4 Introduction :—

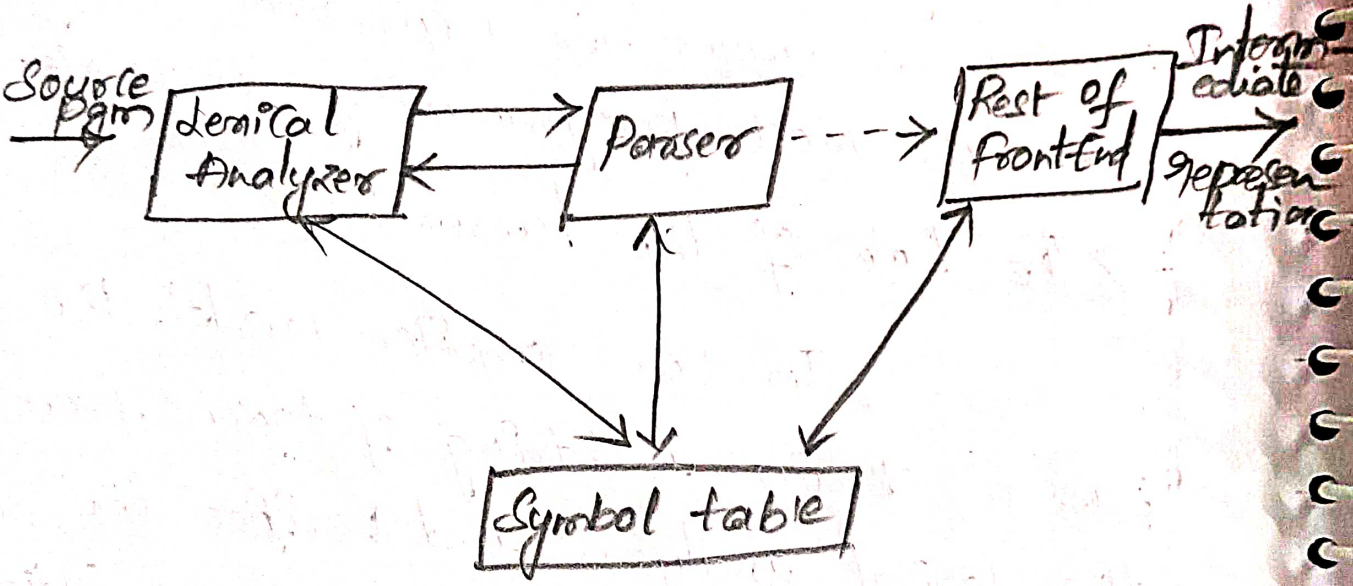
In this chapter we look after the typical grammars for arithmatic expressions. Grammars for expressions suffice for illustrating the essence of parsing, since parsing techniques for expressions carry over the most programming constructs.

⇒ The Role of the Parser :

In our compiler model, the parser obtain a string of tokens from the lexical analyzer and verifies the string of token names can be generated by the grammar for the source language. The Parser constructs a parse tree and passes it to the rest of the compiler for further processing.

There are three general type of parsers for grammar : universal, topdown, bottom-up. The methods commonly used in compilers can be classified as being either top down or bottomup.

as implied by their names topdown method build parse tree from top (root) to the bottom (leaves); while bottomup method start from leaves and work their up to the root. In either case, the input to the parser is scanned left to right, one symbol at a time.



The most efficient topdown and bottomup methods work only for subclasses of grammars, but several of these classes particularly, LL and LR grammars. Parsers implemented by hand often use LL grammars. There are number of tasks that might be conducted during parsing, such as collecting information about various tokens into the symbol table, performing type checking and other kinds of semantic analysis.

⇒ Representative Grammars :—

$$\left.\begin{array}{l} E \rightarrow E+T \mid T \\ T \rightarrow T*F \mid F \\ F \rightarrow (E) \mid id \end{array}\right\} \text{LR, LL}$$

E represents expressions consist of terms seperated by + and sign, T represents Consisting factors seperated by * sign. and F represents consisting of factors that can be represented either parenthesized expressions (or) identifiers: The above grammar represent LR grammars.

$$\left.\begin{array}{l} E \rightarrow TE' \\ E' \rightarrow +TE' \mid \epsilon \\ T \rightarrow FT' \\ T' \rightarrow *FT' \mid \epsilon \\ F \rightarrow (E) \mid id \end{array}\right\} \text{non left recursive.}$$

The following grammar treats + and * alike, so it is useful for illustrating techniques for handling ambiguities during parsing.

$$E \rightarrow E+E \mid E*E \mid (E) \mid id$$

⇒ Syntax Error Handling:

This section consider the nature of syntactic errors and general strategies for error discover recovery. Two of these strategies called Panic mode and phrase level recovery.

Some common Programming errors can occur at many different levels:

* lexical errors include misspelling of identifiers, keywords or operators.
* Syntactic errors include misplaced semi-colon (or) extra (or) missing braces, that is "(" (or) ")".
* Semantic errors include type mismatches between operators and operands.
* Logical errors can be anything from incorrect reasoning on the part of programmer to the use in C program of assignment operator = instead of comparision operator ==.

The error handler in a parser has goals that are simple to state but challenging to realize:

→ Report the presence of errors clearly and accurately.

→ Recover from each error quickly enough to detect subsequent error.
→ Add minimal overhead to the processing of correct programs.

⇒ Error Recovery Strategies :

1. In Panic mode, the parser discards the input symbols one at a time until one of a designated set of synchronizing tokens is found. The synchronizing tokens are usually delimiters such as semicolon or } while panic mode skips a considerable amount of input without checking it for additional errors.

2. In phrase level Recovery, A typical local correction is to replace a comma by a semicolon, delete an extraneous semicolon or inserting a missing semicolon.

3. Error productions Strategy helps in generating appropriate error messages that are encountered while parsing a given input string.

4. Global Corrections : Any given input string x and grammar G, the algorithm itself can find a parse tree for a related string y, such that a no. of insertions, deletions and changes of token require to transform x into y is possible.

II) Context free Grammar :—

For a regular grammar, the productions are restricted in two ways: The left side must be a single variable and right side can be any string of terminals and non terminals. To create grammars that are more powerful, we must ease off some of the restrictions. By permitting anything on the right side, but retaining restrictions on the left side, we get Context free grammars.

A Grammar $G = (V, P, T, S)$

V, T, P, S are the four important components in the grammatical description of a language.

V – the set of variables also called non terminals. Each variable represents a set of strings, simply a language.

* Upper case letters
* Letters S is Start Symbol
* Lower case strings.

T – the set of terminals, which are set of symbols that form the strings of the language, also called terminal symbols.

* Lower case letters.
* Operator
* Punctuation marks
* Digits
* Bold face string

P – the finite set of productions (or) rules that represents the recursive definition of language.

S — the start symbol.

The language generated (defined, derived, produced) by a CFG is the set of all strings of terminals that can be produced from start symbol S using the productions as substitutions. A language generated by CFG is context free language (CFL)

Ex: terminal: a
nonterminal: S
production: $S \rightarrow aS$
$S \rightarrow \epsilon$

is a simple CFG that defines $L(G) = a^*$
where $V = \{S\}$ $T = \{a\}$

Ex 2: The CFG for defining palindrome over {a, b}

$P: S \rightarrow \epsilon | a | b$
$S \rightarrow aSa$, $S \rightarrow bSb$

Ex 3: CFG for $(01+01)^*(01)^*$

CFG for $(01+1)^*$ is $A \rightarrow CA/\epsilon$

CFG for $(01)^*$ is $B \rightarrow DB/\epsilon$

$D \rightarrow 01$

Hence, $S \rightarrow AB$

$A \rightarrow CA/\epsilon$

$C \rightarrow 011/1$

$B \rightarrow DB/\epsilon$

$D \rightarrow 01$

Ex: 4 $a^*(bb)^*$

$S \rightarrow aSbb/abb$

Ex: 5 The derivation of a^+ is

$S \rightarrow aS$

Ex: Find the language and derive abbabba from the following grammar.

$S \rightarrow xaax$

$x \rightarrow ax/bx/\epsilon$

CFL is $(a+b)^*aa(a+b)^*$

III) Writing a Grammar :—

A Grammar consist of a number of productions. Each production has an abstract Symbol called a nonTerminal and a no Terminal Non terminal as its left-hand side and a Sequence of one (or) more nonterminal (or) terminal symbols as its right hand side.

There are four Categories in writing a grammar.

1. Regular Expression Vs and Context free Grammar.
2. Eliminating ambigous grammar
3. Eliminating left recursion (LR)
4. Left Factoring.

1. Regular Expressions Vs CFG :

Regular Expressions (RE):

* It is used to describe the tokens of programming language.
* It is used to check whether the given input is valid (or) not using transitional
* The transition diagram has a set of states and edges.

* It has no start symbol.
* It is useful for describing the structure of lexical constructs.

CFG:

* It consist of a quadruple where S → start symbol, Production → P, T, V.
* It is used to check whether the given input is valid or not using derivation.
* The CFG has set of productions.
* It has start symbol.

Ambiguous Grammar:

A CFG is ambiguous if there exist more than one parse tree or Equivalently more than one left-most and right-most derivation for atleast one word in its CFL.

Ex: $E \rightarrow id \mid E+E \mid E*E \mid E-E$

id+id*id LMD:

$E \rightarrow E+E$
$\rightarrow id+E$
$\rightarrow id+E*E$
$\rightarrow id+id*E$
$\rightarrow id+id*id$

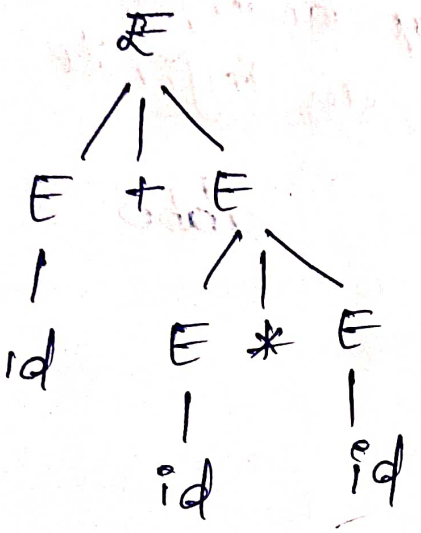
RMD → $E \to E*E$

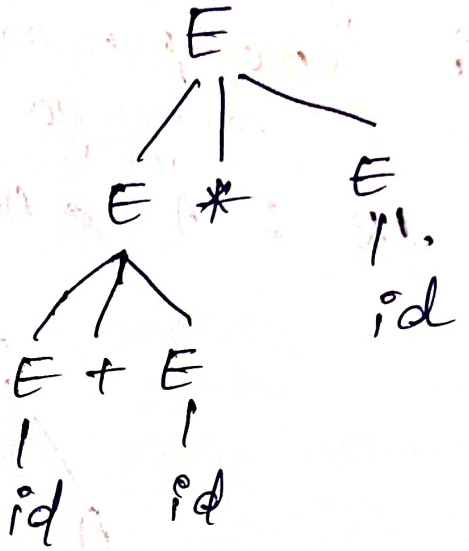
$\to E*id$

$\to E+E*id$

$\to E+id*id$

$\to id+id*id$.





Leftmost Derivation & Rightmost Derivation:

Ex: $S \to baXaS/ab$

$X \to Xab/aa$

Find LMD & RMD for string = baaabab aab

The LMD $S \to baXaS$

$\to baXabaS$

$\to baXababaS$

$\to baaaabababaS$

$\to baaaababaab$

The RMD $S \to baXaS$

$\to baXaaab$

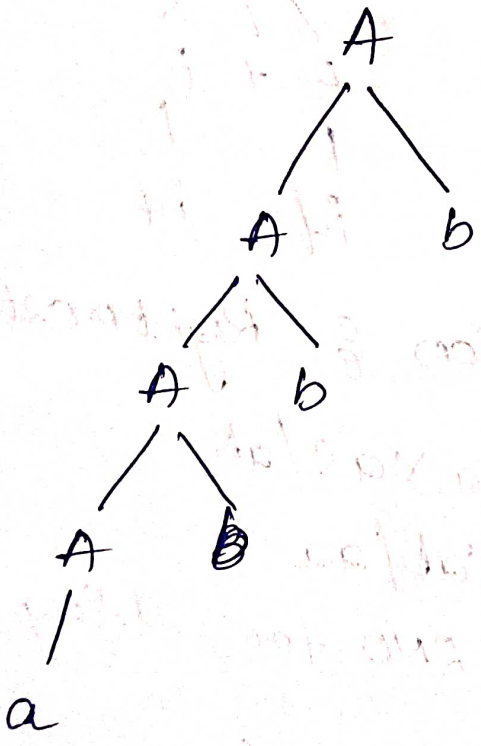
$\to baXabaaab \to baXababaaab$

$\to baaababaaab$

Left Recursion:

A Grammar is left recursive if it has a nonterminal A such that there is a derivation $A \rightarrow^{+} A\alpha$ for some string α. If this grammar is used in some parsers (topdown parser), parser may go into an infinite loop.

LR : $A \rightarrow Ab/a$ abbb



Eliminating Ambiguous (or) Un Ambiguous Grammar:—

We can remove ambiguity solely on the basis of following two properties

1. Precedence
2. Associativity

1. Precedence : If the different operators are used, we will consider the precedence of the operators.

1. The level at which the production is present denotes the priority of operators used.
2. The production at higher levels will have operators with less priority.
3. The production at lower levels will have operators with high priority.

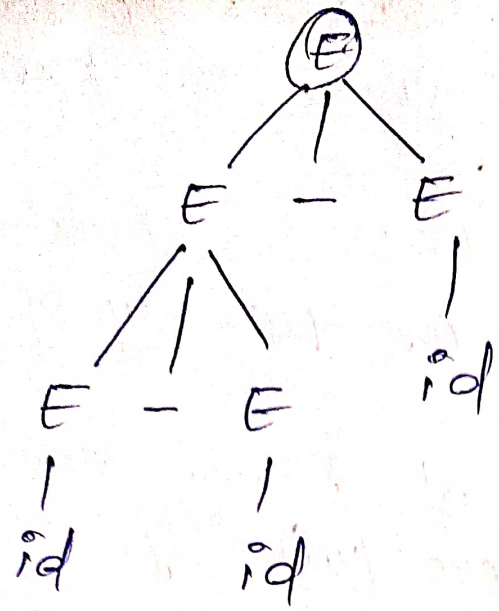
2. Associativity : If the same precedence operators are in production, then

1. If the associativity is left to right then we have to prompt a left recursion in the production. (+, −, *, /)
2. If the associativity is right to left then we have to prompt a right recursion in the productions. (^)

Ex: $E \rightarrow E-E/id$

$S = \{id, id-id, id-id-id \ldots\}$

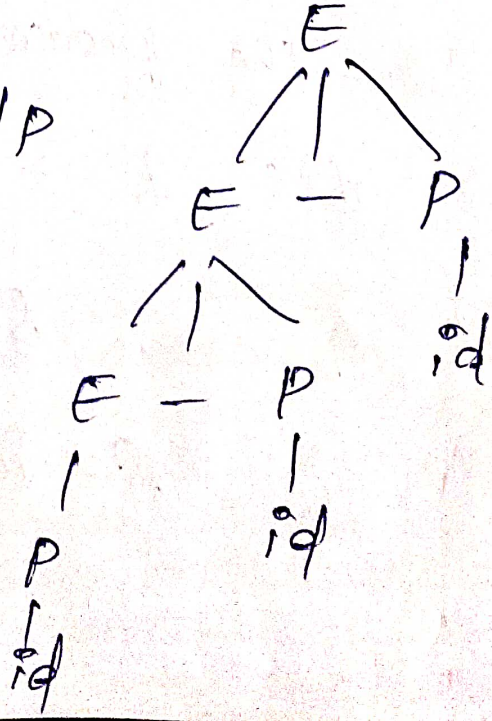
$S = id-id-id$



The parse tree which grows on the left-side of the root will be the correct parse tree in order to make it unambiguous.

So, to make the above grammar unambiguous, simply make the grammar left recursive by replacing the left-most non-terminal E in the right-side of production with another random variable.

$$E \rightarrow E - P \mid P$$
$$P \rightarrow id$$



Similarly, the unambiguous grammar of grammar $E \rightarrow P \wedge E / P$

$P \rightarrow id$

Ex 2: $E \rightarrow E+E / E*E / id$

$S = id + id * id$

$E \rightarrow E+P / P$

$P \rightarrow P*Q / Q$

$Q \rightarrow id$

(or)

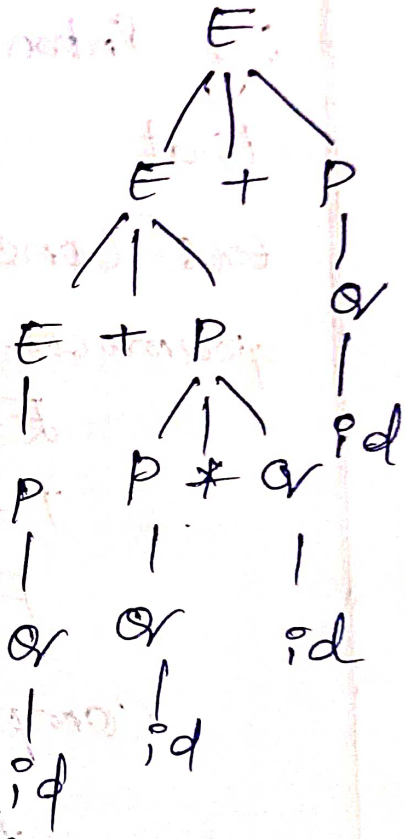
$E \rightarrow E+P$

$E \rightarrow P$

$P \rightarrow P*Q$

$P \rightarrow Q$

$Q \rightarrow id$



Eliminating left Recursion :-

A Grammar $G(V, T, P, S)$ is left recursive if it has a production in the form. $A \rightarrow A\alpha / \beta$

The above grammar is left recursive because the left of production

is occuring at a first position on the right side of production. It can eliminate left recursion by replacing a pair of production with

$$A \rightarrow \beta A'$$
$$A \rightarrow \alpha A' / \epsilon$$

Left Recursion can be eliminated by introducing new nonterminal A such that

Ex: Consider the left Recursion from the Grammar.

$$E \rightarrow E+T / T$$
$$T \rightarrow T*F / F$$
$$F \rightarrow E / id$$

Comparing $E \rightarrow E+T / T$ with $A \rightarrow A\alpha / \beta$

$A = E$, $\alpha = +T$, $\beta = T$

$A \rightarrow \beta A'$ means $E \rightarrow TE'$

$A' \rightarrow \alpha A' / \epsilon$ means $E' \rightarrow +TE' / \epsilon$

Comparing $T \rightarrow T*F / F$ with $A \rightarrow A\alpha / \beta$

$A = T$, $\alpha = *F$, $\beta = F$

$A \rightarrow \beta A'$ means $T \rightarrow FT'$

$T' \rightarrow *FT' / \epsilon$

After Eliminating LR

$E \rightarrow TE'$

$E' \rightarrow +TE' / \epsilon$

$T \rightarrow FT'$

$T \rightarrow *FT' / \epsilon$

$F \rightarrow (E) / id$

Ex:2 $E \rightarrow E(T) / T$

$T \rightarrow T(F) / F$

$F \rightarrow id$

Sol: ~~$E \rightarrow /E$~~ ~~T'~~ $E \rightarrow TE'$

$E \rightarrow (T)E' / \epsilon$

$T \rightarrow FT'$

$T' \rightarrow (F)T' / \epsilon$

$F \rightarrow id$

Left Factoring :-

Left factoring is a process by which the grammars with common prefixes is transformed to make useful for Top down parsers.

In left factoring

* we make one production for each common prefixes.
* The common prefix may be a terminal (or) non terminal (or) combination of both
* Rest of the derivation is added by new productions.

Ex:

$A \rightarrow a\alpha_1 + a\alpha_2 + a\alpha_3 \xrightarrow{LF} A \rightarrow aA'$

$A' \rightarrow \alpha_1 | \alpha_2 | \alpha_3$

Ex2:

$S \rightarrow iEtS | iEtSeS | a$

$E \rightarrow c$

A) $S \rightarrow iEtSS' | a$

$S' \rightarrow \epsilon | eS$

$E \rightarrow c$

Ex3: $A \rightarrow aAB | aBc | aAc$

A) $A \rightarrow aA'$

$A' \rightarrow AB | Bc | Ac \rightarrow A' \rightarrow AD | Bc$

$D \rightarrow B | c$

IV) Top Down Parser :—

Top down parsing is based on left most Derivation where as Bottom up parsing is dependent on Reverse Right Most Derivation.

The process of constructing the parse tree which starts from the root and goes down to the leaf is Top down parsing.

1. Top Down parsers constructs from the Grammar which free from ambiguity and left recursion.
2. Top Down parsers uses left most derivation to construct a parse tree.
3. It does not allow Grammar with Common prefixes.

Top Down Parser

- Recursive Decent Parsing and LL parsing
- Non Recursive parsing (or) Predictive Parsing (or) LL(1) Parsing (or) Table Driven parsing

Recursive Descent Parsing :-

It is a kind of Top Down parse

A top down parser builds the parse tree from the top to down, starting with the start nonterminal. A Predictive Parser is a special case of Recursive Descent Parser, where no Backtracking is required.

Ex: 1. $E \rightarrow iE'$

$E' \rightarrow +iE' / \epsilon$

```
E()
{
if(input==i)
   input++;
EPrime();
}
```

```
EPrime()
{
if(input=='+')
  {
   input++;
  }
if(input=='i')
   input++;
EPrime();
else
return();
}
```

The major approach of recursive descent parsing is to relate each nonterminal with a procedure. The objective of each procedure is to read sequence of input characters that can be produced by the corresponding nonterminal and return a pointer to the root parse tree for the nonterminal. The structure of procedure is prescribed by the productions for the equivalent nonterminal.

Eg:

Imp

$E \rightarrow TE'$

$E' \rightarrow +TE'/\epsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT'/\epsilon$

$F \rightarrow (E)/id$

```
E()
{
  T();
  Eprime();
}
```

```
Eprime()
{
if(input==+')
{ input++;
  T();
  Eprime();
else
  return();
}
```

```
T()
{
 F();
 Tprime();
}
```

```
Tprime()
{
if(input==*)
 {input++;
  F();
   Tprime();
else
 return();
```

```
F()
{
if(input=='(')
 {
  input++;
   E();
   if(input==')')
       input++;
 }
else if (input=='id')
       input++;
 }
```

Ex: id+id*id

LL(1) Parser :—

Here the first L represents that the scanning of input will be done from left to right and second L shows that in this parsing technique, we are going to use Left Most Derivation Tree. Finally 1, represents the number of look-ahead, which means how many symbols are you going to see when you want to make a decision.

Essential conditions to check are as follows:

* The grammar is free from left Recursion
* The grammar should not be ambiguous.
* The grammar has to be left-factored so that the grammar is deterministic.
* Calculate First and Follow.
* Construction of parsing table.
* check whether input string is accepted by the parser (or) not.

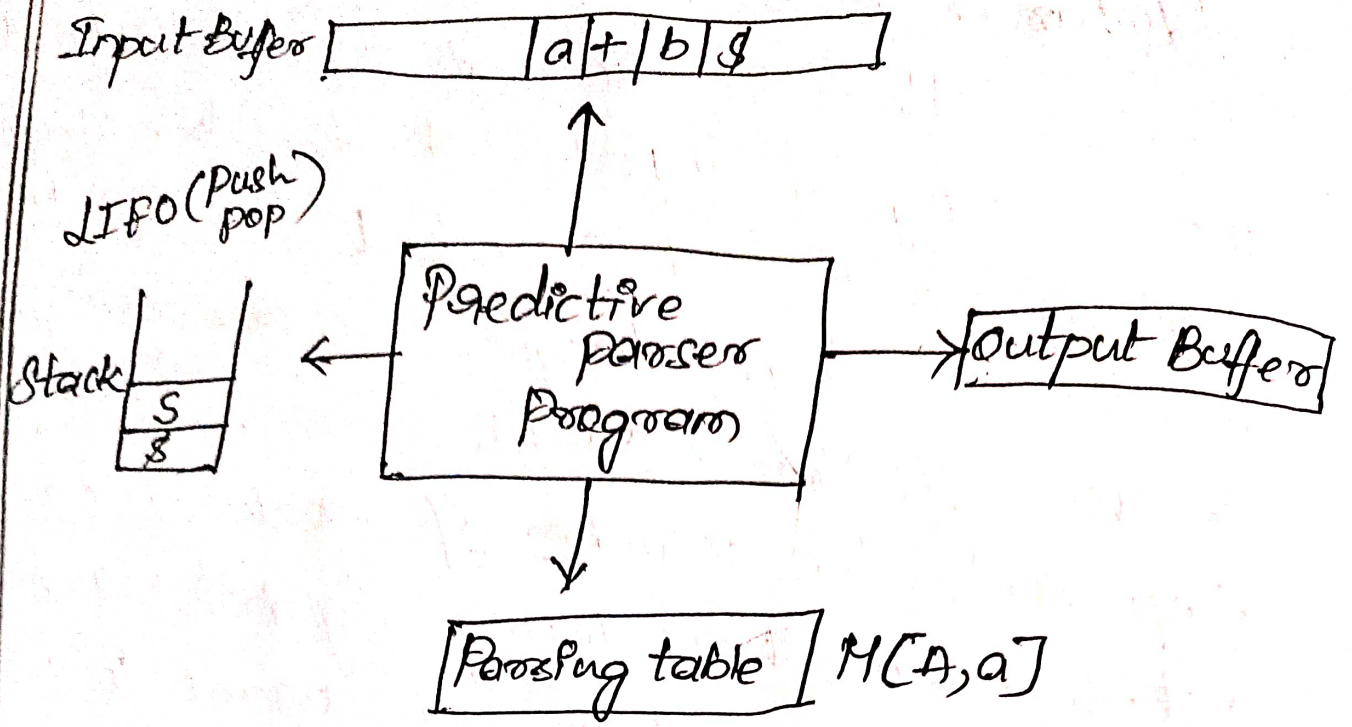


Fig: Model of LL(1) parser.

3. Calculate First and Follow :—

1. FIRST :

→ If $A \rightarrow a\alpha$, $\alpha \in (V \cup T)^*$ then

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

→ If $A \rightarrow \epsilon$ then $First(A) = \{\epsilon\}$

→ If $A \rightarrow BC$ then

$First(A) = First(B)$ if $First(B)$ doesn't contain ϵ.

if $first(B)$ contains ϵ then

$First(A) = First(B) \cup First(C)$

2. Follow:

→ 's' then Follow(S) = {$}

→ If $A \to \alpha B\beta$ then Follow(B) = First(β) if First(β) doesn't contain 'ϵ'.

→ If $A \to \alpha B$ then Follow(B) = Follow(A)

→ If $A \to \alpha B/\beta$ where $\beta \to \epsilon$ then follow(B) = Follow(A)

Ex: 1 $E \to TE'$

$E' \to +TE'/\epsilon$

$T \to FT'$

$T' \to *FT'/\epsilon$

$F \to (E)/id$

First(E) = {(, id}

First(E') = {+, ϵ}

First(T) = {(, id}

First(T') = {*, ϵ}

First(F) = {(, id}

Follow(E) = {$,)}

Follow(E') = {$,)}

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

Follow(T') = {+, $,)}

Follow(F) = {*, +, $,)}

Ex2: $S \rightarrow ABCDE$

$A \rightarrow a/\epsilon$

$B \rightarrow b/\epsilon$

$C \rightarrow c$

$D \rightarrow d/\epsilon$

$E \rightarrow e/\epsilon$

First(S) = { a, b, c }

First(A) = { a, ε }

First(B) = { b, ε }

First(C) = { c }

First(D) = { d, ε }

First(E) = { e, ε }

Follow(S) = { $ }

Follow(A) = { b, c }

Follow(B) = { c }

Follow(C) = { d, e, $ }

Follow(D) = { c, $ }

Follow(E) = { $ }

Ex3: $S \rightarrow Bb/Cd$

$B \rightarrow aB/\epsilon$

$C \rightarrow cC/\epsilon$

Ex:4 $S \rightarrow ACB/CbB/Ba$

$A \rightarrow da/BC$

$B \rightarrow g/\epsilon$

$C \rightarrow h/\epsilon$

Ex: 5 $S \rightarrow aABb$

$A \rightarrow c/\epsilon$

$B \rightarrow d/\epsilon$

4. Calculate / Construction of parsing table:

$A \rightarrow \alpha$

1. If the production is in form of
   $A \rightarrow \alpha$ then $First(\alpha) \rightarrow a$
   Add $A \rightarrow \alpha$ to $M[A, a]$
2. $First(\alpha)$ contains ϵ (or) $A \rightarrow \epsilon$
   $Follow(A) = b$
   $A \rightarrow \epsilon$ to $M[A, b]$

Eg: $E \rightarrow E+T/T$
$T \rightarrow T*F/F$
$F \rightarrow (E)/id$

Step 1: Elimination of left recursion

$A \rightarrow A\alpha/\beta$
Replace with
$A \rightarrow \beta A'$
$A' \rightarrow \alpha A'/\epsilon$

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

Step 2: Elimination of left factoring

The above production doesn't have left factoring.

Step 3: Calculate First and follow

First(E) = { (, id }
First(E') = { +, ε }
First(T) = { (, id }
First(T') = { *, ε }
First(F) = { (, id }

Follow(E) = { $,) }
Follow(E') = { $,) }
Follow(T) = { +, $,) }
Follow(T') = { +, $,) }
Follow(F) = { *, +, $,) }

Step 4: Construction of parse table.

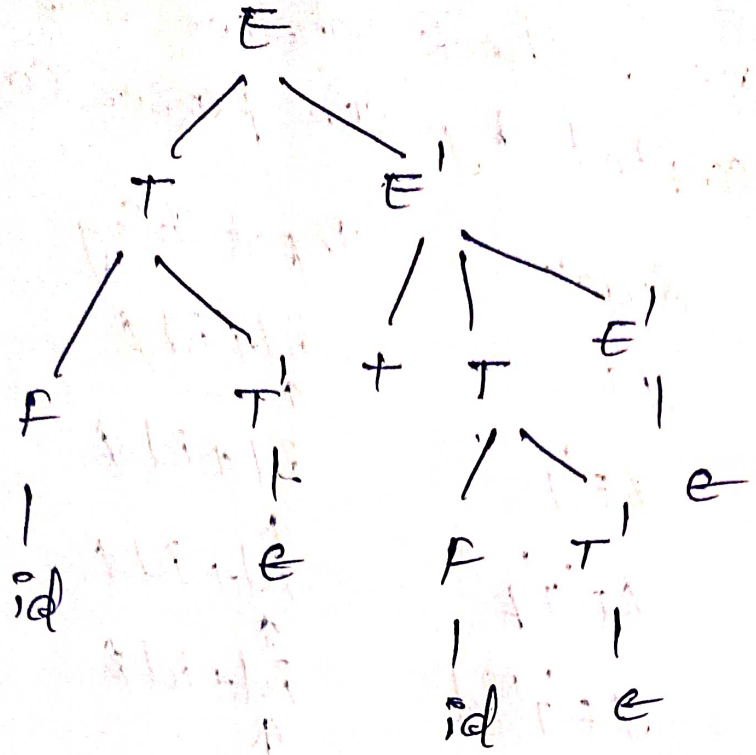
	+	*	(	)	$	id
E			E→TE'			E→TE'
E'	E'→+TE'			E'→ε	E'→ε	
T			T→FT'			T→FT'
T'	T'→ε	T'→*FT'		T'→ε	T'→ε	
F			F→(E)			F→id

1. E → TE'
   A α

First(TE') = { (, id }
Add E→TE' to M[E, (]
M[E, id]

Step 5 : check whether the string is accepted by the parser (or) not.

Stack	Input String	Action
$E	id+id$	
	↑	$E \rightarrow TE'$
$E'T	id+id$	
	↑	$T \rightarrow FT'$
$E'T'F	id+id$	
	↑	$F \rightarrow id$
$E'T'id	id+id$	
	↑	pop
$E'T'	+id$	
	↑	$T' \rightarrow \epsilon$
$E'	+id$	
	↑	$E' \rightarrow +TE'$
$E'T+	+id$	
	↑	pop
$E'T	id$	
	↑	$T \rightarrow FT'$
$E'T'F	id$	
	↑	$F \rightarrow id$
$E'T'id	id$	
	↑	pop
$E'T'	$	
	↑	$T' \rightarrow \epsilon$
$E'	$	
	↑	$E' \rightarrow \epsilon$
$	$	
	↑	Accepted.



Eg: $S \rightarrow (L) / a$

$L \rightarrow L, S / S$

Step 1: Left recursion Elimination

$S \rightarrow (L) / a$

$L \rightarrow SL'$

$L' \rightarrow , SL' / \epsilon$

Step 2: No left-factoring

Step 3: Calculate First & follow

First(S) = { (, a }

First(L) = { (, a }

First(L') = { , , ε }

Follow(S) = {$, >, ,}

Follow(L) = {>}

Follow(L') = {>}

Step 4: Construction of parsing table

	(	)	a	,	$
S	S→(L)		S→a		
L	L→SL'		L→SL'		
L'		L'→ϵ		L'→,SL'	

Step 5:

Stack	Input	Action
$S	(a)$	S→(L)
$)L(	(a)$	pop
$)L	a)$	L→SL'
$)L'S	a)$	S→a
$)L'a	a)$	pop
$)L'	)$	L'→ϵ
$)ϵ	)$	pop
$	$	accepted

1. Shift Reduce Parser:

It is a Bottom up parser techniquer. we use two datastructure that is Stack to store Symbols of the grammar ($) and Input buffer stores Inputstring ends with ($). Actions of Shift reduce parser

1. Shift (push)
2. Reduce (POP)
3. Accept
4. Error

Ex: 1 $E \rightarrow E+T/T$

$T \rightarrow T*F/F$

$F \rightarrow (E)/id$

$S = id*id$

Stack	Input Buffer	Action
$	id*id$	Shift
$id	*id$	Reduce by F→id
$F	*id$	Reduce by T→F
$T	*id$	Shift
$T*	id$	Reduce by Shift

$ T * id	$	reduce F → id
$ T * F	$	reduce E → T → T * F
$ T	$	reduce E → T
$ E	$	accepted.

Ex: $S \rightarrow (L) / a$

$L \rightarrow L, S / S$

$S = (a, (a, a))$

Stack	Input Buffer	Action.
$	(a, (a, a))$	Shift
$ (	a, (a, a))$	Shift
$ (a	, (a, a))$	reduce S → a
$ (S	, (a, a))$	Shift
$ (L	, (a, a))$	reduce L → S
$ (L,	(a, a))$	Shift
$ (L, (	a, a))$	Shift
$ (L, (a	, a))$	Shift
$ (L, (S	, a))$	reduce S → a
$ (L, (L	, a))$	reduce L → S

$ (L , (L	, a)) $	Shift
$ (L , (L ,	a)) $	Shift
$ (L , (L , a	)) $	reduce S → a
$ (L , (L , S	)) $	reduce L → L, S
$ (L , (L	)) $	Shift
$ (L , (L)	) $	reduce S → (L)
$ (L , S	) $	Shift
$ (L , S)	$	reduce L → L, S
$ (L)	$	reduce (L) → S
$ S	$	Accepted.

2) CLR Parser :—

The CLR parser stands for Canonical LR parser. It is a "more powerful LR parser". It make use of lookahead Symbols. This method uses a large set of items called LR(1) items. The general Syntax becomes [A → α.B, a]

where A → α.B is the production and a is a terminal or right