pair of the form A → Aα | β. Hence this grammar is left recursive. So, it can not be LL(1). It contain the productions A → Ac/Sd/ε.

To eliminate the left recursion from the grammer replace this productions by the following production –

$$A \rightarrow SdA'$$
$$A' \rightarrow cA' \mid \varepsilon$$

Hence, after the elimination of left factoring the grammar will be –

$$S \rightarrow Aa \mid b$$
$$A \rightarrow SdA' \mid \varepsilon$$
$$A' \rightarrow cA' \mid \varepsilon$$

For a pair of productions

$$S \rightarrow Aa \mid b$$

FIRST (Aa) ∩ FIRST (b)
= FIRST (A) – {ε} ∪ FIRST (a) ∩ FIRST (b)
= {ε} – {ε} ∪ {a} ∩ {b}
= {a} ∩ {b}
= ϕ

Hence, the grammer is LL(1).

Prob.17. Design LL(1) parsing table for the following grammar :

$A \rightarrow AcB/cC/C$

$B \rightarrow bB/id$

$C \rightarrow CaB/BbB/B$

(R.G.P.V., June 2008)

Sol. For designing the LL(1) parsing table for the following grammar. We find the FIRST and FOLLOWS.

The given grammar is

$A \rightarrow AcB/cC/C$

$B \rightarrow bB/id$

$C \rightarrow CaB/BbB/B$

FIRST (A) → First (ACB) U First (cC) U First (C)

= {(∈) U (c) U (b, id)}

= {C, b, id, ∈}

First (B) → First (bB) U first (id)

= {(b) U (d)}

= {b, id}

First (c) → First (CcB) U First (BbB) U First (B)

= {(∈) U (b, id) U (b, id)}

= {b, id, ∈}

FOLLOWS (A) = {C, $}

FOLLOWS (B) = {c, b, $}

FOLLOWS (C) = {C, a, $}

Design of LL(1) parsing table of the following grammar is given below –

Table 2.17

Non-terminal	*Terminals*					
	a	*b*	*c*	*d*	*id*	*$*
A			A → ∈			
		A → C	A → cC		A → C	A → ∈
B		B → bB			B → id	
C	C → ∈	C → BbB/B	C → ∈		C → BbB/B	C → ∈

Prob.18. Consider the grammar –

$S \rightarrow iCtSA / a$

$A \rightarrow eS / \epsilon$

$C \rightarrow b$

whether it is LL(1) grammar. Give the explanation whether (i, t, e, b, a) are terminal symbols.

(R.G.P.V., Dec. 2002)

Sol. The given grammar is –

$S \rightarrow iCtSA/a$

$A \rightarrow eS/\in$

$C \rightarrow b$

Now, whather it is LL(1) grammar or not. We find the FIRST and FOLLOWS

First (s) = First (iCtSA) U first (a)
= {(i) U (a)}
= {i, a}

First (A) = First (eS) U first (∈)
= {() U(∈)}
= {e, ∈}

First (C) = First (b)
= {b}

	a	b	c	id	$
A	A→eA'		A→CCA'bCA'		
A'			A'→CBA'		A'→∧
B		B→bB		B→id	
C		C→BC''		C→BC''	
C'	C'→aBC'		C'→∧		C'→∧
C''	C''→C'		C''→bBC'	C→∧	C→∧

Using the production S → iCtSA and A → eS/∈, we get

Follows (S) = {e, $}

Using the production S → iCtSA, we get

Follows (A) = {$, e}

Using the production C → b

Follows (C) = {t}

The parsing table for this grammar is –

Table 2.18

Nonter-minal	Input Symbol					
	a	b	e	i	t	$
S	S → a			S → iC t SA		A → ∈
A			A → ∈ A → eS			
C		C → b				

The entry for M [S', e] contains both A → eS and A → ∈, since FOLLOW (A) = {e, $}. The grammar is ambiguous and the ambiguity is manifested. From the LL(1) grammar, table has no multiply defined entries so in above one entry is multiply so that it is not LL(1) grammar.

Prob.19. Test whether the grammar is LL(1) or not and construct predictive parsing table for it –

$S \rightarrow AaAb \mid BbBa$

$A \rightarrow \in$

$B \rightarrow \in$.

(R.G.P.V., Dec. 2006)

Sol. Since the grammar contains a pair of productions S → AaAb|BbBa, for the grammar to be LL(1), it is required that –

FIRST (AaAb) ∩ FIRST (BbBa) = ϕ

FIRST (AaAb) = FIRST(A) – {∈} ∪ FIRST (aAb) = {a}

FIRST (BbBa) = FIRST(B) – {∈} ∪ FIRST (bBa) = {b}

FIRST (AaAb) ∩ FIRST(BbBa) = {a} ∩ {b} = ϕ

Hence, the grammar is LL(1).

To construct a parsing table, the FIRST and FOLLOW sets are computed, as shown below –

FIRST(S) = FIRST (AaAb) ∪ FIRST (BbBa)

FIRST(S) = {a} ∪ {b} = {a, b}

FIRST(A) = {∈}

FIRST(B) = {∈}

FOLLOW(S) = {$}

1. Using S → AaAb, we get –

FOLLOW(A) = FIRST (aAb) = {a}, and

FOLLOW(A) = FIRST (b) = {b}, Therefore,

FOLLOW(A) = {a, b}

2. Using S → BbBa, we get

FOLLOW(B) = FIRST (bBa) = {b}, and

FOLLOW(B) = FIRST (a) = {a}, Therefore,

FOLLOW(AB) = {a, b}

Table 2.19 Parsing Table

	a	b	$
S	S → AaAb	S → BbBa	
A	A → ∈	A → ∈	
B	B → ∈	B → ∈	

Prob.20. Construct the parsing table for the following grammar –

$S \rightarrow a\ XYb$

$X \rightarrow c/\epsilon$

$Y \rightarrow d/\epsilon$

Sol. In parsing table, first we calculate FIRST for the given grammar – ***(R.G.P.V., June 2009)***

FIRST (S) = FIRST (aXYb) = {a}

FIRST (X) = FIRST (C) ∪ FIRST (∈)

= {C} ∪ {∈}

= {c, ∈}

Prob.22. Construct the collection of LR(0) item sets and draw the goto graph for the following grammar –

$$S \rightarrow SS \mid a \mid \varepsilon$$

Indicate the conflicts (if any) in the various states of the SLR parser.

(R.G.P.V., Dec. 2008)

Sol. The augmented grammar will be –

$S' \rightarrow S$
$S \rightarrow SS$
$S \rightarrow a$
$S \rightarrow \varepsilon$

The canonical collection sets of LR(0) items are computed as follows –

I_0 = closure ({S' → .S}) = {S' → .S
S → .SS
S → .a
S → .
}

goto (I_0, S) = closure ({S' → S.})
= {S' → S.
S → S.S
S → .SS
S → .a
S → .
} = I_1

goto (I_0, a) = {S → a.} = I_2

goto (I_1, S) = {S → SS.
S → S.S
S → .SS
S → .a
S → .
} = I_3

goto (I_1, a) = {S → a.} = same as of I_2

goto (I_3, S) = {S → SS.
S → S.S
S → .SS
S → .a
S → .
} = same as of I_3

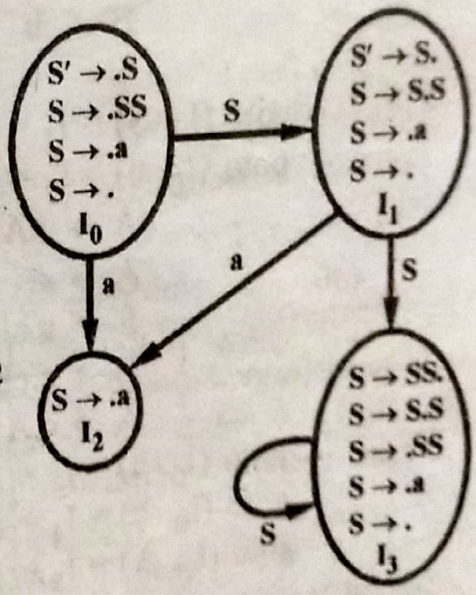


Fig. 2.22 The Goto Graph

Yes, there is a conflict in the I_1 state of the SLR parser

Prob.23. Construct LR(0) Parsing table for the following grammar –

$S \rightarrow cA/ccB$
$A \rightarrow cA/a$
$B \rightarrow ccB/b$

(R.G.P.V., June 2008)

Sol. The augmented grammar is –

S' → S
S → cA/ccB
A → cA/a
B → ccB/b

Items that can be generated using these productions are –

The canonical collections are –

I_0 = {S' → .S
S → .cA
S → .ccB
A → .cA
A → .a
B → .ccB
B → .b
}

goto (I_0, S) = I_1 = {S' → S.}

goto (I_0, c) = I_2 = {S → c.A
A → .cA
A → .a
S → c.cB
B → c.cB
A → c.A}

goto (I_0, a) = I_3 = {A → a.}

goto (I_0, b) = I_4 = {B → b.}

goto (I_2, A) = I_5 = {S → cA.
A → cA.}

goto (I_2, c) = I_6 = {A → c.A
A → .a
A → c.A
S → cc.B
A → .cA
B → cc.B}

goto (I_2, a) = {A → a.} similar to I_3

goto (I_6, A) = {A → cA.
A → cA.} similar to I_5

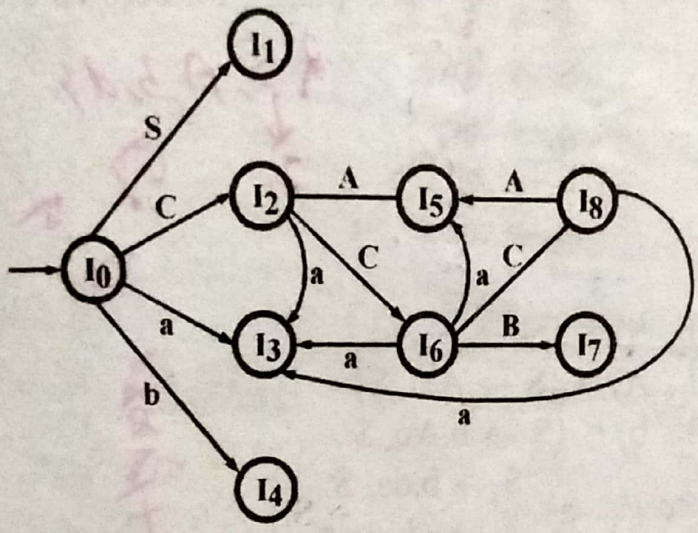
goto (I_6, a) = {A → a.}similar to I_3

goto $(I_6, B) = \{S \rightarrow ccB.\} = I_7$
$B \rightarrow ccB.$

goto $(I_6, c) = \{A \rightarrow c.A$
$A \rightarrow c.A$
$A \rightarrow c.A$
$A \rightarrow .a\} = I_8$

goto $(I_8, A) = \{A \rightarrow cA.$
$A \rightarrow cA.$
$A \rightarrow cA.\}$ similar to I_5

goto $(I_8, a) = \{A \rightarrow a.\}$ similar to I_3



GoTo Graph

Fig. 2.23

Table 2.22 LR(0) Parsing Table

State	Action			Goto			
	c	*a*	*b*	*$*	*S*	*A*	*B*
I_0	S_2	S_3/R_4	S_4/R_6		1		
I_1				accept			
I_2	S_6	S_3				5	
I_3							
I_4							
I_5							
I_6	S_8	S_3				5	7
I_7							
I_8		S_3				5	

Prob.24. Construct an LALR(1) parsing table for the following grammar–

$S \rightarrow Aa \mid bAc \mid dc \mid bda$
$A \rightarrow d.$

Sol. The augmented grammar is –

$S| \rightarrow S$
$S \rightarrow Aa$
$S \rightarrow bAc$
$S \rightarrow dc$
$S \rightarrow bda$
$A \rightarrow d$

The canonical collection of sets of LR(1) items is –

$I_0 = \{S| \rightarrow .S, \$$
$S \rightarrow .Aa, \$$
$S \rightarrow .bAc, \$$
$S \rightarrow .dc, \$$
$S \rightarrow .bda, \$$
$A \rightarrow .d, a$
$\}$

goto $(I_0, S) = \{S| \rightarrow S., \$\} = I_1$
goto $(I_0, A) = \{S \rightarrow A.a, \$\} = I_2$
goto $(I_0, b) = \{S \rightarrow b.Ac, \$$
$S \rightarrow b.da, \$$
$A \rightarrow .d, c$
$\} = I_3$
goto $(I_0, d) = \{S \rightarrow d.c, \$$
$A \rightarrow d., a$
$\} = I_4$
goto $(I_2, a) = \{S \rightarrow Aa., \$\} = I_5$
goto $(I_3, A) = \{S \rightarrow bA.c, \$\} = I_6$
goto $(I_3, d) = \{S \rightarrow bd.a, \$$
$A \rightarrow d., c$
$\} = I_7$
goto $(I_4, C) = \{S \rightarrow dc., \$) = I_8$
goto $(I_6, C) = \{S \rightarrow bAc., \$\} = I_9$
goto $(I_7, a) = \{S \rightarrow bda. \$\} = I_{10}$

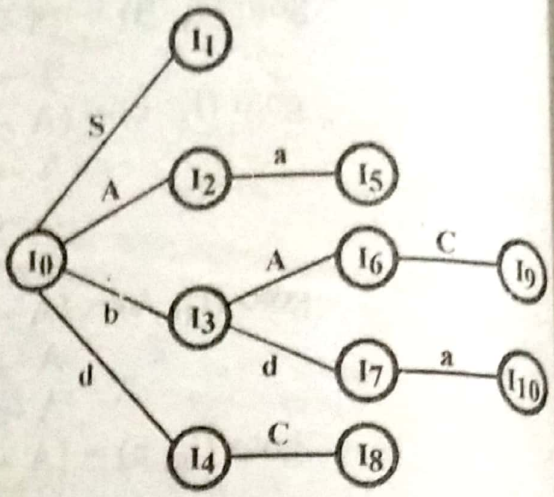


Fig. 2.24

There no sets of LR(1) items in the canonical collection that have identical *LR(0)*-part items and that differ only in their lookaheads. So, the LALR(1) parsing table for the above grammar is as shown in table 2.23.

The production of the grammar are numbered as shown below –

1. $S \rightarrow Aa$
2. $S \rightarrow bAc$
3. $S \rightarrow dc$
4. $S \rightarrow bda$
5. $A \rightarrow d$

Table 2.23 LALR(1) Parsing Table for Prob.24

	Action Table					GOTO Table	
	a	b	c	d	$	S	A
I_0		S_3		S_4		1	2
I_1					Accept		
I_2	S_5						
I_3				S_7		1	
I_4	R_5		S_8				
I_5					R_1		
I_6	S_{10}		S_9				
I_7			R_5				
I_8					R_3		
I_9					R_2		
I_{10}					R_4		

Prob.25. Show that the following grammar is LR(1) but not LALR (1).

$S \rightarrow Aa|bAc|Bc|bBa$

$A \rightarrow d$

$A \rightarrow d$ ***(R.G.P.V., Dec. 2008)***

Sol. The augmented grammar is –

$S| \rightarrow S$

$S \rightarrow Aa$

$S \rightarrow bAC$

$S \rightarrow BC$

$S \rightarrow bBa$

$A \rightarrow d$

$B \rightarrow d$

The canonical collection of sets of LR(1) items is –

I_0 { S1 →.S, $

S → .Aa, $

S → .bAc, $

S → .Bc, $

S → .bBa, $

A → .d, a

A → .d, a

A → .d, c

goto (I_0, S) = {S| → S., $} = I_1

goto (I_0, A) = {S → A.a, $} = I_2

goto (I_0, B) {S → B.c, $} = I_3

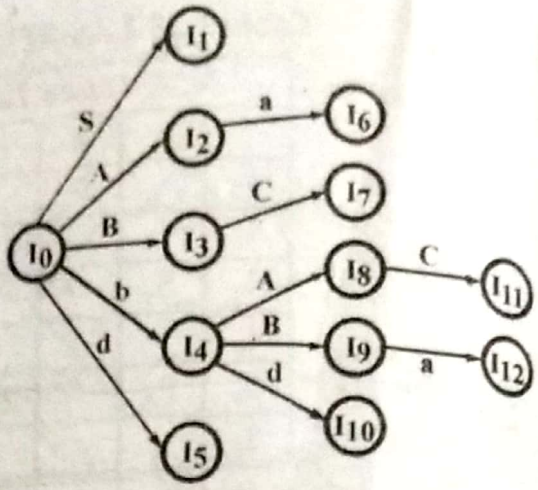
goto (I_0, b) = {S → b.Ac, $

A parser

$S \rightarrow b.Ba, \$$
$A \rightarrow .d, c$
$B \rightarrow .d, a$
$\} = I_4$

goto $(I_0, d) = \{A \rightarrow d., a$
$B \rightarrow d., c$
$\} = I_5$

goto $(I_2, a) = \{S \rightarrow Aa., \$\} = I_6$
goto $(I_3, c) = \{S \rightarrow Bc., \$\} = I_7$
goto $(I_4, A) = \{S \rightarrow bA.c, \$\} = I_8$
goto $(I_4, B) = \{S \rightarrow bB.a, \$\} = I_9$
goto $(I_4, d) = \{A \rightarrow d..c$
$B \rightarrow d., a$
$\} = I_{10}$
goto $(I_8, c) = \{S \rightarrow bAc., \$\} = I_{11}$
goto $(I_9, a) = \{S \rightarrow bBa., \$\} = I_{12}$



GoTo Graph

Fig. 2.25

There, no sets of LR(1) items in the canonical collection that have identical LR(0) part items and that differ only in their look a heads. So, the LALR (1) parsing table for the grammar is as shown in table 2.24.

Table 2.24

	Action Table					*GoTo Table*		
	a	b	c	d	$	S	A	B
I_0		S_4		S_5		1	2	3
I_1								
I_2	S_6							
I_3			S_7					
I_4				S_{10}			8	9
I_5	R_5/R_6		R_5/R_6					
I_6					R_1			
I_7					R_3			
I_8			S_{11}					
I_9	S_{12}							
I_{10}	R_5/R_6		R_5/R_6					
I_{11}					R_2			
I_{12}					R_4			

+++