1. Role of Parser.

- In our compiler model, the parser obtains a string of tokens from lexical analyzer, as shown in fig. 3.1.1.

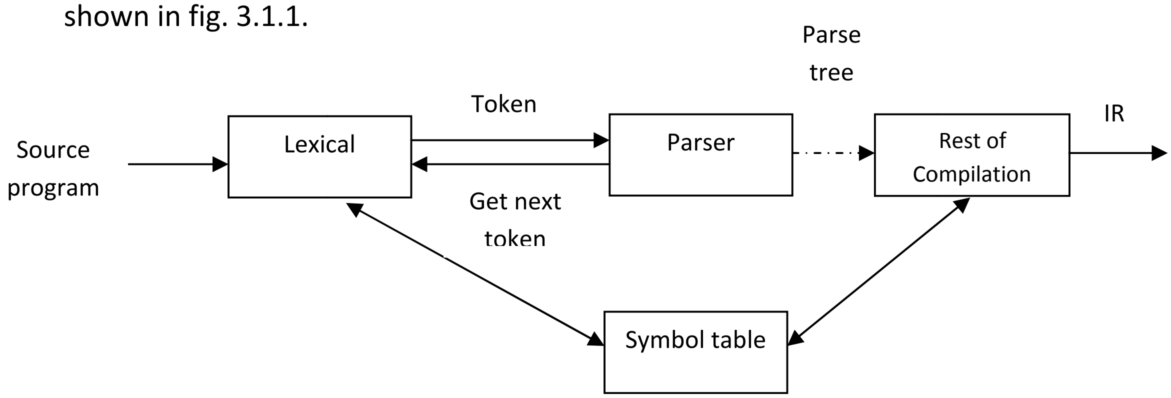


Fig.3.1.1. Position of parser in compiler model

- We expect the parser to report any syntax error. It should also recover from commonly occurring errors.
- The methods commonly used for parsing are classified as a top down or bottom up parsing.
- In top down parsing parser, build parse tree from top to bottom, while bottom up parser starts from leaves and work up to the root.
- In both the cases, the input to the parser is scanned from left to right, one symbol at a time.
- We assume the output of parser is some representation of the parse tree for the stream of tokens produced by the lexical analyzer.

2. Difference between syntax tree and Parse tree.

Syntax tree v/s Parse tree

No.	Parse Tree	Syntax Tree
1	Interior nodes are non-terminals, leaves are terminals.	Interior nodes are "operators", leaves are operands.
2	Rarely constructed as a data structure.	When representing a program in a tree structure usually use a syntax tree.
3	Represents the concrete syntax of a program.	Represents the abstract syntax of a program (the semantics).

Table 3.1.1. Difference between syntax tree & Parse tree

- Example: Consider grammar following grammar,
  $E \rightarrow E + E$
  $E \rightarrow E * E$
  $E \rightarrow Id$
- Figure 3.1.2. Shows the syntax tree and parse tree for string id + id*id.

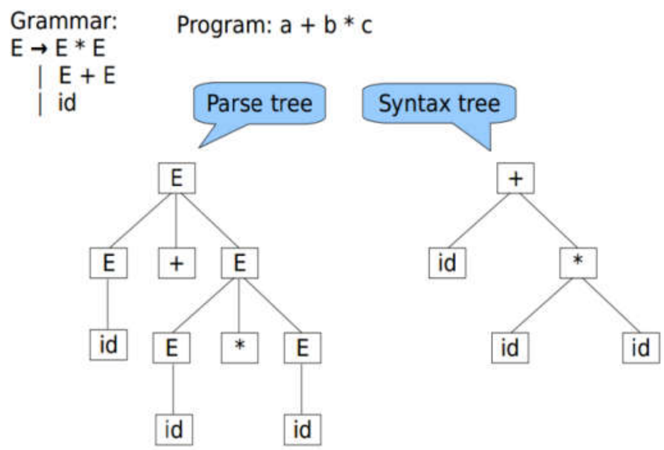


Fig.3.1.2. Syntax tree and parse tree

3. Types of Derivations. (Leftmost & Rightmost)

There are mainly two types of derivations,

1. Leftmost derivation
2. Rightmost derivation

Let Consider the grammar with the production S→S+S | S-S | S*S | S/S |(S)| a

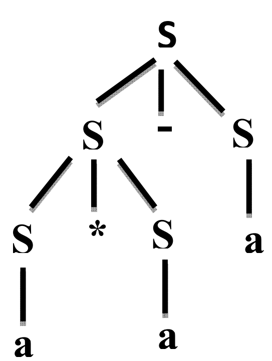
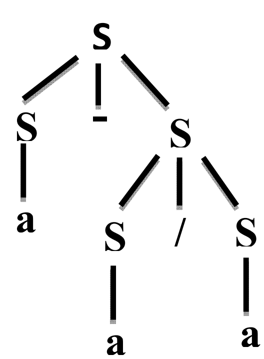
Left Most Derivation	Right Most Derivation
A derivation of a string W in a grammar G is a left most derivation if at every step the left most non terminal is replaced.	A derivation of a string W in a grammar G is a right most derivation if at every step the right most non terminal is replaced.
Consider string a*a-a S→S-S S*S-S a*S-S a*a-S a*a-a	Consider string: a-a/a S→S-S S-S/S S-S/a S-a/a a-a/a
Equivalent left most derivation tree 	Equivalent Right most derivation tree 

Table 3.1.2. Difference between Left most Derivation & Right most Derivation

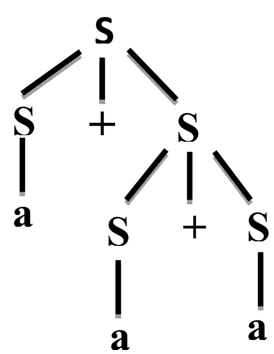
4. Explain Ambiguity with example.

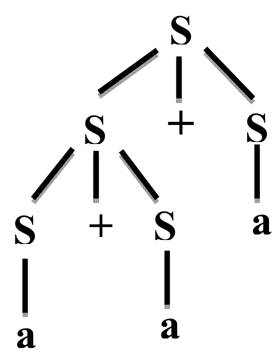
- An ambiguous grammar is one that produces more than one leftmost or more than one rightmost derivation for the same sentence.

1) Prove that given grammar is ambiguous. S→S+S / S-S / S*S / S/S /(S)/a (IMP)

String : a+a+a

S→S+S	S→S+S
a+S	S+S+S
a+S+S	a+S+S
a+a+S	a+a+S
a+a+a	a+a+a





- Here we have two left most derivation hence, proved that above grammar is ambiguous.

2) Prove that S->a | Sa | bSS | SSb | SbS is ambiguous

String: baaab

S→bSS	S→SSb
baS	bSSSb
baSSb	baSSb
baaSb	baaSb
baaab	baaab

- Here we have two left most derivation hence, proved that above grammar is ambiguous.

5. Elimination of left recursion.

- A grammar is said to be left recursive if it has a non terminal A such that there is a derivation A→Aα for some string α.
- Top down parsing methods cannot handle left recursive grammar, so a transformation that eliminates left recursion is needed.

Algorithm to eliminate left recursion

1. *Assign an ordering $A_1,\dots,A_n$ to the nonterminals of the grammar.*
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 productions $A_i \to \delta_1 \gamma \mid \delta_2 \gamma \mid ..... \mid \delta_k \gamma$
where $Aj \to \delta_1 \mid \delta_2 \mid ..... \mid \delta_k$ are all the current Aj productions;
end
eliminate the intermediate left recursion among the A_i-productions
end

- Example 1 : Consider the following grammar,
  E→E+T/T
  T→T*F/F
  F→(E)/id
  Eliminate immediate left recursion from above grammar then we obtain,
  E→TE'
  E'→+TE' | ε
  T→FT'
  T'→*FT' | ε
  F→(E) | id
- Example 2 : Consider the following grammar,
  S→Aa | b
  A→Ac | Sd | ε
  Here, non terminal S is left recursive because S→Aa→Sda, but it is not immediately left recursive.
  S→ Aa | b
  A→Ac | Aad | bd | ε
  Now, remove left recursion
  S→ Aa | b
  A→ bdA' | A'
  A→ cA' | adA' | ε

6. Left factoring.

- Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parsing.
  Algorithm to left factor a grammar
  Input: Grammar G
  Output: An equivalent left factored grammar.
  1. *For each non terminal A find the longest prefix α common to two or more of its alternatives.*
  2. *If α!= E, i.e., there is a non trivial common prefix, replace all the A productions $A \to \alpha\beta_1 \mid \alpha\beta_2 \mid .... \mid \alpha\beta_n \mid \gamma$ where γ represents all alternatives that do not begin with α by*
     $A \Rightarrow \alpha A' \mid \gamma$
     $A' \Rightarrow \beta_1 \mid \beta_2 \mid .... \mid \beta_n$

  Here A' is new non terminal. Repeatedly apply this transformation until no two alternatives for a non-terminal have a common prefix.

- EX1: Perform left factoring on following grammar,
  A→ xByA | xByAzA | a
  B→b
  Left factored, the grammar becomes
  A→ xByAA' | a
  A'→zA | Є
  B→ b
- EX2: Perform left factoring on following grammar,
  S→iEtS | iEtSeS | a
  E→b
  Left factored, the grammar becomes
  S→ iEtSS' | a
  S'→eS | Є
  E→b

7. Types of Parsing.

- Parsing or syntactic analysis is the process of analyzing a string of symbols according to the rules of a formal grammar.
- Parsing is a technique that takes input string and produces output either a parse tree if string is valid sentence of grammar, or an error message indicating that string is not a valid sentence of given grammar. Types of parsing are,
  1. **Top down parsing**: In top down parsing parser build parse tree from top to bottom.
  2. **Bottom up parsing**: While bottom up parser starts from leaves and work up to the root.

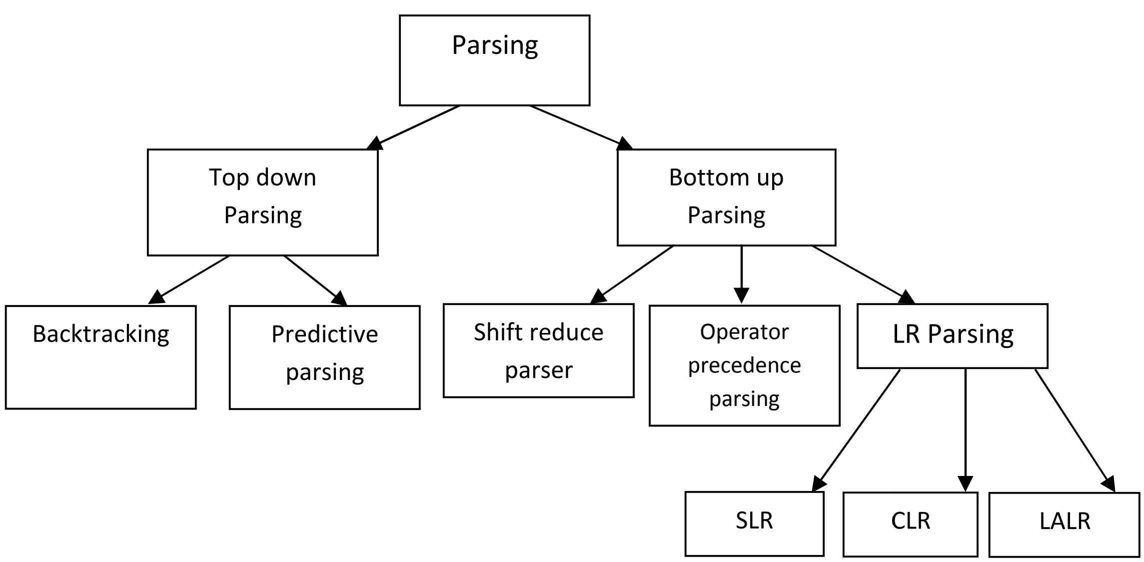


Fig.3.1.3 Parsing Techniques

8. Recursive Decent Parsing.

- A top down parsing that executes a set of recursive procedure to process the input without backtracking is called recursive decent parser.
- There is a procedure for each non terminal in the grammar.
- Consider RHS of any production rule as definition of the procedure.
- As it reads expected input symbol, it advances input pointer to next position.

Example:

E→ T {+T}*
T→ V{*V}*
V→ id

```
Procedure proc_E: (tree_root);
        var
                a, b : pointer to a tree node;
        begin
                proc_T(a);
                While (nextsymb = '+') do
                        nextsymb = next source symbol;
                        proc_T(b);
                        a= Treebuild ('+', a, b);
                tree_root= a;
                return;
end proc_E;
Procedure proc_T: (tree_root);
        var
                a, b : pointer to a tree node;
        begin
                proc_V(a);
                While (nextsymb = '*') do
                        nextsymb = next source symbol;
                        proc_V(b);
                        a= Treebuild ('*', a, b);
                tree_root= a;
                return;
end proc_T;
Procedure proc_V: (tree_root);
        var
                a : pointer to a tree node;
        begin
                If (nextsymb = 'id') then
                        nextsymb = next source symbol;
                        tree_root= tree_build(id, , );
                else print "Error";
```

return;

end proc_V;

Advantages

- It is exceptionally simple.
- It can be constructed from recognizers simply by doing some extra work.

Disadvantages

- It is time consuming method.
- It is difficult to provide good error messages.

9. Predictive parsing. OR LL(1) Parsing.

- This top-down parsing is non-recursive. LL (1) – the first L indicates input is scanned from left to right. The second L means it uses leftmost derivation for input string and 1 means it uses only input symbol to predict the parsing process.
- The block diagram for LL(1) parser is given below,

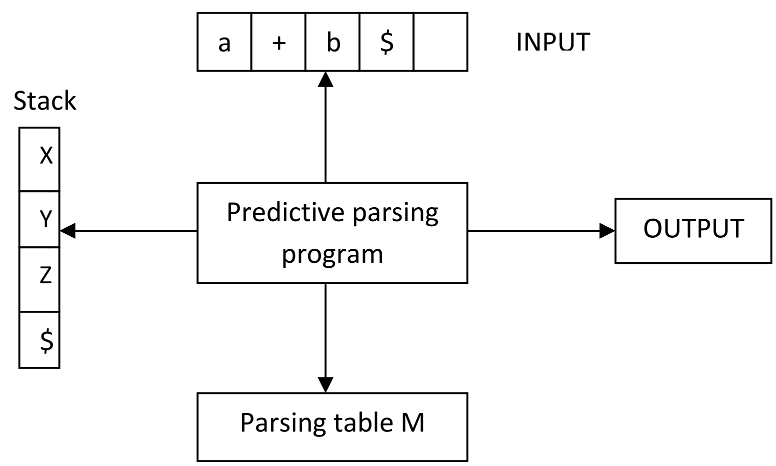


Fig.3.1.4 Model of nonrecursive predictive parser

- The data structure used by LL(1) parser are input buffer, stack and parsing table.
- The parser works as follows,
- The parsing program reads top of the stack and a current input symbol. With the help of these two symbols parsing action can be determined.
- The parser consult the table M[A, a] each time while taking the parsing actions hence this type of parsing method is also called table driven parsing method.
- The input is successfully parsed if the parser reaches the halting configuration. When the stack is empty and next token is $ then it corresponds to successful parsing.

Steps to construct LL(1) parser

1. Remove left recursion / Perform left factoring.
2. Compute FIRST and FOLLOW of nonterminals.
3. Construct predictive parsing table.
4. Parse the input string with the help of parsing table.

Example:

E→E+T/T
T→T*F/F
F→(E)/id

Step1: Remove left recursion

E→TE'
E'→+TE' | ϵ
T→FT'
T'→*FT' | ϵ
F→(E) | id

Step2: Compute FIRST & FOLLOW

	FIRST	**FOLLOW**
E	{(,id}	{$,)}
E'	{+,ϵ}	{$,)}
T	{(,id}	{+,$,)}
T'	{*,ϵ}	{+,$,)}
F	{(,id}	{*,+,$,)}

Table 3.1.3 first & follow set

Step3: Predictive Parsing Table

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ϵ	E'→ϵ
T	T→FT'			T→FT'		
T'		T'→ϵ	T'→*FT'		T'→ϵ	T'→ϵ
F	F→id			F→(E)		

Table 3.1.4 predictive parsing table

Step4: Parse the string

Stack	Input	Action
$E	id+id*id$	
$E'T	id+id*id$	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'→ ϵ
$ 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'→ ϵ
$	$	E'→ ϵ

Table 3.1.5. moves made by predictive parse

10. Error recovery in predictive parsing.

- Panic mode error recovery is based on the idea of skipping symbols on the input until a token in a selected set of synchronizing token appears.
- Its effectiveness depends on the choice of synchronizing set.
- Some heuristics are as follows:
  - ✓ Insert 'synch' in FOLLOW symbol for all non terminals. 'synch' indicates resume the parsing. If entry is "synch" then non terminal on the top of the stack is popped in an attempt to resume parsing.
  - ✓ If we add symbol in FIRST (A) to the synchronizing set for a non terminal A, then it may be possible to resume parsing if a symbol in FIRST(A) appears in the input.
  - ✓ If a non terminal can generate the empty string, then the production deriving the ε can be used as a default.
  - ✓ If parser looks entry M[A,a] and finds that it is blank then i/p symbol a is skipped.
  - ✓ If a token on top of the stack does not match i/p symbol then we pop token from the stack.
- Consider the grammar given below:

E ::= TE'
E' ::= +TE' | ε
T ::= FT'
T' ::= *FT' | ε
F ::= (E)|id

  - ✓ Insert 'synch' in FOLLOW symbol for all non terminals.

	FOLLOW
E	{$,)}
E'	{$,)}
T	{+,$,)}
T'	{+,$,)}
F	{+,*,$,)}

Table 3.1.6. Follow set of non terminals

NT	**Input Symbol**					
	id	**+**	*****	**(**	**)**	**$**
E	E =>TE'			E=>TE'	synch	Synch
E'		E' => +TE'			E' => ε	E' => ε
T	T => FT'	synch		T=>FT'	Synch	synch
T'		T' => ε	T' =>* FT'		T' => ε	T' => ε
F	F => <id>	synch	Synch	F=>(E)	synch	synch

Table 3.1.7. Synchronizing token added to parsing table

Stack	Input	Remarks
$E	)id*+id$	Error, skip)
$E	id*+id$	
$E' T	id*+id$	
$E' T' F	id*+id$	
$E' T' id	id*+id$	
$E' T'	*+id$	
$E' T' F*	*+id$	
$E' T' F	+id$	Error, M[F,+]=synch
$E' T'	+id$	F has been popped.
$E'	+id$	
$E' T+	+id$	
$E' T	id$	
$E' T' F	id$	
$E' T' id	id$	
$E' T'	$	
$E'	$	
$	$	

Table 3.1.8. Parsing and error recovery moves made by predictive parser

11. Explain Handle and handle pruning.

Handle: A "handle" of a string is a substring of the string that matches the right side of a production, and whose reduction to the non terminal of the production is one step along the reverse of rightmost derivation.

Handle pruning: The process of discovering a handle and reducing it to appropriate Left hand side non terminal is known as handle pruning.

Right sentential form	Handle	Reducing production
id1+id2*id3	id1	E→id
E+id2*id3	id2	E→id
E+E*id3	id3	E→id
E+E*E	E*E	E→ E*E
E+E	E+E	E→ E+E
E		

Table 3.1.9. Handles

12. Shift reduce Parsing.

- The shift reduce parser performs following basic operations,
- Shift: Moving of the symbols from input buffer onto the stack, this action is called shift.
- Reduce: If handle appears on the top of the stack then reduction of it by appropriate rule is done. This action is called reduce action.
- Accept: If stack contains start symbol only and input buffer is empty at the same time then that action is called accept.

- Error: A situation in which parser cannot either shift or reduce the symbols, it cannot even perform accept action then it is called error action.
  Example: Consider the following grammar,
  E→E + T | T
  T→T * F | F
  F→id
  Perform shift reduce parsing for string id + id * id.

Stack	Input buffer	Action
$	id+id*id$	Shift
$id	+id*id$	Reduce F->id
$F	+id*id$	Reduce T->F
$T	+id*id$	Reduce E->T
$E	+id*id$	Shift
$E+	id*id$	shift
$E+ id	*id$	Reduce F->id
$E+F	*id$	Reduce T->F
$E+T	*id$	Shift
$E+T*	id$	Shift
$E+T*id	$	Reduce F->id
$E+T*F	$	Reduce T->T*F
$E+T	$	Reduce E->E+T
$E	$	Accept

Table 3.1.10. Configuration of shift reduce parser on input id + id*id

13. Operator Precedence Parsing.

- **Operator Grammar**: A Grammar in which there is no Є in RHS of any production or no adjacent non terminals is called operator precedence grammar.
- In operator precedence parsing, we define three disjoint precedence relations <·**,** ·>and = between certain pair of terminals.

Relation	Meaning
a <·b	a "yields precedence to" b
a=·b	a "has the same precedence as" b
a·>b	a "takes precedence over" b

Table 3.1.11. Precedence between terminal a & b

Leading:-

Leading of a nonterminal is the first terminal or operator in production of that nonterminal.

Trailing:-

Traling of a nonterminal is the last terminal or operator in production of that nonterminal

Example:

E→E+T/T

T→T*F/F
F→id

Step-1: Find leading and trailing of NT.

Leading	Trailing
(E)={+,*,id}	(E)={+,*,id}
(T)={*,id}	(T)={*,id}
(F)={id}	(F)={id}

Step-2:Establish Relation

1. a <· b
   Op · NT → Op <· Leading(NT)
   +T   + <· {*,id}
   *F   * <· {id}
2. a ·> b
   NT · Op →Traling(NT) ·> Op
   E+   {+,*, id} ·> +
   T*   {*, id} ·> *
3. $ <· {+, *,id}
4. {+,*,id} ·> $

Step-3: Creation of table

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

Table 3.1.12. precedence table

Step-4: Parsing of the string using precedence table.

We will follow following steps to parse the given string,

1. Scan the input string until first ·> is encountered.
2. Scan backward until <· is encountered.
3. The handle is string between <· And ·>.

$ <· Id ·> + <· Id ·> * <· Id ·> $	Handle id is obtained between <· ·> Reduce this by F->id
$ F+ <· Id ·> * <· Id ·> $	Handle id is obtained between <· ·> Reduce this by F->id
$ F + F * <· Id ·> $	Handle id is obtained between <· ·> Reduce this by F->id
$ F + F * F $	Perform appropriate reductions of all non terminals.
$ E + T * F$	Remove all non terminal
$ +* $	Place relation between operators
$ <· + <· * >$	The * operator is surrounded by <· ·>. This

	indicates * becomes handle we have to reduce T*F.
$ <· + ·>$	+ becomes handle. Hence reduce E+T.
$ $	Parsing Done

Table 3.1.13. moves made by operator precedence parser

Operator Precedence Function

Algorithm for Constructing Precedence Functions

1. *Create functions f_a and g_a for each a that is terminal or $.*
2. *Partition the symbols in as many as groups possible, in such a way that f_a and g_b are in the same group if a =· b.*
3. *Create a directed graph whose nodes are in the groups, next for each symbols a and b do:*
   (a) if a <· b, place an edge from the group of g_b to the group of f_a.
   (b) if a ·> b, place an edge from the group of f_a to the group of g_b.
4. *If the constructed graph has a cycle then no precedence functions exist. When there are no cycles collect the length of the longest paths from the groups of f_a and g_b respectively.*

- Using the algorithm leads to the following graph:

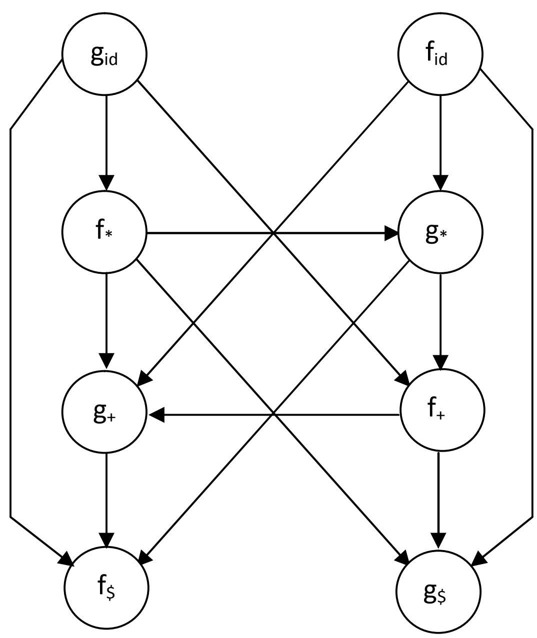


Fig. 3.1.5 Operator precedence graph

- From which we can extract the following precedence functions:

	id	+	*	$
f	4	2	4	0
g	5	1	3	0

Table 3.1.14 precedence function

14. LR parsing.

- LR parsing is most efficient method of bottom up parsing which can be used to parse large class of context free grammar.
- The technique is called LR(k) parsing; the “L” is for left to right scanning of input symbol, the “R” for constructing right most derivation in reverse, and the k for the number of input symbols of lookahead that are used in making parsing decision.
- There are three types of LR parsing,
    1. SLR (Simple LR)
    2. CLR (Canonical LR)
    3. LALR (Lookahead LR)
- The schematic form of LR parser is given in figure 3.1.6.
- The structure of input buffer for storing the input string, a stack for storing a grammar symbols, output and a parsing table comprised of two parts, namely action and goto.

Properties of LR parser

- LR parser can be constructed to recognize most of the programming language for which CFG can be written.
- The class of grammars that can be parsed by LR parser is a superset of class of grammars that can be parsed using predictive parsers.
- LR parser works using non back tracking shift reduce technique.
- LR parser can detect a syntactic error as soon as possible.

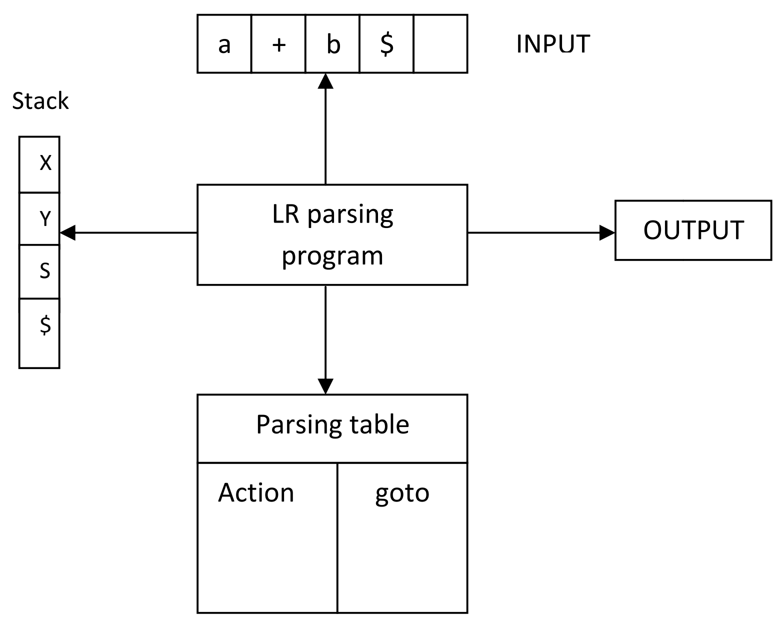


Fig.3.1.6. Model of an LR parser

15. Explain the following terms.

1. **Augmented grammar:** If grammar G having start symbol S then augmented grammar is the new grammar G’ in which S’ is a new start symbol such that S’ -> .S.
2. **Kernel items:** It is a collection of items S’->.S and all the items whose dots are not at the

left most end of the RHS of the rule.

3. **Non-Kernel items**: It is a collection of items in which dots are at the left most end of the RHS of the rule.
4. **Viable prefix:** It is a set of prefix in right sentential form of the production A-> α, this set can appear on the stack during shift reduce action.

16. SLR Parsing.

- SLR means simple LR. A grammar for which an SLR parser can be constructed is said to be an SLR grammar.
- SLR is a type of LR parser with small parse tables and a relatively simple parser generator algorithm. It is quite efficient at finding the single correct bottom up parse in a single left to right scan over the input string, without guesswork or backtracking.
- The parsing table has two states (action, Go to).

  The parsing table has four values:
  - ✓ Shift S, where S is a state
  - ✓ reduce by a grammar production
  - ✓ accept, and
  - ✓ error

Example:

E→ E + T | T
T→ TF | F
F→ F * | a | b
Augmented grammar: E' → .E
Closure(I)

I_0 : E' → .E E → .E + T E → .T T→.TF T → .F F → .F * F → .a F → .b	I_1 : Go to (I_0,E) E' → E. E → E.+T	I_2 : Go to (I_0, T) E → T. T→ T.F F→.F * F→.a F→.b
I_3 : Go to (I_0,F) T→ F. F→F.*	I_4 : Go to (I_0,a) F→a.	I_5 : Go to (I_0,b) F→ b.
I_6 : Go to (I_1,+) E→ E+.T T→.TF T→ .F F→.F* F→ .F* F→.a	I_7 : Go to (I_2,F) T→TF. F→F.*	I_8 : Go to (I_3,*) F→F *.

F→.b		
I_9 : Go to(I_6,T) E→ E + T. T→T.F F→.F * F→.a F→.b		

Table 3.1.15. Canonical LR(0) collection

Follow:
Follow (E) : {+,$}
Follow (T) :{+,a,b,$}
Follow (F) : {+,*,a,b,$}

SLR parsing table :

Action						Go to		
state	**+**	*****	**a**	**b**	**$**	**E**	**T**	**F**
0			S_4	S_5		1	2	3
1	S_6				Accept			
2	R_2		S_4	S_5	R_2			7
3	R_4	S_8	R_4	R_4	R_4			
4	R_6	R_6	R_6	R_6	R_6			
5	R_6	R_6	R_6	R_6	R_6			
6			S_4	S_5			9	3
7	R_3	S_8	R_3	R_3	R_3			
8	R_5	R_5	R_5	R_5	R_5			
9	R_1		S_4	S_5	R_1			7

Table 3.1.16. SLR Parsing table

17. CLR parsing.

Example : S→ C C
C→ a C | d

Augmented grammar: S' → .S, $

Closure(I)

I_0 : S'→ .S, $ S→ .CC, $ C→ .a C , a | d C→ .d , a | d	I_1 : Go to(I_0,S) S'→S. , $	I_2 : Go to(I_0,C) S→ C.C, $ C→ .a C , $ C→ .d, $
I_3 : Go to (I_0,a) C→ a.C, a | d C→ .a C, a | d	I_4 : Go to (I_0,d) C→ d. , a | d	I_5 : Go to (I_2,C) S→ C C., $

C → .d , a | d		
I_6 : Go to (I_2,a) C→ a.C , \$ C→ .a C , \$ C→ .d , \$	I_7 : Go to (I_2,d) C→ d. ,\$	I_8 : Go to (I_3,C) C→ a C. , a | d
I_9 : Go to (I_6,C) C→ a C. ,\$		

Table 3.1.17. Canonical LR(1) collection

Parsing table:

	Action			**Go to**	
state	**a**	**d**	**\$**	**S**	**C**
0	S_3	S_4		1	2
1			Accept		
2	S_6	S_7			5
3	S_3	S_4			8
4	R_3	R_3			
5			R_1		
6	S_6	S_7			9
7			R_3		
8	R_2	R_2			
9			R_2		

Table 3.1.18. CLR Parsing table

18. LALR Parsing.

- LALR is often used in practice because the tables obtained by it are considerably smaller than canonical LR.

  Example : S→ C C

  C→ a C | d

  Augmented grammar: S' → .S, \$

  Closure(I)

I_0 : S'→ .S, \$ S→ .CC, \$ C→ .a C , a | d C→ .d , a | d	I_1 : Go to(I_0,S) S'→S. , \$	I_2 : Go to(I_0,C) S→ C.C, \$ C→ .a C , \$ C→ .d, \$
I_3 : Go to (I_0,a) C→ a.C, a | d C→ .a C, a | d C → .d , a | d	I_4 : Go to (I_0,d) C→ d. , a | d	I_5 : Go to (I_2,C) S→ C C., \$
I_6 : Go to (I_2,a) C→ a.C , \$ C→ .a C , \$ C→ .d , \$	I_7 : Go to (I_2,d) C→ d. ,\$	I_8 : Go to (I_3,C) C→ a C. , a | d

I_9 : Go to (I_6,C) C→ a C. ,$		

Table 3.1.19. Canonical LR(1) collection

Now we will merge state 3, 6 then 4, 7 and 8, 9.

I_{36} : C→ a.C , a \| d \| $
C→ .a C , a \| d \| $
C→ .d , a \| d \| $
I_{47} : C→ d. , a \| d \| $
I_{89}: C→ aC. ,a \| d \| $

Parsing table:

Action				Go to	
State	**a**	**d**	**$**	**S**	**C**
0	S_{36}	S_{47}		1	2
1			Accept		
2	S_{36}	S_{47}			5
36	S_{36}	S_{47}			8 9
47	R_3	R_3	R_3		
5			R_1		
89	R_2	R_2	R_2		

Table 3.1.20. LALR parsing table

19. Error recovery in LR parsing.

- An LR parser will detect an error when it consults the parsing action table and finds an error entry.
- Consider the grammar, E-> E+E \| E*E \| (E) \| id

I_0: E'->.E E->.E+E E->.E*E E->.(E) E->.id	I_1: E'->E. E->E.+E E->E.*E	I_2: E-> (E.) E->.E+E E->.E*E E->.(E) E->.id	I_3: E->id.	I_4: E-> E+.E E->.E+E E->.E*E E->.(E) E->.id
I_5: E-> E*.E E->.E+E E->.E*E E->.(E) E->.id	I_6: E-> (E.) E->E.+E E->E.*E	I_7: E->E+E. E->E.+E E->E.*E	I_8: E->E*E. E->E.+E E->E.*E	I_9: E->(E).

Table 3.1.21. Set of LR(0) items for given grammar

- Parsing table given below shows error detection and recovery.

States	Action						goto
	id	+	*	(	)	$	E
0	S3	E1	E1	S2	E2	E1	1
1	E3	S4	S5	E3	E2	Acc	
2	S3	E1	E1	S2	E2	E1	6
3	R4	R4	R4	R4	R4	R4	
4	S3	E1	E1	S2	E2	E1	7
5	S3	E1	E1	S2	E2	E1	8
6	E3	S4	S5	E3	S9	E4	
7	R1	R1	S5	R1	R1	R1	
8	R2	R2	R2	R2	R2	R2	
9	R3	R3	R3	R3	R3	R3	

Table 3.1.22. LR parsing table with error routines

The error routines are as follow:

- E1: push an imaginary id onto the stack and cover it with state 3.
  Issue diagnostics "missing operands". This routine is called from states 0, 2, 4 and 5, all of which expect the beginning of an operand, either an id or left parenthesis. Instead, an operator + or *, or the end of the input found.
- E2: remove the right parenthesis from the input. Issue diagnostics "unbalanced right parenthesis". This routine is called from states 0, 1, 2, 4, 5 on finding right parenthesis.
- E3: push + on to the stack and cover it with state 4
  Issue diagnostics "missing operator". This routine is called from states 1 or 6 when expecting an operator and an id or right parenthesis is found.
- E4: push right parenthesis onto the stack and cover it with state 9. Issue diagnostics "missing right parenthesis". This routine is called from states 6 when the end of the input is found. State 6 expects an operator or right parenthesis.

Stack	Input	Error message and action
0	id+)$	
0id3	+)$	
0E1	+)$	
0E1+4	)$	
0E1+4	$	"unbalanced right parenthesis" e2 removes right parenthesis
0E1+4id3	$	"missing operands" e1 pushes id 3 on stack
0E1+4E7	$	
0E1	$	

Table 3.1.23. Parsing and Error recovery moves made by LR parser