UNIT –II

Syntax Analysis (Parser)

Topics:

1. Syntax specification of programming languages
2. Dealing with ambiguity of the grammar
3. Push Down Automata(PDA)
4. Design of top-down & bottom-up parsing technique
   Design of LL(1) parser.
   LR parsing-
   - Design of SLR
   - Design of CLR
   - Design of LALR parsers
5. Parser generator (yacc,bison)

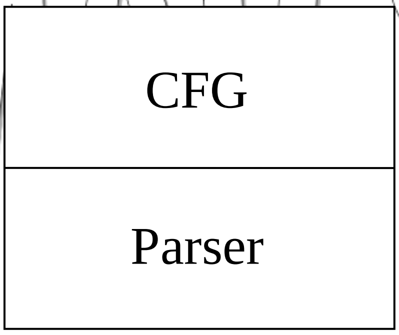
Introduction:

- Syntax analysis is the second phase of compiler
- In syntax analysis phase the compiler verifies whether or not the token generated by the lexical analysis is group according to syntatic rules of the language
  -if string is according to the syntax then it accepted as valid string otherwise error handler is called
- It is also called as Parser
- The output is a parse tree

Parser:

Program that takes token and grammar (CFG) as input and validate the input token according to grammar is called as Parser.
There are two issues involved in designing the syntax analysis phase of compiler

1. On valid construct of a programming language must be specifies. To do so we need grammar. The grammar which we are called (CFG).
2. A suitable recognizer is required to recognize whether a string of token generated by a lex is a valid construct or not (Parser).



1. Syntax Specification of Programming Languages:

➢ Context Free Grammar (CFG):

CFG specifies the CFL that consist of Non-terminals, terminals, production rules and start symbol. The context free grammar is defined as

G = (V, T, P, S)
Where,
V: finite set of non-terminals / Variables
T: finite set of terminals
S: starting symbol $S \in V$
P: production rules of form,
$$\alpha \rightarrow \beta$$
where $\alpha \in V$ and $\beta \in (V \cup T)^*$

In this grammar, every production rule is of the form:
$A \rightarrow \alpha$, where α , is arbitrary string of grammar symbol i.e. the left side of the production rule is a single non-terminal or variable and the right side can be any string of terminals and non- terminals.
Example: Let the grammar G defined as,

G = ({S}, {a, b}, P, S})

Where, P consists of following productions,

$S \rightarrow aSa \mid bSb \mid a \mid b$

➢ Parse tree / Derivation Tree:

The process of generating string from given grammar is called as derivation.

The derivation in a CFG can be represented using tree, such tree representing derivations are called derivation /parse trees. While deriving a string w from S ,if every derivation is considered to be step in the tree construction, then we get a graphical display of derivation of a string w as a tree ,called as derivation

The purpose of CFG is used to generate the string.

The CFG that are not linear, a derivation may involve sentential form with more than one variables/ non-terminals. In such cases, we have a choice in the order in which variables are replaced. So there are following two order of derivation –

1. Left most derivation (LMD)
2. Right most derivation (RMD

1. Left most derivation (LMD):

A derivation is said to be left most, if in each step the left most variable in the sentential form is replaced is called as left most derivation(LMD).

2. Right most derivation (RMD):

A derivation is said to be right most, if in each step the right most variable in the sentential form is replaced is called as right most derivation(RMD).

Example : Show the LMD and RMD for the given grammar

$$E \rightarrow E + E \,/ E * E \,/ id$$

$$string(w) = id + id * id$$

Ans:

1. Left most derivation (LMD):

$E \rightarrow E + E$

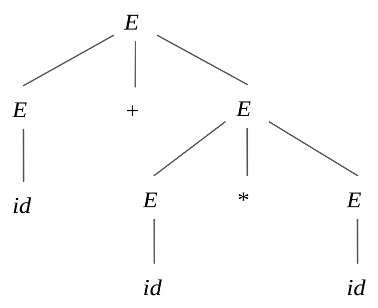
$E \rightarrow id + E$ $\{E \rightarrow id\}$

$E \rightarrow id + E * E$ $\{E \rightarrow E * E\}$

$E \rightarrow id + id * E$ $\{E \rightarrow id\}$

$E \rightarrow id + id * id$ $\{E \rightarrow id\}$

Derivation tree:



2. Right most derivation (RMD):

$E \rightarrow E + E$

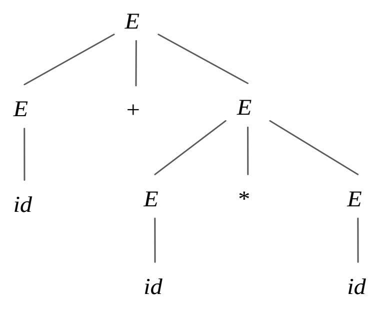
$E \rightarrow E + E * E$ $\{E \rightarrow E * E\}$

$E \rightarrow E + E * id$ $\{E \rightarrow id\}$

$E \rightarrow E + id * id$ $\{E \rightarrow id\}$

$E \rightarrow id + id * id$ $\{E \rightarrow id\}$

Derivation tree:



2. Dealing with Ambiguity of the Grammar:

➢ Ambiguous Grammar:

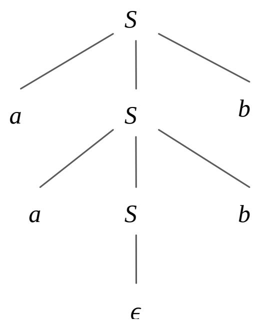
A grammar G is said to be ambiguous, if there exists at least one string in L(G) having more than one parse trees or derivation trees generating the string i.e. more than one LMD or RMD is called as ambiguous grammar.

To find ambiguous grammar the procedure is completely undecidable.
There are no direct algorithm to find ambiguity.
Parser does not allow ambiguous grammar.

Example: Check whether the grammar is ambiguous or not $S \to aSb\ /SS\ /\epsilon$
Ans:
Let string(w) = aabb
Show the derivations:
1. LMD
$S \to aSb$

$S \to aaSbb \qquad \{S \to aSb\}$

$S \to aabb \qquad \{S \to \epsilon\}$



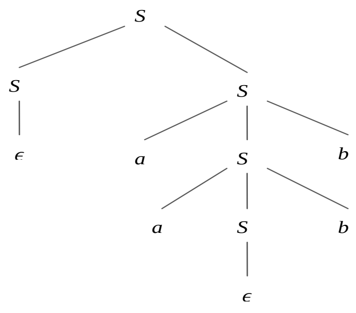
2. LMD:

$S \to SS$

$S \to S \qquad \{S \to \epsilon\}$

$S \to aSb \qquad \{ S \to aSb\}$

$S \to aabb \qquad \{ S \to aSb\}$



Conclusion: There exists more than one parse tree for the given grammar, hence the grammar is ambiguous.

➢ Unambiguous Grammar:

A grammar G is said to be unambiguous, if there any string in L(G) having only one parse trees or derivation trees generating the string is called as unambiguous grammar.

Left Recursion(LR):

A Production of form-

$A \rightarrow A\alpha \;/\beta$ is called as left recursion. Hence eliminate left recursion, by adding production rules, such that

$$A \rightarrow \beta A'$$
$$A' \rightarrow \alpha A' \;/\in$$

General rule: $A \rightarrow A\alpha_1 \;/A\alpha_2 \;\;/A\alpha_3 \;\;/\ldots/\;\beta_1 \;/\;\beta_2 \;/\beta_3 \;\ldots$

Hence eliminate left recursion, by adding production rules, such that

$$A \rightarrow \;\beta_1 A'/\;\beta_2\;A'/\beta_3 A'\;\;\ldots$$
$$A' \rightarrow \;\alpha_1 A'\;/\alpha_2 A'\;\;\;/\alpha_3\,A'\;\;\ldots/\;\epsilon$$

Left Factoring(LF):

A Production of form-

$A \rightarrow \alpha\;\beta/\alpha\gamma$ is called as Left Factoring. Hence eliminate LF, by adding production rules, such that

$$A \rightarrow \alpha A'$$
$$A' \rightarrow \beta\;/\gamma$$

General rule- $A \rightarrow \;\alpha\beta_1 \;/\;\alpha\beta_2 \;/\alpha\beta_3 \;\;\ldots/\gamma_1 \;/\gamma_2 \;\;/\gamma_3 \;\;/\ldots$

Hence eliminate left recursion, by adding production rules, such that

$$A \rightarrow \;\alpha A'/\gamma_1\;/\gamma_2\;\;/\gamma_3\;\;/\ldots$$
$$A' \rightarrow \;\beta_1\;/\;\beta_2\;/\beta_3\;\;\ldots$$

3. Push Down Automata (PDA):

The limitation of FA is that, it has limited amount of memory and it cannot accept the language which performs some mathematical operations like ($a^n b^n$ | n > 0).

Hence to eliminate this draw back a new types of automata call Push Down Automata (PDA) is used which contain FA and memory element (stack).

PDA = FA + STACK

A pushdown Automata is a finite automaton with a stack. A stack can contain any number

Of elements, but only the top element may be accessed.

Formal definition:

It is represented as 7 tuples

$$M = (Q, \Sigma, \delta, q_0, F, Z_0, \Gamma\,)$$

Where,

Q → Finite set of state

Σ → Finite set of input alphabet

δ → Transition function

q_0 → Initial state

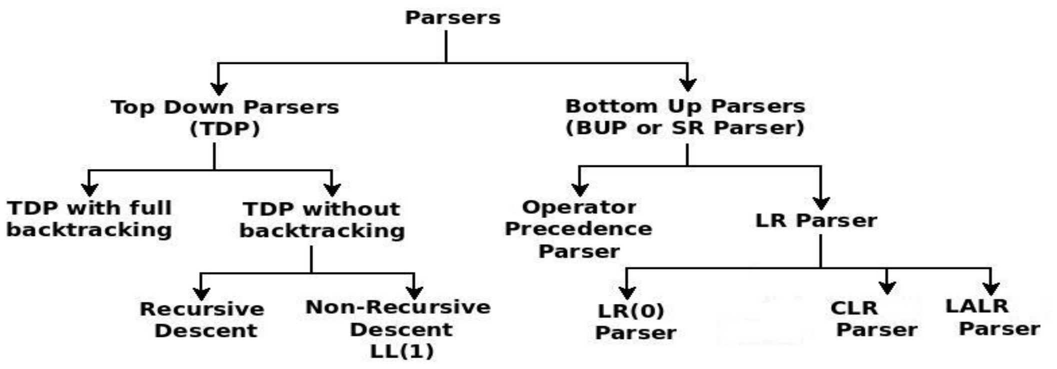
F → Set of final state

Z_0 → Bottom of stack

Γ → Stack alphabet

4. Design of top-down & bottom-up parsing technique:

Syntax analyzers follow production rules defined by the context-free grammar. The way production rules are implemented (derivation) divides parsing into two types shown in fig.

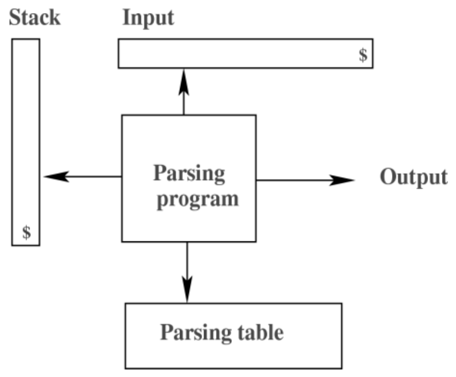

Parsers
Top Down Parsers (TDP)
Bottom Up Parsers (BUP or SR Parser)
TDP with full backtracking
TDP without backtracking
Operator Precedence Parser
LR Parser
Recursive Descent
Non-Recursive Descent LL(1)
LR(0) Parser
CLR Parser
LALR Parser

➢ 1. Top-Down Parser:

Top-down parsing is an attempt to drive string w from the start symbol of the grammar by constructing parse tree top down in the left most derivation (LMD) order.

LL (1) Parser:

An LL (1) Parser accepts LL grammar. LL grammar is a subset of context-free grammar but with some restrictions to get the simplified version, in order to achieve easy implementation. LL grammar can be implemented by means of both algorithms namely, recursive-descent or table-driven.
LL parser is denoted as LL (k). The first L in LL(k) is parsing the input from left to right, the second L in LL(k) stands for left-most derivation and k itself represents the number of look aheads. Generally k = 1, so LL (k) may also be written as LL(1). Shown in a fig.



LL(1) parsing can be performed using a pushdown stack, avoiding recursive calls.

- Initially the stack holds just the start symbol of the grammar.
- At each step a symbol *X* is popped from the stack:
  - if *X* is a terminal symbol then it is matched with *lookahead* and *lookahead* is advanced,
  - if *X* is a non-terminal, then using *look-ahead* and a *parsing table* (implementing the FIRST sets) a production is chosen and its right hand side is pushed onto the stack.
- This process goes on until the stack and the input string become empty. It is useful to have an *end_of_stack* and an *end_of_input* symbols. We denote them both by $.

To implement LL(1) Parser required following functions-

- FIRST( ): The process of finding all first terminal symbols
- FOLLOW ( ): The process of finding all immediately next symbols

RULES FOR CALCULATION OF FIRST AND FOLLOW SET:

FIRST SET:

1. $First(\,terminal) = \{\,terminal\,\}$

2. $if\ A \to a\beta\ (\,where\ a \in T\ and\ \beta \in (V \cup T)^*\,)$

   $then\ \ First(A) = \{\,a\,\}$

3. $if\ A \to B\alpha\ (\,where\ B \neq \epsilon\,)$

   $then\ \ First(A) = First(B)$

4. $if\ A \to B\alpha\ (\,where\ B = \epsilon\,)$

   $then\ \ First(A) = First(B) - \epsilon \cup First(\alpha)$

FOLLOW SET:

1. $Follow(\,start) = \{\ \$\ \}$

2. $if\ A \to \alpha B$

   $then\ \ Follow(B) = Follow(A)$

3. $if\ A \to \alpha B\beta\ (\,where\ B \neq \epsilon\,)$

   $then\ \ Follow(B) = First(\beta)$

4. $if\ A \to \alpha B\beta\ (\,where\ B = \epsilon\,)$

   $then\ \ Follow(B) = First(\beta) - \epsilon \cup Follow(A)$

NOTE: Follow set never contain ϵ

Design of LL (1) Parser:

To design of LL (1) Parser follow the following procedure-

1. Check whether the given grammar contains LR or LF then eliminated first.

   Note: Before eliminating LF, LR should be eliminated.
2. Calculate First and Follow set
3. Construct LL(1) Parsing Table:

   If the table contains multiple entries in a single cell then it is not LL (1) otherwise LL(1).

Example: Check whether the following grammar is LL(1) or not ?

$S \rightarrow AaAb/\ BbBa$
$A \rightarrow \epsilon$
$A \rightarrow \epsilon$

Ans:

1. There is no LR or LF
2. Calculation of First and Follow set-
   - **First set:**

     $First(B) = \{\epsilon\}$
     $First(A) = \{\epsilon\}$
     $First(S) = \{a, b\}$
   - **Follow set:**

     $First(S) = \{\$\}$
     $First(A) = \{a\ , b\}$
     $First(A) = \{a, b\}$

3. **Parsing Table:**

T / V	a	b	$\$$
S	$S \rightarrow AaAb$	$S \rightarrow BbBa$	
A	$A \rightarrow \epsilon$	$A \rightarrow \epsilon$	
B	$A \rightarrow \epsilon$	$A \rightarrow \epsilon$	

Since table does not contains multiple entries hence it is LL(1).

2. Bottom -Up Parser:

Bottom up parsing is an attempt of reducing string w to the start symbol of grammar by constructing parse tree bottom up in the rivers RMD order.
There are following types-

1. Simple Left to Right (SLR) / LR (0) Parser:

To design LR (0) Parser follow the following procedure-

1. Augmented Grammar
2. Calculation of First and Follow set
3. Transition diagram
4. **Parsing table:**
   If table contains Shift(S) / Reduced(R) or Reduced(R) / Reduced(R) conflict then it is not LR(0) otherwise LR(0).

Example: Check whether the following grammar is LR(0) or not ?

$$\boldsymbol{S \rightarrow 1S1 / \ 0S0 \ / \ 10}$$

ANS:

1. Augmented Grammar:

$S' \rightarrow .S \qquad - 0$
$S \rightarrow .1S1 \qquad - 1$
$S \rightarrow .0S0 \qquad - 2$
$S \rightarrow .10 \qquad - 3$

2. Calculation of First and Follow set:

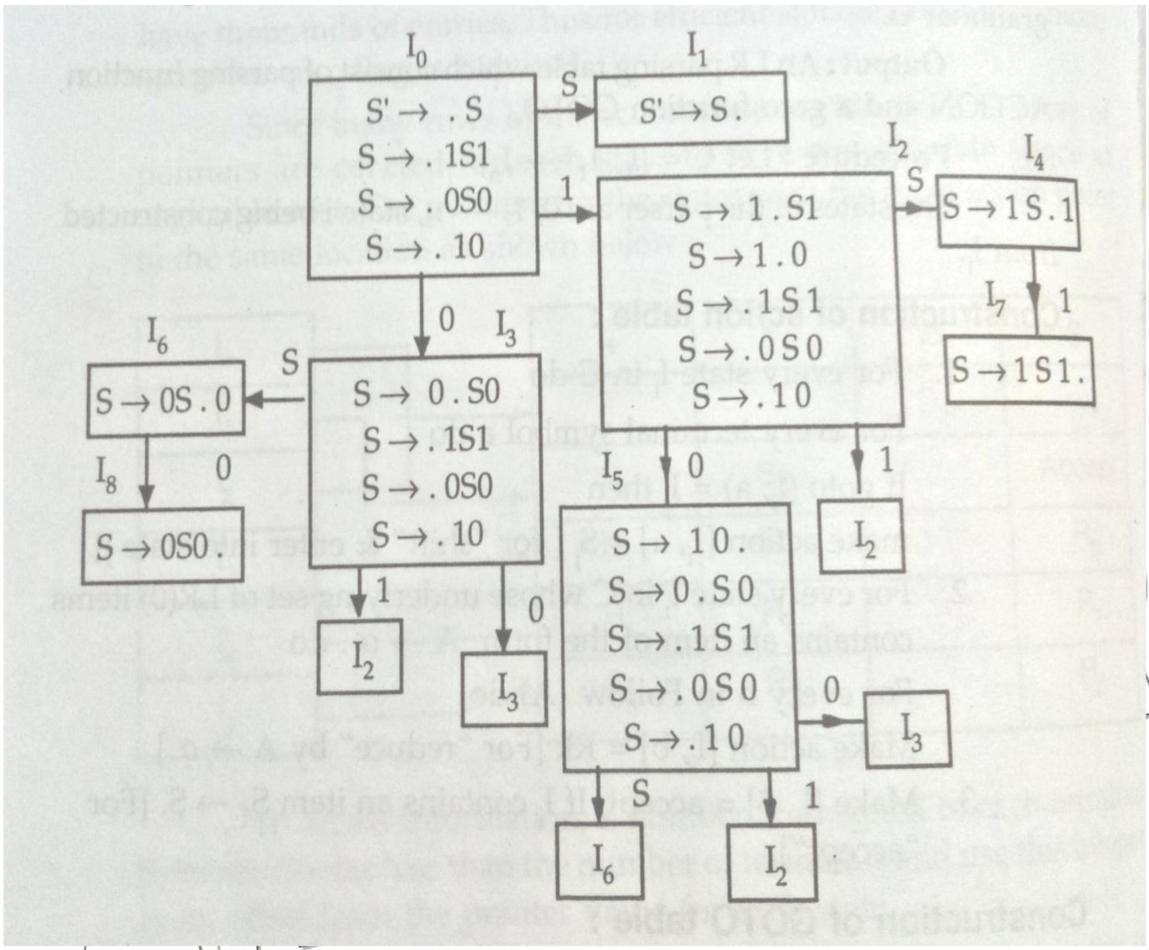
First set:

First $(S) = \{0,1\}$

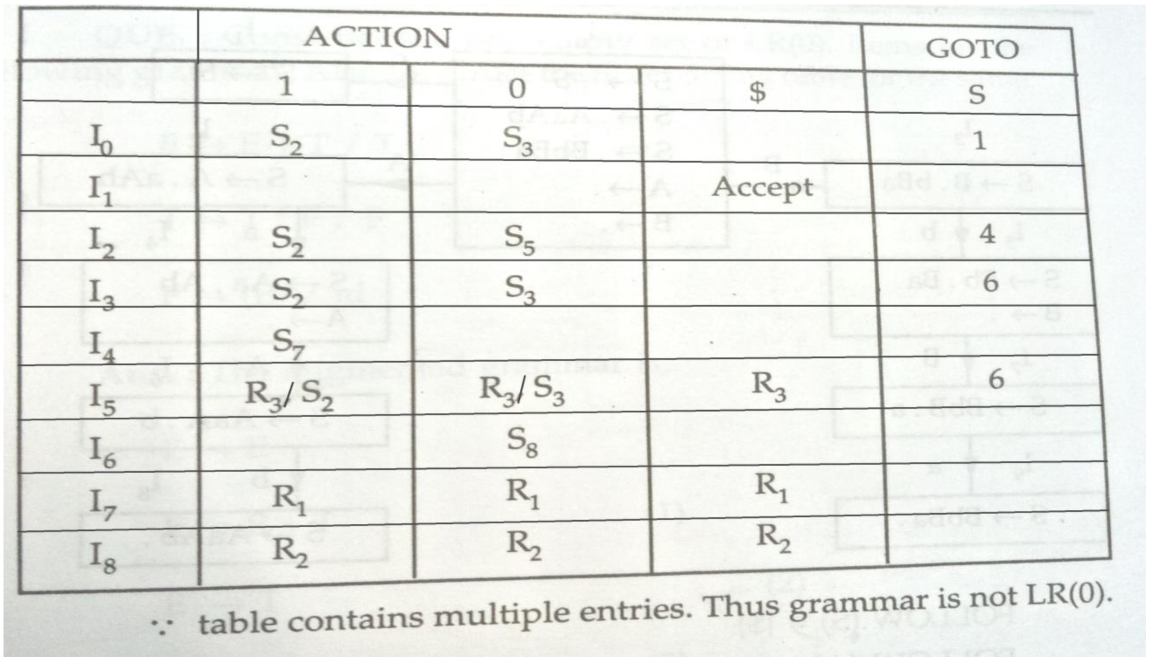
Follow set:

Follow $(S) = \{0,1,\$\}$

3. Transition Diagram:



4. Parsing table:



	ACTION			GOTO
	1	0	$	S
I_0	S_2	S_3		1
I_1			Accept	
I_2	S_2	S_5		4
I_3	S_2	S_3		6
I_4	S_7			
I_5	R_3/S_2	R_3/S_3	R_3	6
I_6		S_8		
I_7	R_1	R_1	R_1	
I_8	R_2	R_2	R_2	

∵ table contains multiple entries. Thus grammar is not LR(0).

2. Canonical Left to Right (CLR) / LR (1) Parser:

1. Augmented Grammar
2. Calculation of First set
3. **Transition diagram**
   LR(1) = LR(0) , Look Ahead(LA)
   Rules for calculate LA-
   - LA(S') = $
   - If $A \rightarrow \alpha.B\beta\ , a$ Then LA(B) = $First(\beta, a)$
4. **Parsing table:**
   If table contains Shift(S) / Reduced(R) or Reduced(R) / Reduced(R) conflict then it is not LR(0) otherwise LR(0).

Example: Check whether the following grammar is LR (1) or not ?

$\boldsymbol{S \rightarrow AaAb / BbBa}$
$\boldsymbol{A \rightarrow \epsilon}$
$\boldsymbol{A \rightarrow \epsilon}$

ANS:

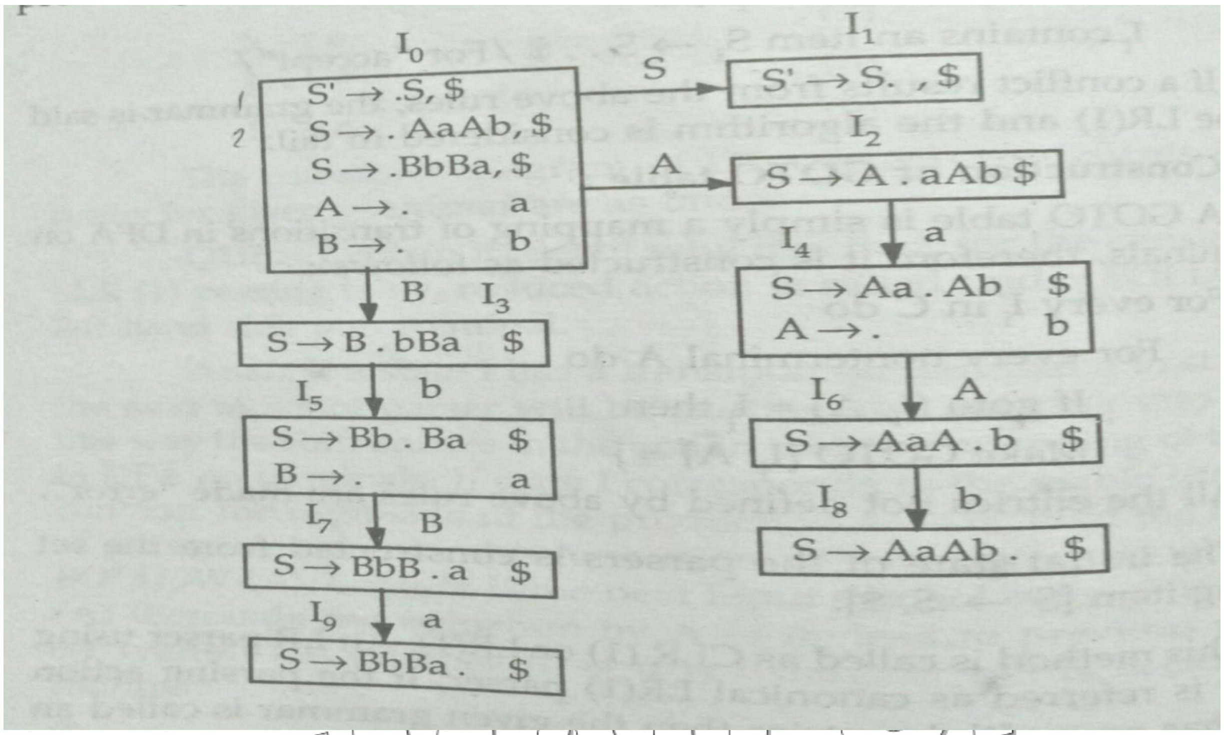
1. **Augumented Grammar:**

$S' \rightarrow .S$ $\quad - 0$
$S \rightarrow .AaAb$ $\quad - 1$
$S \rightarrow .BbBa$ $\quad - 2$
$A \rightarrow .\epsilon$ $\quad - 3$
$B \rightarrow .\epsilon$ $\quad - 4$

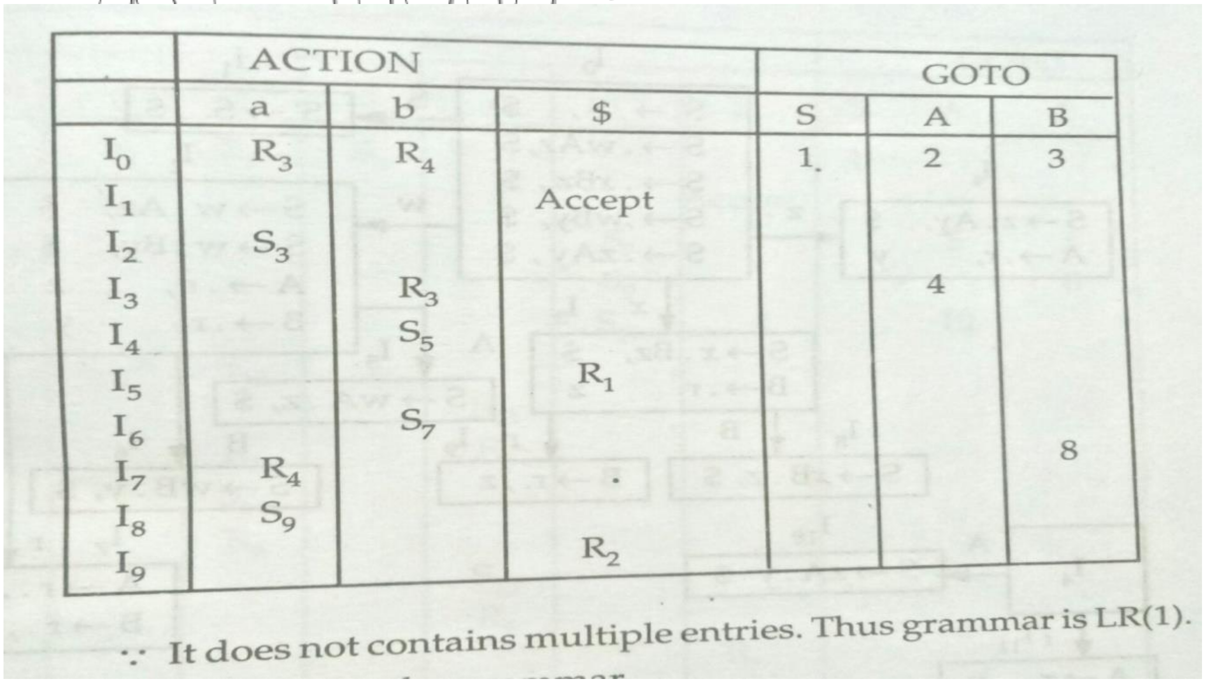
2. **Calculation of First and Follow set:**
   - **First set:**

$First(B) = \{\ \epsilon\ \}$
$First(A) = \{\ \epsilon\ \}$
$First(S) = \{\ a, b\ \}$

3. Transition Diagram:



4. Parsing Table:



	ACTION			GOTO		
	a	b	$	S	A	B
I_0	R_3	R_4		1.	2	3
I_1			Accept			
I_2	S_3					
I_3		R_3			4	
I_4		S_5				
I_5			R_1			
I_6		S_7				8
I_7	R_4					
I_8	S_9					
I_9			R_2			

∵ It does not contains multiple entries. Thus grammar is LR(1).

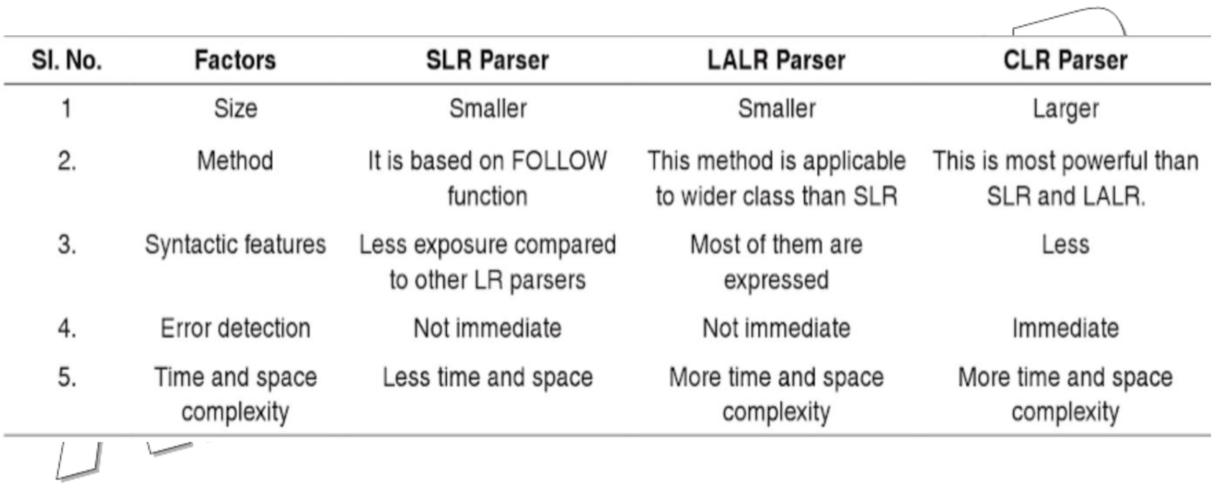
3. Look Ahead Left to Right (LALR) Parser:

To design of LALR parser follow the following steps-

1. **Design LR (1) Parser-**
   - Augmented Grammar
   - Calculation of First set
   - Transition diagram
   - From constructed states identifying those states whose production rules are same but different look ahead then merge those states.

2. **Design LALR Parser-**
   - Transition diagram
   - Parsing table

Comparison between LR (0) ,LR(1) and LALR Parser:



Sl. No.	Factors	SLR Parser	LALR Parser	CLR Parser
1	Size	Smaller	Smaller	Larger
2.	Method	It is based on FOLLOW function	This method is applicable to wider class than SLR	This is most powerful than SLR and LALR.
3.	Syntactic features	Less exposure compared to other LR parsers	Most of them are expressed	Less
4.	Error detection	Not immediate	Not immediate	Immediate
5.	Time and space complexity	Less time and space	More time and space complexity	More time and space complexity

Shift Reduce Technique / Stack Implementation:

Shift reduce technique is used to implement bottom up parsing. In this technique, a Parse goes to shifting the input symbol onto the stack till a handle comes on the top of the stack. When a handle appears on the top of the stack is performs reduction.

Here the actions are either shifting of the next input symbols onto the stack or performing reduction hence it is referred as shift reduce technique.

Handle:

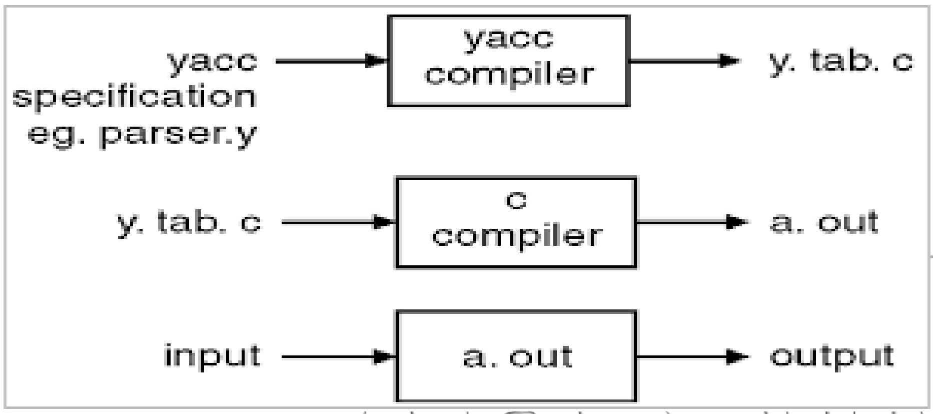
Handle is a substring of a string that matches the right side of a production, and whose reduction to the non terminal on the left side of the production represents one step along the rivers of rightmost derivation.

Viable prefix:

A viable prefix of a right sentential form is prefix that contains a handle but no symbol to the right of the handle.

5. Parser generator (Yacc, Bison) :

YACC is a parser generator that takes an input file with an attribute-enriched BNF grammar specification, and generates the output C file ***y.tab.c*** containing the function ***int yyparse(void)*** that implements its table-driven LALR(1) parser. This function automatically invokes ***yylex()*** every time it needs a token to continue parsing. Such function can be provided by the user or by a ***lex*** -generated scanner. Shown in fig.



Section definitions contains token declarations, the value-stack type, the type declaration of attributes associated to non-terminals and tokens, and the precedences and associativity of tokens.
%{%} are allowed only in definitions. Useful to declare prototypes of routines used in the user routines section, for header file inclusion, for declaring of global variables, etc.

The rules section contains the BNF grammar specification together with the semantic actions for each production rule. It is actually a syntax-directed translation scheme. It is customary to write tokens in uppercase and non-terminals in lower case.The grammar is automatically checked by yacc to assure that:
1. There are rules (productions) for all non-terminals.
2. All non-terminals must be reachable from the axiom.
3. The grammar is not ambiguous
4. The grammar is LALR(1) parsable.
The yacc program alerts us if any of these conditions do not hold. The first two are trivial to arrange.