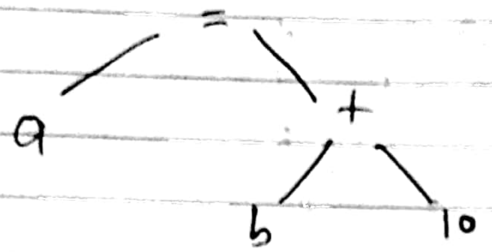
Chapter 3: Parsing theory

Parser:

A parsing or Syntax analysis is a process which takes the input string ω and produces either a parse tree or generates the syntactic errors.

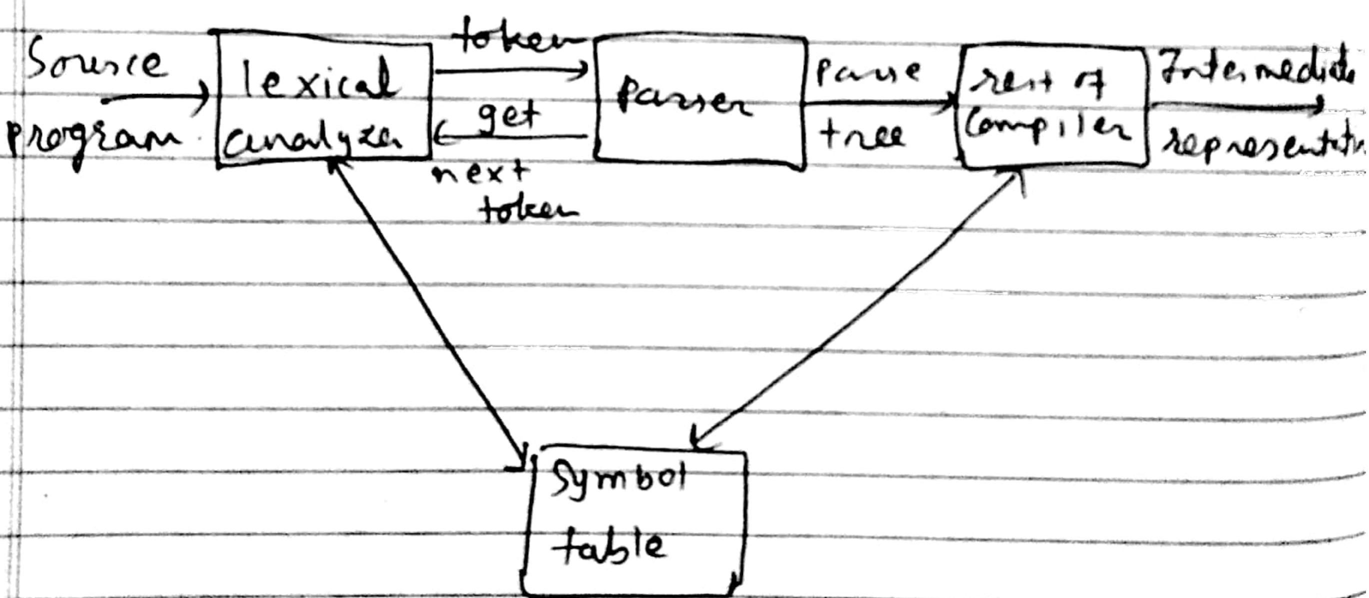
for ex:

a = b + 10



Role of Parser:

Position of parser in Compiler Model.



→ The parser obtains a String of tokens from the lexical analyzer as shown in fig. and verifies that the String can be generated by the grammar for Source language we expect the parser to report any Syntax errors in an intellible fashion It should also recover from commonly occurring errors so that it can continue processing the remainder of its input.

→ why lexical and syntax analyzer are separated out?
→ Simplicity.
→ processing of secondary function become easier
→ portability.
→ It accelerates the process of compilation and secondly the errors in the source input can be identified precisely.

Two types of parsing :

Top down parsing | Bottom up parsing.

These parsing techniques work on the following principle.

1 The parser scans the input string from left to right and identifies that the derivation is leftmost or rightmost

2 The parser makes use of production rules for choosing the appropriate derivation. The different parsing techniques uses different approaches in selecting the appropriate rules for

derivation. And finally a parse tree is constructed.

When the parse tree can be constructed from root and expanded to leaves then such type of parser is called top-down parser. The name itself tells us that the parse tree can be built ~~in~~ from top to Bottom.

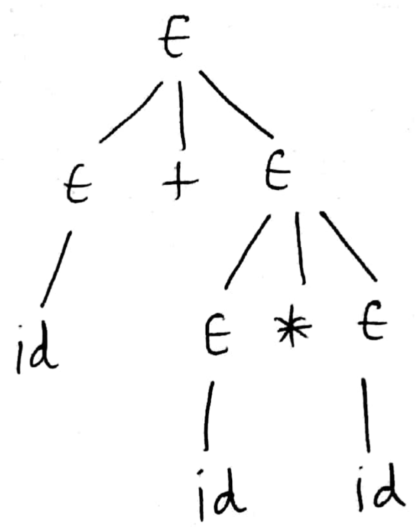
When parse tree can be constructed from leaves to root, then such type of parser is called as bottom up parser.

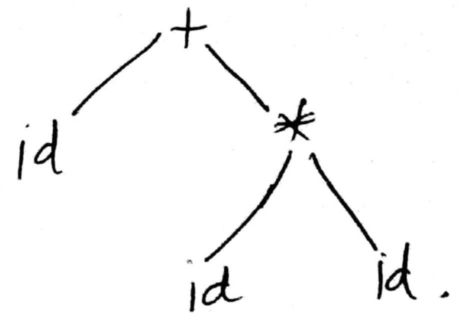
→ difference between syntax tree and parse tree

Parse tree	Syntax tree.
1) Interior nodes are non-terminals, leaves are terminals.	1. Interior nodes are "operators". leaves are operands
2) Rarely constructed as data structure.	When representing a program in a tree structure usually use a syntax tree.
3) Represent the concrete syntax of a program	Represents the abstract syntax of program.

Parse tree   Syntax tree.

4)





difference between top-down and bottom up parsing :

1). Parse tree can be built from root to leaves.

1) Parse tree can be built from leaves to root.

2) Top down parsing is simple to implement

2) Bottom up parsing is Complex to implement

3) It is applicable to small class of language.

3) It is applicable to broad class of language

4 This is less efficient parsing technique. Various problems that occur during top down parsing are ambiguity, left recursion
4 When the bottom up parser handles ambigious grammar conflicts occur in parse table.

5 Types of parsing technique : 1) RDP 2) Predictive.
5 Types of parsing technique : 1) SR 2) operator procedence 3) LR parser.

Context free grammar :

→ Context free grammar can be formally defined as a set denoted by G = (V, T, P, S). Where V and T are set of non-terminals and terminals resp respectively. P is set of production rules, where each production rule is in the form of,

non-terminals → non-terminals
or non-terminals → terminals.
S is a start symbol.

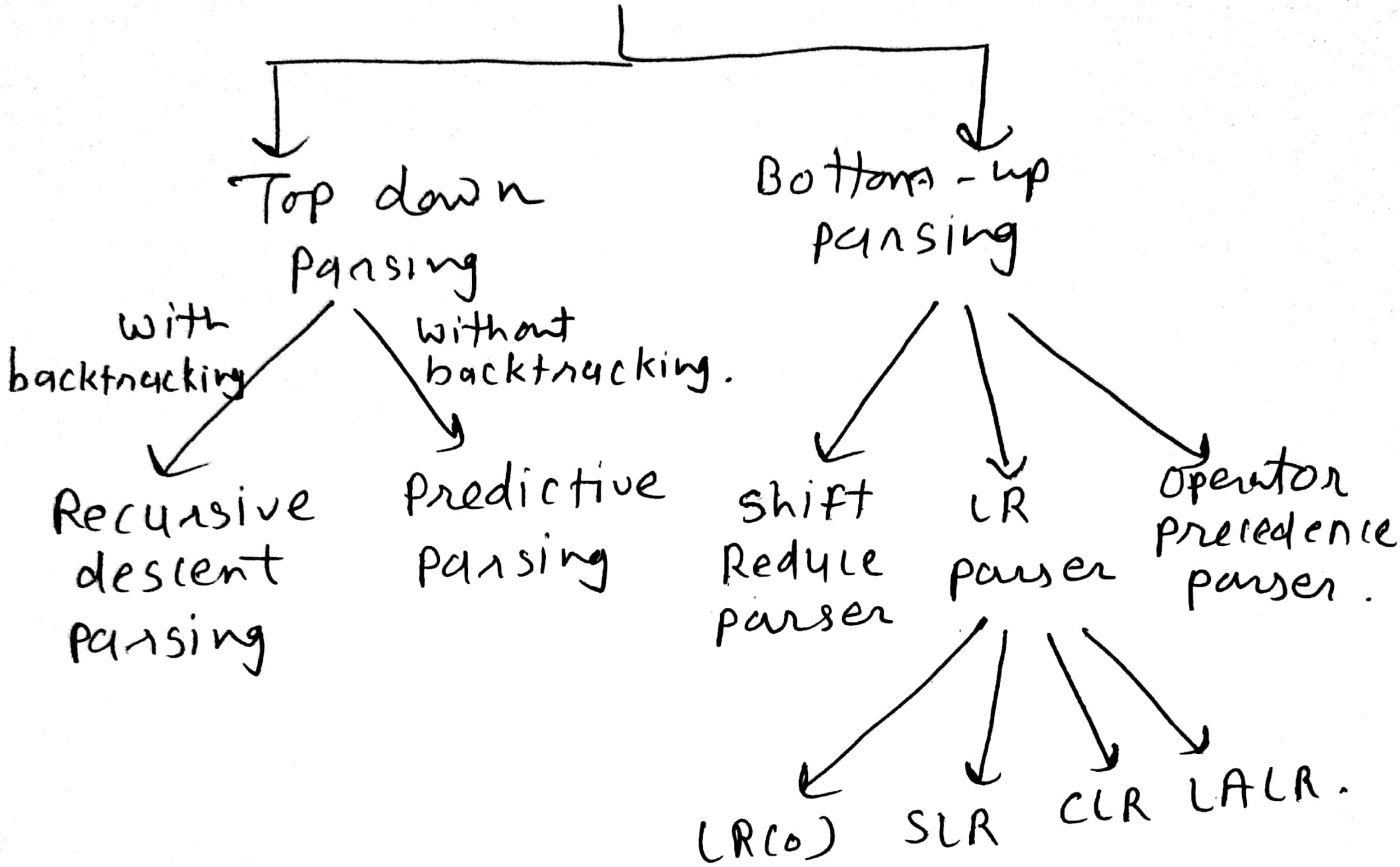
A ::= π

Which can be applied Independent of its context. These grammars are therefore known as context free grammars. CFGs are ideally suited for programming language specification.

r.e : $(0+1)^*$

P : {S → 0S | 1S | ε}   V : S
or S → 0S | 1S   T : 0, 1
S → ε.   S = S.

Types of parser.



atleast ~~two~~ two a's

$(a+b)^* a (a+b)^* a (a+b)^*$

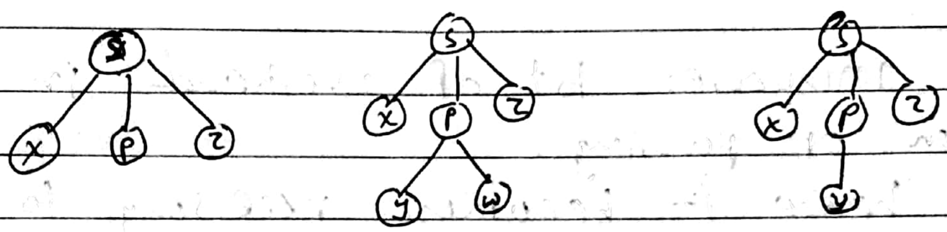
$V: \{S, A\}$ $P: \{S \rightarrow AaAaA$

$T: \{a, b\}$ $S = \{S\}$. $A \rightarrow aA \mid bA \mid \varepsilon\}$.

Top down parsing:

→ In top down parsing the parse tree is generated from top to bottom. The derivation terminates when ~~matches~~ the required input string terminates.

→ The main task in top down parsing is to find the appropriate production rule in order to produce the correct input.



$S \rightarrow xPz$ input string : xyz.

$x \rightarrow yw \mid y$

→ trial and error technique.

→ After trying all the productions if we found every production unsuitable then in that case parse tree cannot be built.

Problems with top down parsing:

1 Back tracking:

Backtracking is a technique in which for expansion of terminal symbol we chose one alternative and if some mismatch occurs then we try another alternative

Scanned by CamScanner

If for a non terminal there are multiple Production rules beginning with the Same input symbol then to get the correct derivation. We need to try all these alternatives.

→ Secondly in backtracking we need to move some levels upward in order to check the possibilities.

→ This increases lot of overhead in implementation of parsing.
And hence it becomes necessary to eliminate the back tracking by modifying the grammar.

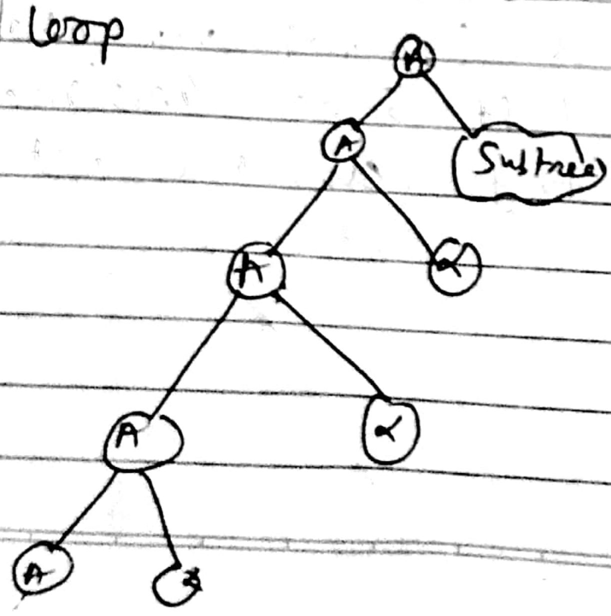
2 left recursion:
The left recursive grammar is a grammar which is as given below:

$A \to A\alpha$

$A \to$ non terminal

$\alpha \to$ Some input string /terminal

Because of left recursion the top down parser can enter in infinite loop.



Thus expansion of A causes further expansion of A only and due to generation of A.

→ This causes major problem in top down parsing and therefore elimination of left recursion is a must

Two types of LR:

1) Immediate LR
2) Indirect LR.

1) Immediate LR:

$A \rightarrow A\alpha \mid \beta$

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \epsilon$

2) Indirect LR:

$\underline{A} \rightarrow \beta\alpha$

$\beta \rightarrow \underline{A}\beta$

$\Big\} A \rightarrow A\beta\alpha.$

ex: 1 $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \beta_1 \mid \beta_2$

$A \rightarrow \beta_1 A' \mid \beta_2 A'$

$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \epsilon$

2 $E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

Scanned by CamScanner

3) $A \to ABd \mid Aa \mid a$
$B \to Be \mid b$

$\to A \to aA'$
$A' \to BdA' \mid aA' \mid \varepsilon$
$B \to bB'$
$B' \to eB' \mid \varepsilon$

4) $A \to AaB \mid x$
$B \to BCb \mid y$
$C \to Cc \mid \varepsilon$

$\to A \to xA'$
$A' \to aBA' \mid \varepsilon$
$B \to yB'$
$B' \to CbB' \mid \varepsilon$
$C \to C'$
$C' \to cC' \mid \varepsilon$

Indirect Left Recursion:

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

Step: 1

$\to S \to Aa \mid b$
$A \to Ac \mid Aad \mid bd \mid \varepsilon$

} Removing Indirect LR:

Step: 2

$\to S \to Aa \mid b$
$A \to bdA' \mid A'$
$A' \to cA' \mid adA' \mid \varepsilon$

} Removing Immediate LR:

3) left factoring:

If the grammar is left factored then it becomes suitable for the use. Basically left factoring is used when it is not clear that which of two alternative is used to expand the nonterminal. by left factoring we may be able to rewrite the production in which the decision can be deferred until enough of the input is seen to make the right choice.

$A \to \alpha\beta_1 \mid \alpha\beta_2$.

is a production then it is not possible for us to take the decision whether to choose first rule or second.

So, Replace it by,

$A \to \alpha A'$

$A' \to \beta_1 \mid \beta_2$.

ex: $S \to iEtS \mid iEtSeS \mid a$

$E \to b$

$\to$ $S \to iEtSS' \mid a$

$S' \to eS \mid \varepsilon$

2) $A \to aAb \mid aA \mid a$

$A \to aA'$

$A' \to Ab \mid A \mid \varepsilon$

$A' \to AA'' \mid \varepsilon$

$A'' \to b \mid \varepsilon$

$\to$

$A \to aA'$

$A' \to AA'' \mid \varepsilon$

$A'' \to b \mid \varepsilon$.

3) $A \to ad \mid a \mid ab \mid abc \mid b$

$A \to aA' \mid b$

$A' \to d \mid \varepsilon \mid b \mid bc$

$A' \to d \mid \varepsilon \mid bA'$

$A'' \to c \mid \varepsilon$

$\to$

$A \to aA' \mid b$

$\cancel{A' \to d \mid \varepsilon \mid b \mid bc}$

$A' \to bA' \mid d \mid \varepsilon$

$A'' \to c \mid \varepsilon$.

left Recursion : Algorithm.

Input : grammar G.

output : equivalent grammar with no left recursion.

Method:

If we have the left recursive pair of productions

$A \to A\alpha \mid \beta$

where β does not begin with A.

we can eliminate left recursion by replacing ~~the~~ this pair of Productions with

$A \to \beta A'$

$A' \to \alpha A' \mid \varepsilon$.

Q.B left factoring: Algorithm (a grammar)

→ Input: grammar G.
Output: An equivalent left factored grammar.

Method: for each nonterminal A, find the prefix α common to two or more of its alternatives. If $\alpha \neq \epsilon$ i.e. there is nontrivial common prefix, replace all of the A productions $A \rightarrow \alpha\beta_1 | \alpha\beta_2 | \cdots | \alpha\beta_n | \gamma$ where γ represents all alternatives that do not begin with α, by

$$A \rightarrow \alpha A' | \gamma$$
$$A' \rightarrow \beta_1 | \beta_2 | \cdots | \beta_n$$

A' is new nonterminal. Repeatedly apply this transformation until no two alternatives for a nonterminal have a common prefix.

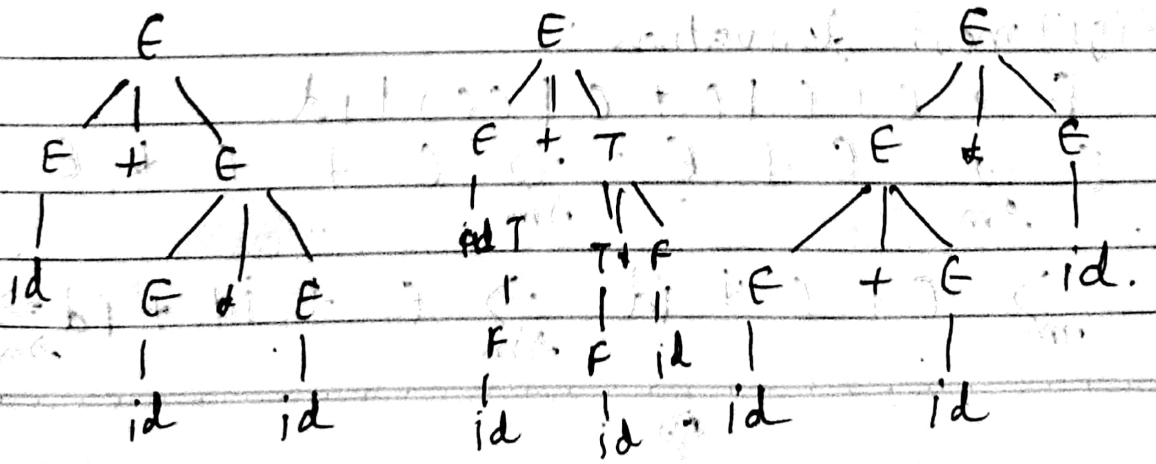
4 Ambiguity :

→ A grammar that produces more than one parse tree for same sentence is said to be ambiguous

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

derivation: $E \rightarrow E+E \mid E*E \mid (E) \mid id$.

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



Sentential form : is any string derivable from the Start symbol or any string that can be derived from S.

Derivation:

1 In left most derivations, the left most nonterminal in each sentential is always chosen. If $\alpha \Rightarrow \beta$ is a step in which left most non terminal in α is replaced, we write $\alpha \underset{lm}{\Rightarrow} \beta$.

2 In right most derivations, the rightmost non-terminal is always chosen. we write $\alpha \underset{rm}{\Rightarrow} \beta$ in this case.

→ $E \rightarrow E+E \mid E*E \mid -E \mid (E) \mid id$.

for string $-(id+id)$.

$E \underset{lm}{\Rightarrow} -E \underset{lm}{\Rightarrow} -(E) \underset{lm}{\Rightarrow} -(E+E)$

$\Rightarrow -(id+E) \underset{lm}{\Rightarrow} -(id+id)$.

If $S \overset{*}{\underset{lm}{\Rightarrow}} \alpha$, then we say that α is a left sentential form of the grammar at hand.

→ Right most derivations are sometimes called <u>canonical derivations</u>.

Right most derivation:

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

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

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

Eliminating Ambiguity :

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

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

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

→ There are two causes of ambiguity in grammar in following example.

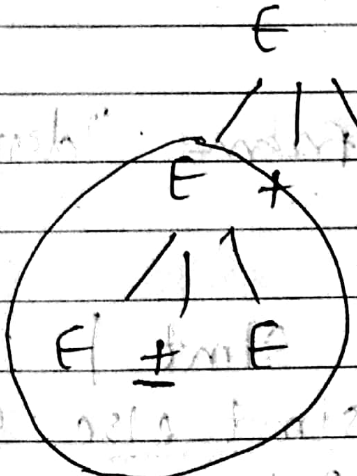
1 The precedence of operators is not respected.

2 associativity is not maintained in grammar

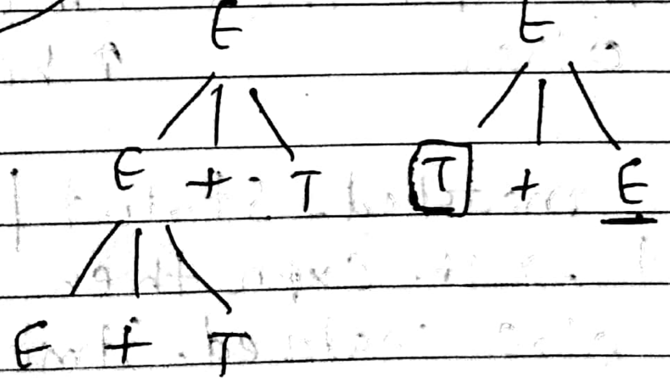
high priority → farther from root :

for + : Which is left associative :

$E \rightarrow E + E + E$



→ So Expand from left not from right. So, E + T, not T + E.



$\underline{E} \rightarrow \underline{E} + T$

left recursive

For $\uparrow$: which is right associative :

2^{3^2} $\quad 2 \uparrow \overline{3 \uparrow 2}$

$2 \uparrow 9$

$F \uparrow F$: $\underline{F \uparrow P}$ ×

$\underline{F} \Rightarrow P \uparrow \underline{F}$ right recursive.

$E \to E+E \mid E*E \mid E \uparrow E \mid (E) \mid id$.

1) $\left.\begin{array}{l} E \to E+E \mid T \\ T \to T*T \mid F \\ F \to F \uparrow F \mid P \\ P \to (E) \mid id \end{array}\right\}$ precedence order.

2) $\left.\begin{array}{l} E \to E+T \mid T \\ T \to T*F \mid F \\ F \to P \uparrow F \mid P \\ P \to (E) \mid id. \end{array}\right\}$ associativity : left or right recursive.

↑ : Right associative.

Remove ambiguity from grammar :

1) bExp → bExp or bExp | bExp and bExp |
not bExp | True | False.

→ bExp → bExp or F | F
F → F and G | G
G → NOT G | True | False.

2) R → R+R | RR | R* | a | b | c.
E → E+T | T
T → TF | F
F → F* | a | b | c.

3) E → E+E | E-E | E/E | E$E | id.
E → E+T | E-T | T
T → T/F / F
F → G $ F | G ($: right associative).
G → id.

eliminate ambiguity from "dangling else" grammar:

stmt → if expr then stmt | if expr then stmt else stmt | other

stmt → open-stmt | matched-stmt
open-stmt → if expr then stmt | if expr then matched-stmt else open-stmt | other

matched-stmt → if expr then matched-stmt else matched-stmt | other.

Recursive descent parser :

A parser that uses Collection of recursive procedures for parsing the given input string is called RD parser.

In this parser CFG is used to build the recursive routines. The R.H.S of production rule is directly Converted to a program for each.

example :

1) $E \to iE'$

$E' \to +iE' \mid \epsilon$

```
E ( )
{
if (lookahead = = 'i')
 {
  match('i');
   E'( );
 }
 else
 { error;
  }
 if (lookahead = = '$')
  {
 printf ("success");
  }
 }.
```

```
E'( )
{
if (lookahead == '+')
 {
  match('+');
  match('i');
  E'();
 }
 else
  return;
 }
match (token t)
{
if (lookahead == t)
{
 lookahead = next-token;
}
else
{
printf("error");
}
```

2) $S \to Aa \mid bAc \mid bBa$

$A \to d$

$B \to d.$

left factoring :

$S \to bS' \mid Aa$

$S' \to Ac \mid Ba$

$A \to d$

$B \to d.$

```
S()
{
 if(lookahead == 'b')
 {
  match('b');
  S'();
 else
 {
  A();
 if(lookahead == 'a')
 {
 match('a');
 }
 if(lookahead == '$')
 {
 printf("success");
 }
 }
```

```
S'()
{
  A();
if (lookahead == 'c')
  {
  match('c');
  }
else if (lookahead == 'a')
 {
  match('a');
 }
}
A()
{
 if (lookahead == 'd')
  {
   match('d');
  }
 B()
 {
 if (lookahead == 'd')
  {
 match('d');
  }
 }
match (token t)
{
if (lookahead == t)
 {
 lookahead == next-token;
 }
```

Scanned by CamScanner

```
else
 {
printf("erros");
 }
}.
```

prepared by kinjal patel

Advantage of RDP:

1) RD Parsers are Simple to build.
2) RD parsers can be Constructed with the help of parse tree.

limitations of RDP:

1) RD parsers are not very efficient as compared to other parsing techniques.
2) There are chances that the program for RD parser may enter in an infinite loop for some input.
3) RD Parser can not provide good error messaging.

Basic steps for construction of RD parser:

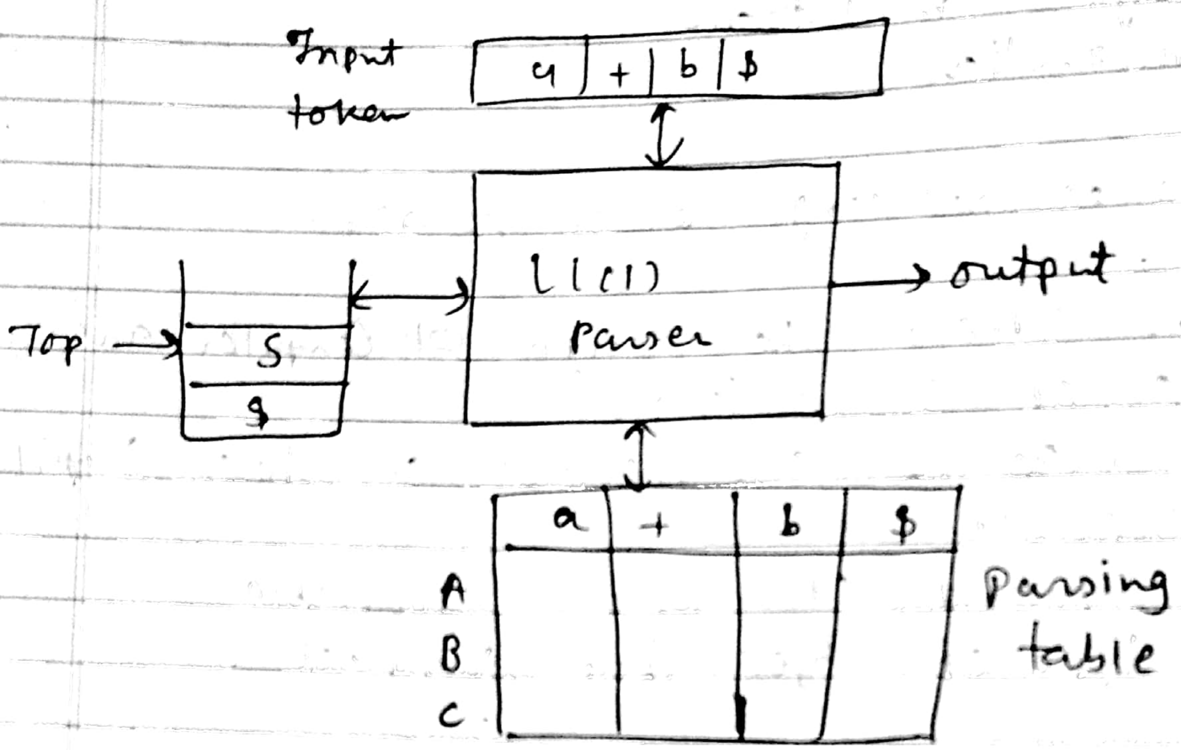
1) If the input symbol is NT then a call to the procedure corresponding to the NT is made.

2) If the i/p symbol is T then it is matched with the look ahead from input. The look ahead pointer has to be advanced on matching of the input symbol.

3) If the production rule has many alternates then all these alternates has to be combined into a single body of procedure.

4) The parser should be activated by a procedure corresponding to the start symbol.

predictive LL(1) parser : non recursive predictive parser : Table driven parser.



→ This top down parsing algorithm is of non-recursive type.

→ In this type of parsing a table is built.

→ for LL(1) -

the first L means : the input is scanned from left to right.

The second L means : it uses left most derivation for input string.

And number 1 in the input symbol means it uses only one input symbol (look ahead) to predict the parsing process.

→ The data structure used by LL(1) are:

1) Input buffer
2) Stack
3) Parsing table

→ LL(1): uses Input buffer to store the input tokens.

→ Stack : is used to hold left sentential form. The Symbols in R.H.S of rule are pushed into stack in reverse order.

→ Parsing table : is basically two dimensional array. The table has row for non terminal and Column for terminals. Table Can be represented as M[A, a] Where A is non terminal and a is current input symbol.

Top of stack	Input token	Parsing action.
$	$	parsing Successful
a	a	pop a and advance lookahead to next token.
a	b	Error
A	a	Refer table M[A, a] if entry at M[A, a] is error, report error
A	a	Refer table M[A, a] if entry at M[A, a] is A → PQR then pop A then push R, then push Q, then push P.

→ The parser consults the table M[A, a] each time while taking the parsing actions hence this type of parsing method is called table driven parsing algorithm.

→ The configuration of LL(1) parser can be defined by top of the ~~token~~ stack and lookahead token.

→ Then at each step of the parse, the action taken depends on the top grammar symbol.

1) If the symbol on top of the stack is a T, then if the current i/p lookahead matches on top of the stack, then the stack symbol is popped, and the input is advanced. If the terminal on top of the stack does not match the current lookahead then an error is signalled.

2) If the symbol on top of the stack is NT, then it is popped off the stack. ~~If~~ The NT is used in conjuction with current lookahead token to predict a rule to be used for that NT. If no rule can be predicted, then an error is signalled. The grammar symbols on the RHS of the predicted rule are pushed on to the stack in reverse order. i.e. first symbol on the RHS is pushed last. The i/p is not affected in any way.

1 Stack : T — Input : T → popped.

2 Stack : NT — Input : (T) → x error → Rule.

PAGE No.
DATE

for Construction of predictive LL(1) Parser we have to follow the following Steps:

1 Computation of FIRST and follow function
2 Construct the predictive parsing table using first and follow functions.
3 parse the Input string with the help of predictive parsing table.

1 First ( ) :

$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \varepsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \varepsilon$
$F \rightarrow (E) \mid id$

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

Following are the rules used to Compute the first functions:

1 $E \rightarrow TE'$
first (E) → first (TE')
→ If T terminal then in first (E) add T.
2 → If T is non terminal then to find first (E) check the production of T
$T \rightarrow FT'$ again $F \rightarrow$ Non terminal.

Scanned by CamScanner

So, check the production of F.

$F \to \underline{(}E) \mid \underline{id}$.

$first(E) = first(TE') = \{(, id\}$

Then check Is T produce ε, for this example No. but if T produce ε then find the ~~production~~ first of E'.

$\underline{E'} \to +TE' \mid \underline{\varepsilon}$ → here E' produce ε

so → When $E \to \underset{\varepsilon}{\underline{E'}}\,\underset{First(T)}{\underline{T}}$

follow :

The rules for Computing follow function are as given below:

1 for the Start Symbol S place \$ in follow(S).

2 $A \to \alpha B\beta$ If β is NT.

first(β) without ε

$A \to \alpha B\beta$ $\beta = T$

add β in follow(B)

3 $A \to \alpha B$ or $A \to \alpha B\underline{\beta}$

follow(A)

β produces $\wedge$

then follow(A) into follow of B.

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

LL(1) parsing table :

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

→

ex. $S \to (L) \mid a$
9th15 $L \to L, S \mid S$

Construct the predictive parser for the following grammar

→ $S \to (L) \mid a$
$L \to SL'$
$L' \to , SL' \mid \epsilon$

} after eliminate the LR.

→ first (S) = { (, a }
first (L) = { (, a }
first (L') = { , , ϵ }

follow(S) : { , , $,) }
follow(L) : {) }
follow(L') : {) }

	a	(	)	,	$
S	$S \to a$	$S \to (L)$			
L	$L \to SL'$	$L \to SL'$			
L'			$\underline{L' \to \varepsilon}$ follow.	$L' \to ,SL'$	

grammar is ~~not~~ LL(1).

$E \to TE'$

$E' \to +TE' \mid \varepsilon$

$T \to FT'$

$T' \to *FT' \mid \varepsilon$

$F \to (E) \mid id$.

parse the string: id + id * id $

Stack	Input	Action
$ E	id + id * id $	[illegible]
$ E'T	id + id * id $	$E \to TE'$
$ E'T'F	id + id * id $	$T \to FT'$
$ E'T'id	id + id * id $	$F \to id$
$ E'T'	+ id * id $	matched
$ E'	+ id * id $	$T' \to \varepsilon$
$ E'T+	+ id * id $	$E' \to +TE'$
$ E'T	id * id $	matched.
$ E'T'F	id * id $	$T \to FT'$
$ E'T'id	id * id $	$F \to id$
$ E'T'	+ id $	matched.
$ E'T'F*	* id $	$T' \to *FT'$
$ E'T'F	id $	matched.
$ E'T'id	id $	$F \to id$
$ E'T'	$	matched
$ E'	$	$T' \to \varepsilon$
$	$	$E' \to \varepsilon$
$	$	Success.

QB. 1

(7 marks)

$S \rightarrow AB$

$A \rightarrow Ca \mid \varepsilon$

$B \rightarrow BaAC \mid c$

$C \rightarrow b \mid \varepsilon$

1) Remove left Recursion from given grammar.

$S \rightarrow AB$

$A \rightarrow Ca \mid \varepsilon$

$B \rightarrow cB' \mid \varepsilon$

$B' \rightarrow aACB' \mid \varepsilon$

$C \rightarrow b \mid \varepsilon$.

2) first (S) = { b, ε, a, c }

first (A) = { b, ε, a }

first (B) = { c }

first (B') = { a, ε }

first (C) = { b, ε }

3) follow (S) = { \$ }

follow (A) = { c, b, a, \$ }. ~~follow(C)~~ first(B') follow(B)

follow (B) = { \$ }

follow (B') = { \$ }

follow (C) = { a, \$ }

Scanned by CamScanner

) LL(1) parsing table:

	a	b	c	\$
S	$S \to AB$	$S \to AB$	$S \to AB$	$S \to \varepsilon$
A	$A \to Ca$ $A \to \varepsilon$	$A \to Ca$ $A \to \varepsilon$	$A \to \varepsilon$	$A \to \varepsilon$
B			$B \to cB'$	
B'	$B' \to aACB'$			$B' \to \varepsilon$
C	$C \to \varepsilon$	$C \to b$		$C \to \varepsilon$

grammar is not LL(1).

QB-7

$S \to AcB \mid cbB \mid Ba$ (6 Marks)

$A \to da \mid BC$

$B \to g \mid \varepsilon$

$C \to h \mid \varepsilon$

first (S) = $\{d, g, \varepsilon, h, c, a\}$

first (A) = $\{d, g, \varepsilon, h\}$

first (B) = $\{g, \varepsilon\}$

first (C) = $\{h, \varepsilon\}$

follow (S) = $\{\$\}$

follow (A) = $\{c\}$

follow (B) = $\{\$, a, h, \underset{\text{follow (A)}}{c}\}$

follow (C) = $\{c\}$

Q-B Construct the behaviour of the parser on the sentence (a, a) using the grammar specified above.

$S \to (L) \mid a.$
$L \to SL'$
$L' \to , SL' \mid \epsilon.$

Stake	Input	Actions
\$ S	(a, a) \$	
\$) L (	(a, a) \$	S → (L)
\$) L	a, a) \$	matched.
\$) L' S	a, a) \$	L → SL'
\$) L' a	a, a) \$	S → a
\$) L' S ,	, a) \$	L' → , SL'
\$) L' S	a) \$	matched
\$) L' a	a) \$	S → a
\$) L'	) \$	matched
\$)	) \$	L' → ε
\$	\$	~~matched~~ Accept
~~\$~~	~~\$~~	~~Accept~~.

Ex: →

$D \to TL;$
$L \to L, id \mid id$
$T \to int \mid float$

$D \to TL;$
$L \to id L'$
$L' \to , id L' \mid \epsilon$
$T \to int \mid float$

→ first (D) = { int, float }
first (L) = { id }
first (T) = { int, float }
first (L') = { , , ε }

follow :

follow(D) = { $ }

follow(L) = { ; }

follow(L') = { ; }

follow(T) = { id }

Parsing table :

	id	,	int	float	;	$
D			D → TL;	D → TL;		
L	L → idL'					
L'		L' → ,idL'			L' → ε	
T			T → int	T → float		

QB-15

$S \to aBDh$

$B \to cC$

$C \to bC \mid \varepsilon$

$D \to EF$

$E \to g \mid \varepsilon$

$F \to f \mid \varepsilon$

first (S) : { a }
first (B) : { c }
first (C) : { b , ε }
first (D) : { g , f , ε }
first (E) : { g , ε }
first (F) : { f , ε }

follow (S) : { $ }
follow (B) : { g , f , h }
follow (C) : { g , f , h }
follow (D) : { h }
follow (E) : { f , h }
follow (F) : { h } .

parsing table :

	a	b	c	g	f	h	$
S	$S \to aBDh$						
B			$B \to cC$				
C		$C \to bC$		$C \to \varepsilon$	$C \to \varepsilon$	$C \to \varepsilon$	
D				$D \to EF$	$D \to EF$	$D \to \varepsilon$	
E				$E \to g$	$E \to \varepsilon$	$E \to \varepsilon$	
F					$F \to f$	$F \to \varepsilon$	

grammar is LL(1).

Q.8 Transition diagram for Predictive Parser:

10 → Transition diagrams for predictive parsers for the following grammar.

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \varepsilon$

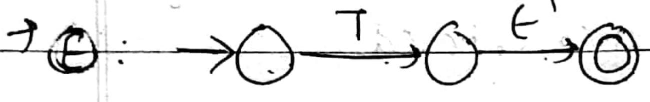
$F \rightarrow (E) \mid id$

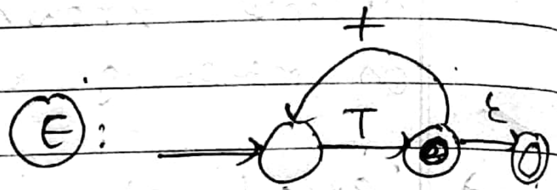
$E \rightarrow TE'$ ($E' \rightarrow \varepsilon$)

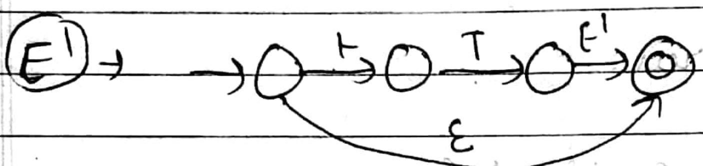
$T+TE'$

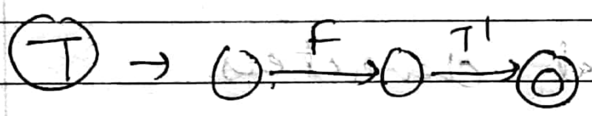
$T+T+TE'$

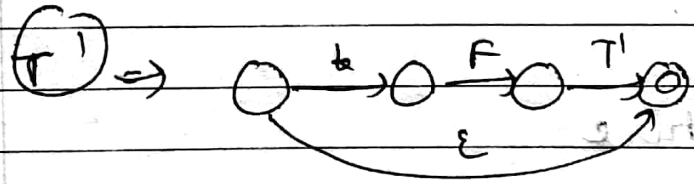
$T+T+T+\ldots\varepsilon$

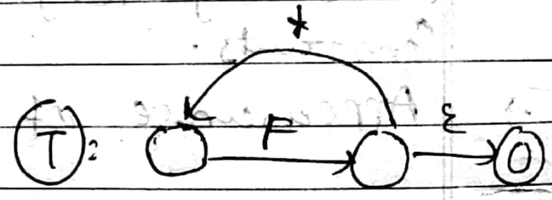


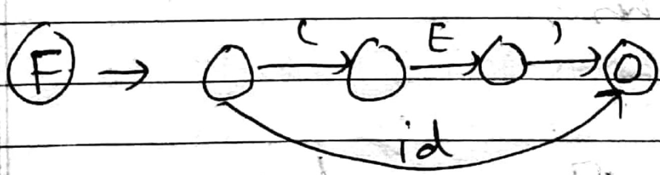










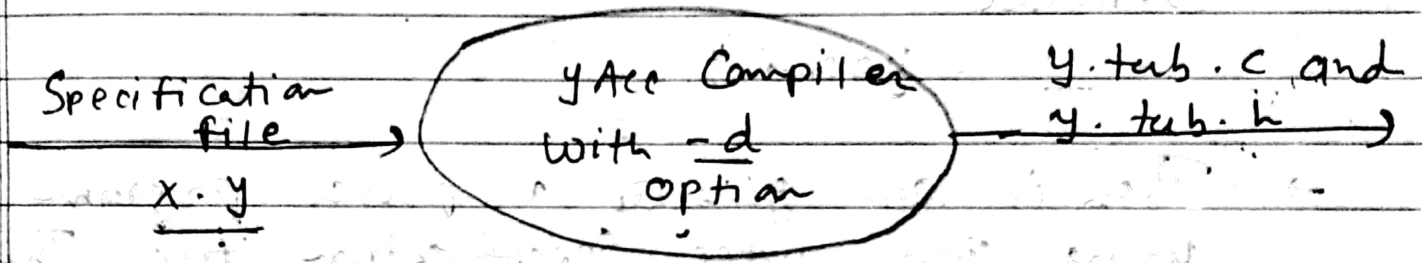


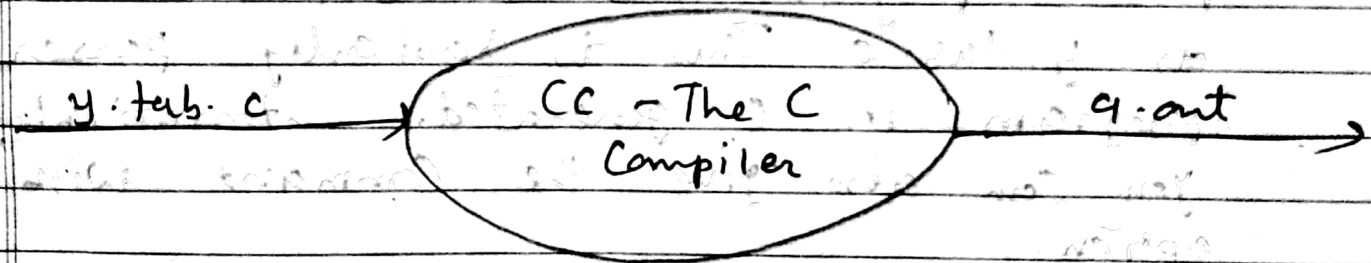
PAGE No.
DATE

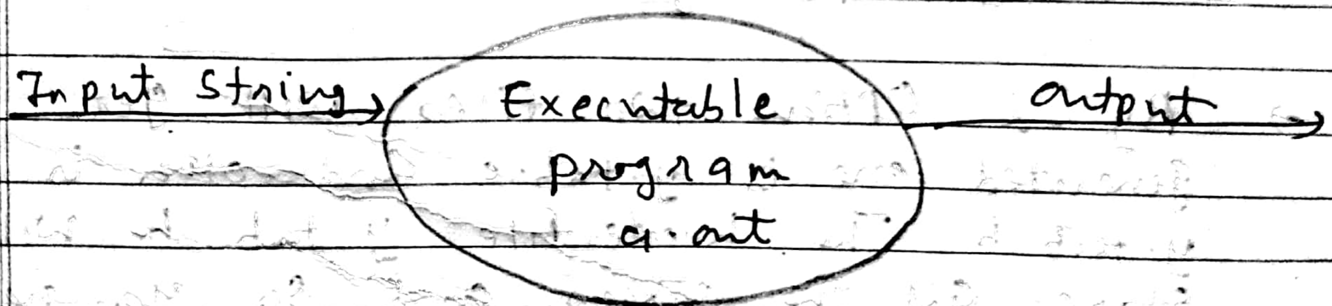
Parser generators:

→ Certain automation tools for parser generation are available. YACC is one such automatic tool for generating the parser program.

→ YACC stands for Yet another Compiler Compiler which is basically the utility available from UNIX.







→ Basically YACC is LALR parser generator.
→ The YACC can report conflicts or ambiguities in the form of error messages.
→ The LEX tool is for lexical Analyzer.

→ LEX and YACC work together to analyse the program syntactically.

→ The typical YACC translator can be represented as shown in fig.

→ First we write a YACC specification file. Let us name it as x.y. This file is given to the YACC compiler by UNIX command

yacc x.y.

→ Then it will generate a parser program using your YACC specification file. This parser program has a standard name as y.tab.c. This is basically parser program in C generated automatically. You can also give the command with -d option.

yacc -d x.y.

→ By -d option two files will get generated one is y.tab.c and other is y.tab.h. The headerfile y.tab.h will store all the tokens and so you need not have to create y.tab.h explicitly

→ The generated y.tab.c program will then be compiled by C compiler and generates the executable a.out file. Then you can test your YACC program with the help of some valid and invalid strings.

Automatic generation of parsers:
yacc specification file consists of three parts:
1 Declaragtion section:
2 Translation rule:
3 Supporting C functions:

1 Declaration section:
In this section ordinary C declarations can be put. Not only this we can also declare grammar tokens in this section.
The declaration of tokens should be within
%{
%}

2 The translation rule section:
It consist of all the procedure rules of CFG with corresponding actions.

3 C functions section: This section consists of one main function in which the routine yyparse() will be called. And it also consists of required C functions.