classmate
Date
Page

2.1 Role of Syntax Analyzer or Parser

The analysis phase of a compiler breaks up a source pgm into constituent pieces and produces an intermediate representation for it, called intermediate code. The synthesis phase translates the intermediate code into the target pgm.

Analysis is organized around the "syntax" of the language to be compiled. The syntax of a pgming language describes the proper form of its pgms, while the semantics of the language defines what its pgms mean, that is what each pgm does when it executes

- Context-free grammars or BNF (Backus Naur Form) are widely used for specifying syntax

- Informal descriptions and suggestive examples are used for specifying semantics.

- The parser obtains a string of tokens from the lexical analyzer and verifies that 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.

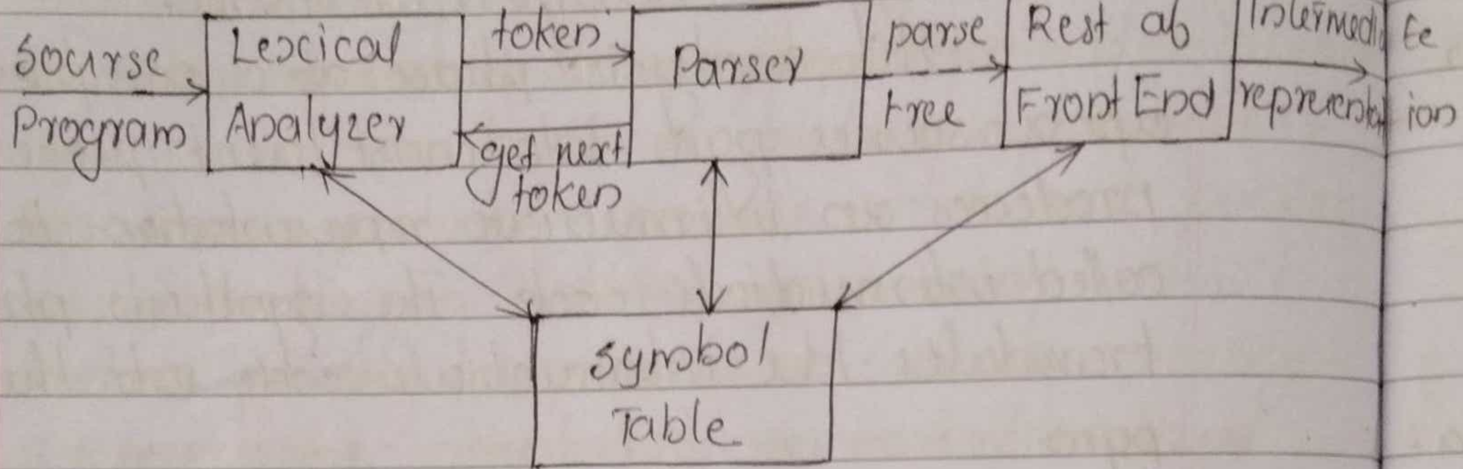


Fig: position ab parser in compiler model.

- The parser should report any syntax error and recover brom commonly occuring errors to continue processing the remainder ab the pgm.

A grammar generally describes the hierarchical structure ab most pgming language constructs.

-There are three general types ab parsers bor grammars.

1. Universal - can parse any grammar. Too inebbicient to use in production compilers.
2. Top-down - build parse tree brom top (Root) to the bottom (leaves) moves from top to leaves
3. Bottom-up - start brom the leaves and work their way up to the root. moves from leaves to root

The I/p to the parser is scanned brom left to right one symbol at a time.

Definition of Grammars / Context-free grammar.

A context-free grammar has four components (V, T, P, S).

1) A set of non terminal symbols, sometimes referred to as tokens. Tokens are elementary symbols of the language defined by the grammar

2. A set of ~~non~~terminals - sometimes called syntactic variables. Each nonterminal represents a set of strings of terminals (eg: stmt expr).

3. Productions, where each pdn consists of a non-terminal, called the head or leftside of the pdn, an arrow (→ or ::=), and a sequence of terminals or nonterminals, called body or right side of the pdn.

4. Start symbol. One non-terminal is designated as the start symbol

- Grammars are specified by listing their pdns, with the pdns for the start symbol listed first

eg.
list → list + digit
list → list - digit
list → digit
digit → 0|1|2|3|4|5|6|7|8|9

can be also written as
list → list+digit | list-digit | digit

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, - are terminals.

here, list, digit are nonterminals

- A grammar derives strings by beginning with the start symbol and repeatedly replacing a nonter~~mat~~minal by the body of a pdn for that nonterminal.

eg: Consider the context-free grammar

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

Show how the string aa+a* can be generated by this grammar

Ans:

$$S \rightarrow SS*$$
$$\rightarrow SS+S*$$
$$\rightarrow aS+S*$$
$$\rightarrow aa+S*$$
$$\rightarrow aa+a*$$

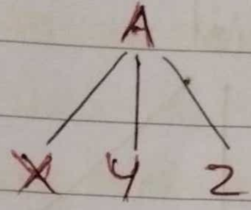
Parsing is the pblm of taking a string of terminals and figuring out how to derive it from the start symbol of the grammar, and if it cannot be derived from the start symbol of the grammar then reporting syntax errors within the string

Parse trees

- A parse tree pictorially shows how the start symbol of a grammar derives a string in the language.

eg: If nonterminal A has a pdn

$A \rightarrow XYZ$, then the parse tree may have an intirior node labeled A with three children labeled x, y and z brom left to right



Thus, given a context bree grammer, a parse tree according to grammer is a tree with the bollowing properties:

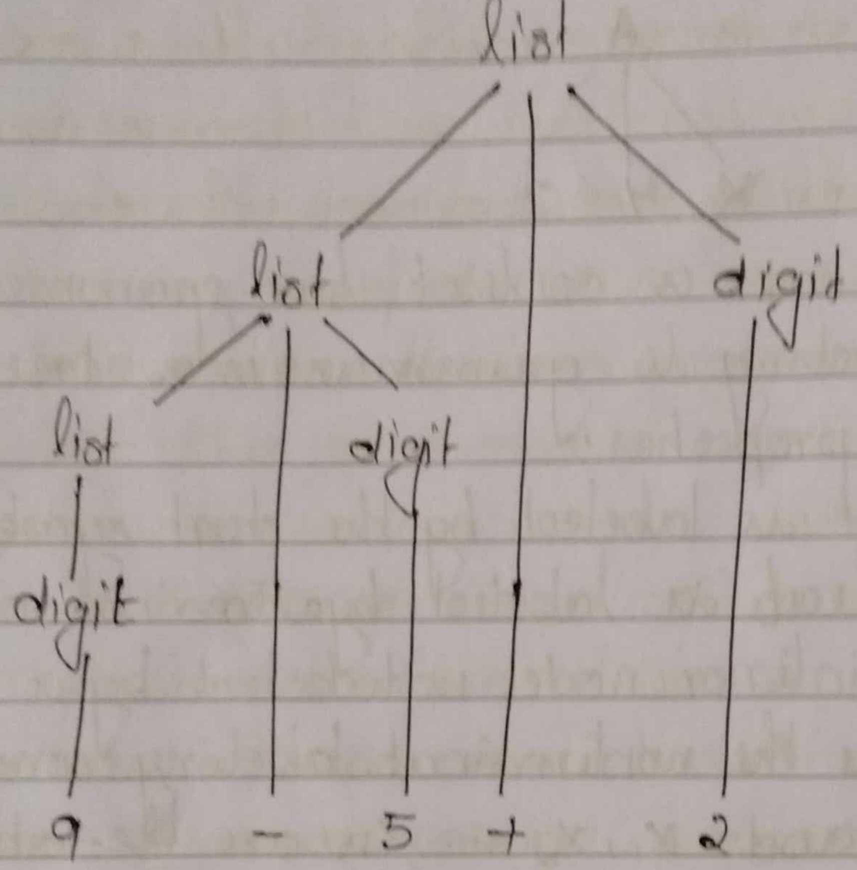
1. The root is labeled by the start symbol
2. Each leab is labeled by a terminal or ϵ
3. Each interior node is labeled by a nonterminal
4. If A is the nonterminal labeling some interior nodes and $x_1, x_2 \cdots, x_n$ are the labels ob the children ob that node brom left to right, then there must be a pdn $A \rightarrow x_1 x_2 \cdots x_n$. Here, $x_1, x_2 \cdots, x_n$, each stand bor a symbol that is either a terminal or a nonterminal.

If $A \rightarrow \epsilon$ is a pdn, then a node labed A mey have a single child labeled ϵ.

Consider the bollowing pdns,

list → list + digit
list → list - digit
list → digit
digit → 0|1|2|3|4|5|6|7|8|9

Parse tree bor 9-5+2 according to the gramm



- The bollowing grammar is used bor simple arithmeti expressions:

expression → expression + term
expression → expression - term
expression → term
term → term * bactor
term → term / factor
term → bactor
factor → (expression)
factor → id

expression, bactor and term are non terminals
expression is the start symbol.
id, + - * ( ) - terminals.

Notalional connentions

The bollowing symbols are terminals

a) lowercase letters such as a, b, c etc

b) operator symbols such as +, −, * etc

c) punctuation symbol such as parentheses, comma -

d) digits such as 0, 1, . . . , 9

e) Boldbace strings such as id, or if

The bollowing symbol are nonterminals

a) uppercase letters such as A, B, C . .

b) letter s is usually the start symbol

c) lowercase italic names such as expr or stmt

~~d)~~ when discussing pgming construct, uppercase letters may be used to represent nonterminals ~~eg: expr, term~~

eg: expr, term → E, T resp.

So the above grammer can be replaced by

$$E \rightarrow E+T \mid E-T \mid T$$

$$T \rightarrow T*F \mid T/F \mid F$$

$$F \rightarrow (E) \mid \mathbf{id}.$$

here, E, T and F are nonterminals, with E as the start symbol. The remaining symbols are terminals.

Derivations:

Productions are treated as rewriting rules. Beginning with the start symbol, each rewriting step replaces a nonterminal by body ab one ab its pdns.

Eg. Consider, the bollowing grammar, with a single nonterminal E

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

The bollowing sequence represents the derivation ab -(id) brom E

$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(id)$

Such a sequence ab replacement is called derivation ab -(id) brom E. This derivation provides a proof that the string -(id) is one particular instance ab an expression.

Debinition ab derivation

Consider a nonterminal A in the middle ab a sequence ab grammar symbols, as in $\alpha A\beta$ where α and β are arbitrary strings ab grammar symbols. Suppose $A \rightarrow \gamma$ is a pdn. then we can write $\alpha A\beta \Rightarrow \alpha\gamma\beta$.

The symbol $\Rightarrow$ means "derives in one step"

eg: $E \Rightarrow -E$, $-E \Rightarrow -(E)$

$\overset{*}{\Rightarrow}$ means derivation in zero or more steps

eg: $E \overset{*}{\Rightarrow} -(id)$, $E \overset{*}{\Rightarrow} E$, $E \overset{*}{\Rightarrow} (E)$

$\xrightarrow{+}$ means derivation in one or more steps.

eg: $E \xrightarrow{+} -(id)$

Sentential form:

A sentential form may contain both terminals & nonterminals, and may be empty.

eg: $S \xRightarrow{*} \alpha$, s - start symbol

α - sentential form of G

A sentence of G is a sentential form with no non terminals. The language generated by a grammar is its set of sentences.

Thus, ω is in $L(G)$, iff ω is a sentence of G (or $S \xRightarrow{*} \omega$)

A language that can be generated by a grammar is said to be a context-free language. If two grammars generate the same language, the grammars are said to be equivalent.

- The string $-(id+id)$ is a sentence of grammar because there is a derivation

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(id+E) \Rightarrow -(id+id)$$

The strings $E, -E, -(E), \ldots, -(id, id)$ are all sentential forms of this grammar

we write $E \xRightarrow{*} -(id+id)$ to indicate that $-(id+id)$ can be derived from E.

LMD $E \Rightarrow E*E$
$\Rightarrow E+E*E$
$\Rightarrow id+E*E$
$\Rightarrow id+id*E$
$id+id*id$

$E*E$
$\Rightarrow E*E$
$\Rightarrow E*id$
$\Rightarrow E+E*id$
$\Rightarrow E+id*id$
$\Rightarrow id+id*id$

- There are two types ab derivations.

1. Leftmost derivation: For each derivation the leftmost nonterminal in each sentential form is replaced. If $\alpha \Rightarrow \beta$ is a step in which the leftmost nonterminal in α is replaced we write $\alpha \underset{lm}{\Rightarrow} \beta$

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

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

2. Rightmost derivation: For each derivation, the rightmost nonterminal in each sentential form is replaced, we write $\alpha \Rightarrow \beta$ in this case.

eg: $E \underset{rm}{\Rightarrow} -E \underset{rm}{\Rightarrow} -(E+E) \underset{rm}{\Rightarrow} -(E+id) \underset{rm}{\Rightarrow} (id+id$

Rightmost derivations are sometimes called canonical derivations.

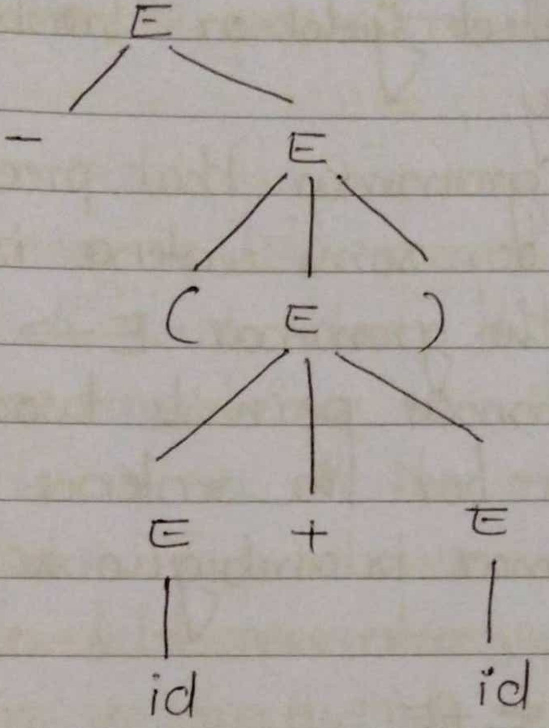
Parse Trees and Derivations:

- A parse tree is a graphical representation ab a derivation that filters out the order in which productions are applied to replace nontermi

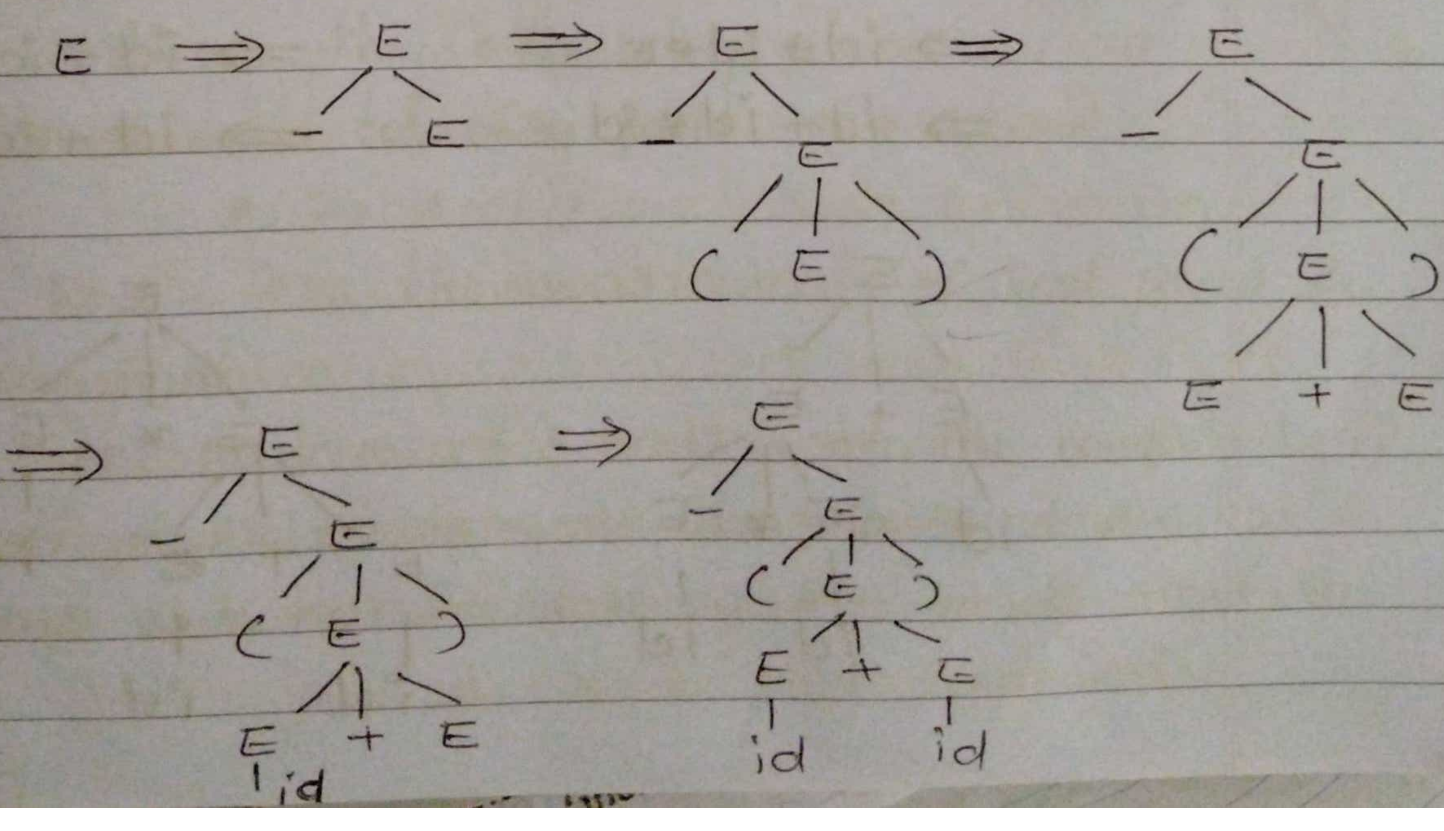
- Each interior node ab parse tree represents the application ab a pdn.

- The children ab the node are labeled, brom left to right, by the symbols in the body ab the pdn

eg. parse tree bor -(id+id)



Sequence ab parse trees bor -(id+id)

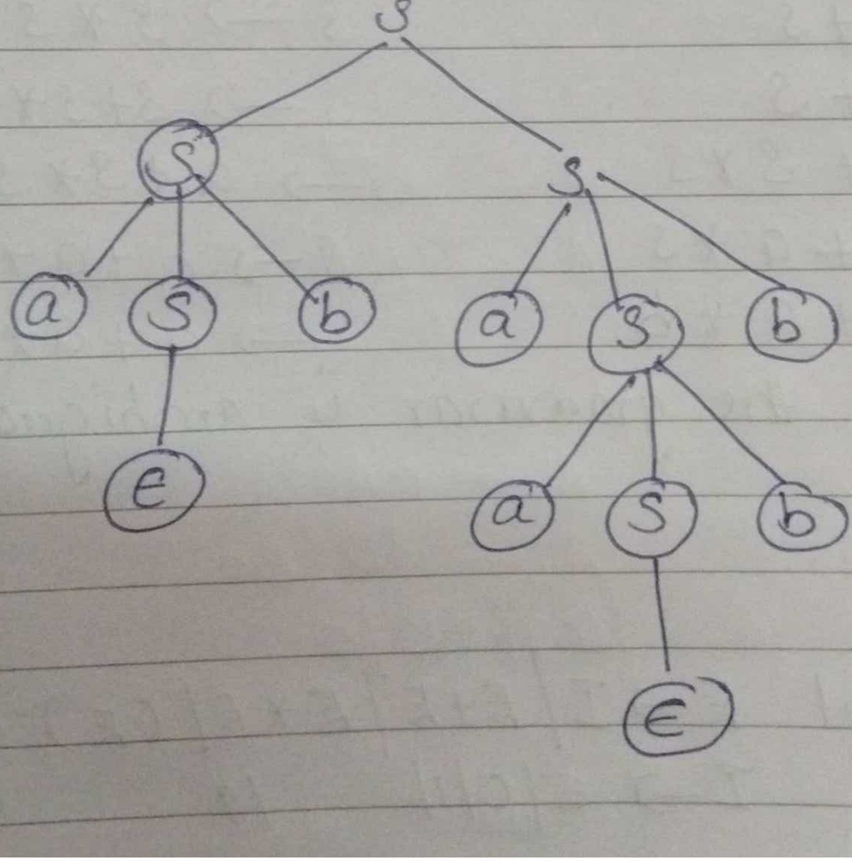


Example for Parse Tree

CFG = ({S}, {a, b}, {P = S → SS | aSb | ∈}, {S})

find the derivation and parse tree for the string abaabb.

S → SS → aSbS → abS → abaSb → abaaSbb → abaabb



Ambiguous Grammar

A grammar is said to be ambiguous if there exist two or more derivation tree for a string w.

Example G = ({S}, {a+b, +, *} P, S}

where P consists of $S \rightarrow S+S \mid S*S \mid a/b$
the string a+a*b can be generated as

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

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

Thus the grammar is ambiguous.

Eliminating left recursion

~~α~~

Let G be a context ~~$A \to \alpha . B\beta,\ a/b$~~
free grammar. A production ~~$B \to \gamma .\ c/d.$~~
of G is said left recursive if it has the form ~~$First(\beta) = \{c,d\}$~~

$$A \to A\alpha$$

where A is a non-terminal and ~~$B \to \gamma .\ a/b.$~~
α is a string of grammar symbols.

→ $A \to A\alpha / \beta$ by $A \to \beta A'$
$A' \to \alpha A' / e$

eg.

$E \to E+T / T$
$T \to T*F / F$
$F \to (E) \mid id$

$E \to TE'$
$E' \to +TE' / e$
$T \to FT'$
$T' \to *FT' / e$
$F \to (E) \mid id$

Eliminating left factoring

$A \to \alpha\beta_1 \mid \alpha\beta_2$ are two A productions

$\Downarrow$

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

① $S \to iEtS \mid iEtSeS \mid a$

$E \to b$

$\Downarrow$

$S \to iEtSS' \mid a$

$S' \to eS \mid \epsilon$

$E \to b$

② ~~$A \to aA'$~~

~~$A' \to$~~

$A \to aAB \mid aBc \mid aAc$

$\Downarrow$

$A \to aA'$

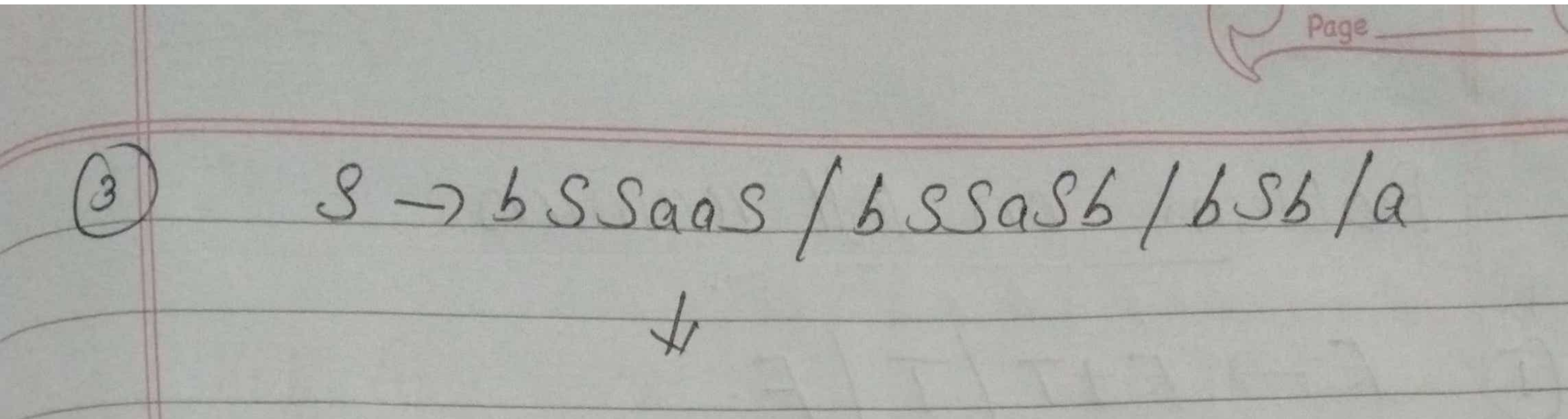
$A' \to AB \mid Bc \mid Ac$

$\Downarrow$

$A \to aA'$

$A' \to AD \mid Bc$

$D \to B \mid c$



(3) $S \rightarrow bSSaaS \mid bSSaSb \mid bSb \mid a$

$\Downarrow$

$S \rightarrow bSS' \mid a$

$S' \rightarrow SaaS \mid SaSb \mid b$

$\Downarrow$

$S \rightarrow bSS' \mid a$

$S' \rightarrow SaA \mid b$

$A \rightarrow aS \mid Sb$

- For most parsers, it is desirable that the grammar be made unambiguous, for if it is not, we cannot uniquely determine which parse tree to select for a sentence.
- In other cases, it is convenient to use carefully chosen ambiguous grammars, together with disambiguating rules that "throw away" undesirable parse trees, leaving only one tree for each sentence.

EXAMPLE

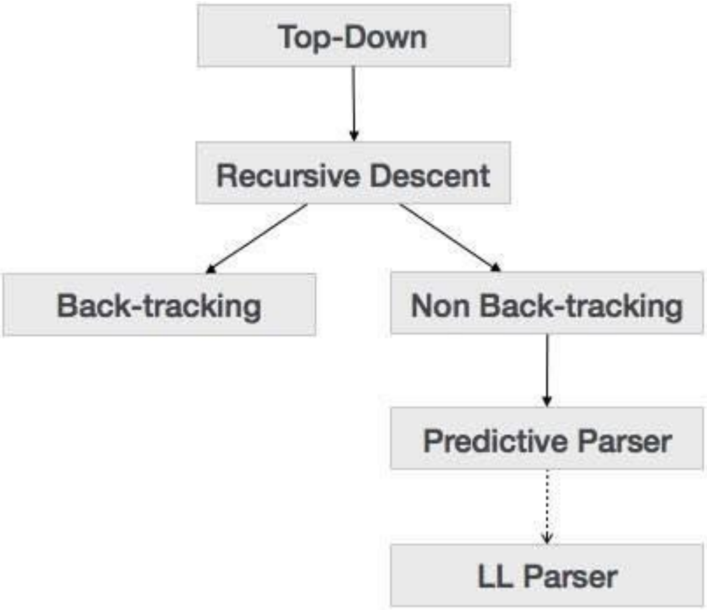
Consider very simple sentence id+ id * id.

```
1st Leftmost Derivation      2nd Leftmost Derivation

E ===> E + E                 E ===> E * E
 ===> id + E                   ===> E + E * E
 ===> id + E * E               ===> id + id * E
 ===> id + id * E              ===> id + id * E
 ===> id + id * id             ===> id + id * id

1st Parse Tree               2nd Parse Tree

    E                                E
   / | \                            / | \
  E  +  E                          E  *  E
  |    / | \                     / | \    |
 id   E  *  E                   E  +  E   id
      |     |                   |     |
      id    id                  id    id
```

2.2 TOP DOWN PARSING

- Parsing is the process of determining if a string of token can be generated by a grammar.
- Mainly 2 parsing approaches:
  - Top Down Parsing
  - Bottom Up Parsing
- In **top down parsing**, parse tree is constructed from top (root) to the bottom (leaves).
- In **bottom up parsing**, parse tree is constructed from bottom (leaves)) to the top (root).
- It can be viewed as an attempt to construct a parse tree for the input starting from the root and creating the nodes of parse tree in preorder.
- Pre-order traversal means: 1. Visit the root 2. Traverse left subtree 3. Traverse right subtree.

- Top down parsing can be viewed as an attempt to find a leftmost derivation for an input string (that is expanding the leftmost terminal at every step).



2.2.1 RECURSIVE DESCENT PARSING

It is the most general form of top-down parsing.

It may involve **backtracking**, that is making repeated scans of input, to obtain the correct expansion of the leftmost non-terminal. Unless the grammar is ambiguous or left-recursive, it finds a suitable parse tree

EXAMPLE

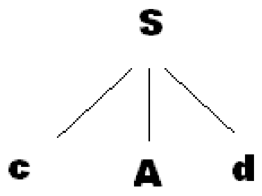
Consider the grammar:

S → cAd

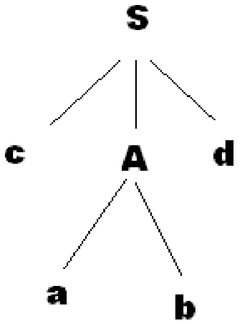
A → ab | a

and the input string w = cad.

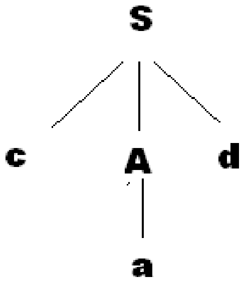
- To construct a parse tree for this string top down, we initially create a tree consisting of a single node labelled **S**.
- An input pointer points to **c**, the first symbol of w. **S** has only one production, so we use it to expand **S** and obtain the tree as:



- The leftmost leaf, labeled **c**, matches the first symbol of input w, so we advance the input pointer to **a**, the second symbol of **w**, and consider the next leaf, labeled **A**.
- Now, we expand **A** using the first alternative **A** → **ab** to obtain the tree as:



- We have a match for the second input symbol, **a**, so we advance the input pointer to **d**, the third input symbol, and compare d against the next leaf, labeled **b**.
- Since **b** does not match **d**, we report failure and go back to **A** to see whether there is another alternative for **A** that has not been tried, but that might produce a match.
- In going back to **A**, we must reset the input pointer to position 2 , the position it had when we first came to **A**, which means that the procedure for **A** must store the input pointer in a local variable.
- The second alternative for **A** produces the tree as:



- The leaf **a** matches the second symbol of **w** and the leaf **d** matches the third symbol. Since we have produced a parse tree for **w**, we halt and announce successful completion of parsing. (that is the string parsed completely and the parser stops).
- The leaf a matches the second symbol of w and the leaf d matches the third symbol. Since we have produced a parse tree for w, we halt and announce successful completion of parsing. (that is the string parsed completely and the parser stops).

2.2.2 PREDICTIVE PARSING

- A predictive parsing is a special form of recursive-descent parsing, in which the current input token unambiguously determines the production to be applied at each step. The goal of predictive parsing is to construct a top-down parser that never backtracks. To do so, we must transform a grammar in two ways:

- **Eliminate left recursion, and**
- **Perform left factoring.**

- These rules eliminate most common causes for backtracking although they do not guarantee a completely backtrack-free parsing (called LL(1) as we will see later).

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.

EXAMPLE

Consider the grammar

A → Aα

A → β

- It recognizes the regular expression **βα***. The problem is that if we use the first production for top-down derivation, we will fall into an infinite derivation chain. This is called left recursion.
- Top–down parsing methods cannot handle left recursive grammars, so a transformation that eliminates left-recursion is needed. The left-recursive pair of productions **A → Aα|β** could be replaced by two non-recursive productions.

A → β A′

A′→ α A′|ε

- Consider The following grammar which generates arithmetic expressions

E → E + T|T

T → T * F|F

F → (E)|id

Eliminating the immediate left recursion to the productions for E and then for T, we obtain

E → T E′

E′ → + T E′|ε

T → F T′

T′ → * F T′|ε

F → (E)|id

- No matter how many A-productions there are, we can eliminate immediate left recursion from them by the following technique. First, we group the A productions as

  $\mathbf{A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n}$

  where no $\boldsymbol{\beta_i}$ begins with an **A**. Then we replace the **A**-productions by

  $\mathbf{A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A'}$

  $\mathbf{A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A' \mid \varepsilon}$

Left Factoring

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

$$\mathbf{A \rightarrow \alpha\beta_1 \mid \alpha\beta_2}$$

- are two A-productions, and the input begins with a non-empty string derived from $\boldsymbol{\alpha}$ we do not know whether to expand **A** to $\boldsymbol{\alpha\beta_1}$ or $\boldsymbol{\alpha\beta_2}$.
- However, we may defer the decision by expanding **A** to $\boldsymbol{\alpha}$**B**. Then, after seeing the input derived from α, we may expand **B** to $\boldsymbol{\beta_1}$ or $\boldsymbol{\beta_2}$.
- The left factored original expression becomes:

$$\mathbf{A \rightarrow \alpha B}$$
$$\mathbf{B \rightarrow \beta_1 \mid \beta_2}$$

- For the “dangling else “grammar:

  stmt →if cond then stmt else stmt |if cond then stmt

  The corresponding left – factored grammar is:

  stmt → if cond then stmt else_clause

  else_clause → else stmt | ε

Non Recursive Predictive parser

- It is possible to build a nonrecursive predictive parser by maintaining a stack explicitly, rather than implicitly via recursive calls.
- The key problem during predictive parsing is that of determining the production to be applied for a nonterminal.
- The nonrecursive parser in looks up the production to be applied in a parsing table

Requirements

1. Stackv
2. Parsing Table
3. Input Buffer
4. Parsing

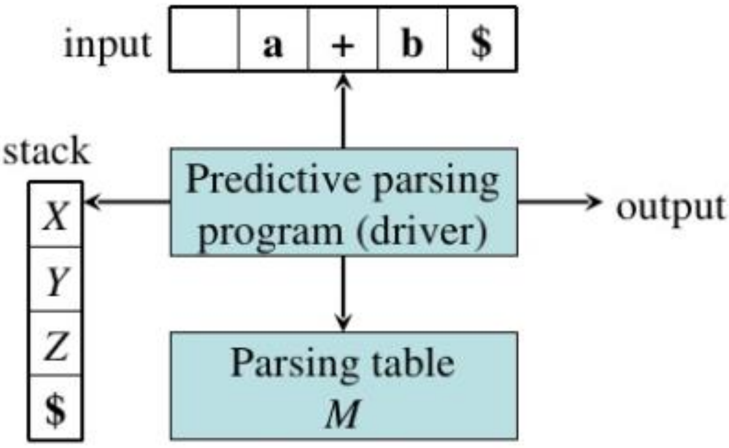


Figure : ***Model of a nonrecursive predictive parser***

- **Input buffer** - contains the string to be parsed, followed by $(used to indicate end of input string)
- **Stack** – initialized with $, to indicate bottom of stack.
- **Parsing table** - 2 D array M[A,a] where A is a nonterminal and a is terminal or the symbol $
- The parser is controlled by a program that behaves as follows. The program considers **X,** the symbol on top of the stack, and **a** current input symbol. These two symbols determine the action of the parser.
- There are three possibilities,
  1. If **X = a = $** , the parser halts and announces successful completion of parsing.
  2. If **X = a ≠ $** , the parser pops **X** off the stack and advances the input pointer to the next input symbol,
  3. If **X** is a nonterminal, the program consults entry **M|X, a |** of the parsing table **M**. The entry will be either an **X**-production of the grammar or an error entry. If, for example, **M |X, u |** = **{X → UVW}**, the parser replaces **X** on top of the stack by WVU (with U on top). As output we shall assume that the parser just prints the production used; any other code could be executed here. If **M|X, a| = error**, the parser calls an error recovery routine.

Predictive Parsing Algorithm

INPUT: A string **w** and a parsing table ***M*** for grammar ***G***.

OUTPUT: If **w** is in **L (G)** , a leftmost derivation of **w**; otherwise, an error indication.

METHOD: Initially, the parser is in a configuration in which it has **$S** on the stack with **S**, the start symbol of **G** on top, and **w$** in the input buffer. The program that utilizes the predictive parsing table M to produce a parse for the input is shown below.

```
set ip to point to the first symbol of w$;
repeat
        let X be the lop stack symbol and a the symbol pointed to by ip;
        if X is a terminal or $ then
                if X = a then
                        pop X from the stack and advance ip
                else error ()
        else   /* X is a nonterminal */
                if M|X, a|= X → $Y_1$ $Y_2$ . . . $Y_i$ then begin
                        pop X from the stack;
                        push $Y_k$, $Y_{k-1}$,. . . , $Y_1$ onto the stack, with $Y_1$ on top;
                        output the production X → $Y_1$ $Y_2$ . . ..$Y_k$
                end
                else error ()
until X = S /* stack is empty */
```

EXAMPLE

Consider Grammar:

E → T E′

E' → +T E' | Є

T → F T'

T' → * F T' | Є

F → (E) | id

Construction Of Predictive Parsing Table

- Uses 2 functions:
    - FIRST()
    - FOLLOW()
- These functions allows us to fill the entries of predictive parsing table

FIRST

- If 'α' is any string of grammar symbols, then FIRST(α) be the set of terminals that begin the string derived from α . If α==*>ϵ then add ϵ to FIRST(α).First is defined for both terminals and non terminals.
- To Compute First Set
    1. If **X** is **a** terminal , then FIRST(**X**) is {**X**}
    2. If **X→ ϵ** then add **ϵ** to FIRST(**X**)
    3. If **X** is a non terminal and $\mathbf{X \rightarrow Y_1Y_2Y_3...Y_n}$, then put '**a**' in FIRST(**X**) if for some **i**, **a** is in FIRST($\mathbf{Y_i}$) and ϵ is in all of FIRST($\mathbf{Y_1}$),...FIRST($\mathbf{Y_{i-1}}$).

EXAMPLE

Consider Grammar:

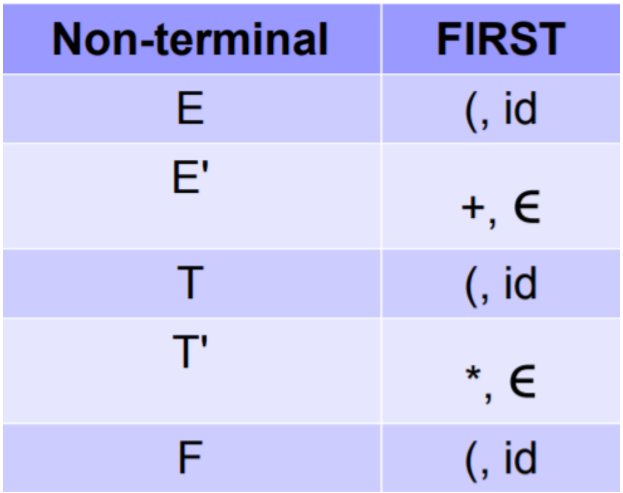
E → T E′

E' → +T E' | Є

T → F T'

T' → * F T' | Є

F → (E) | id



Non-terminal	FIRST
E	(, id
E'	+, Є
T	(, id
T'	*, Є
F	(, id

2.2.2 PREDICTIVE PARSING

A predictive parsing is a special form of recursive-descent parsing, in which the current input token unambiguously determines the production to be applied at each step. The goal of predictive parsing is to construct a top-down parser that never backtracks. To do so, we must transform a grammar in two ways:

Eliminate left recursion, and Perform left factoring.

These rules eliminate most common causes for backtracking although they do not guarantee a completely backtrack-free parsing (called LL(1) as we will see later)

Nonrecursive Predictive Parsing:

- A nonrecursive predictive parser can be built by maintaining a stack explicitly, rather than ~~recursion~~ implicitly via recursive calls.

The table-driven parser has an i/p buffer, a stack containing a sequence of grammar symbols, a parsing table and an o/p stream.

- The i/p buffer contains the string to be parsed, followed by the end marker $.

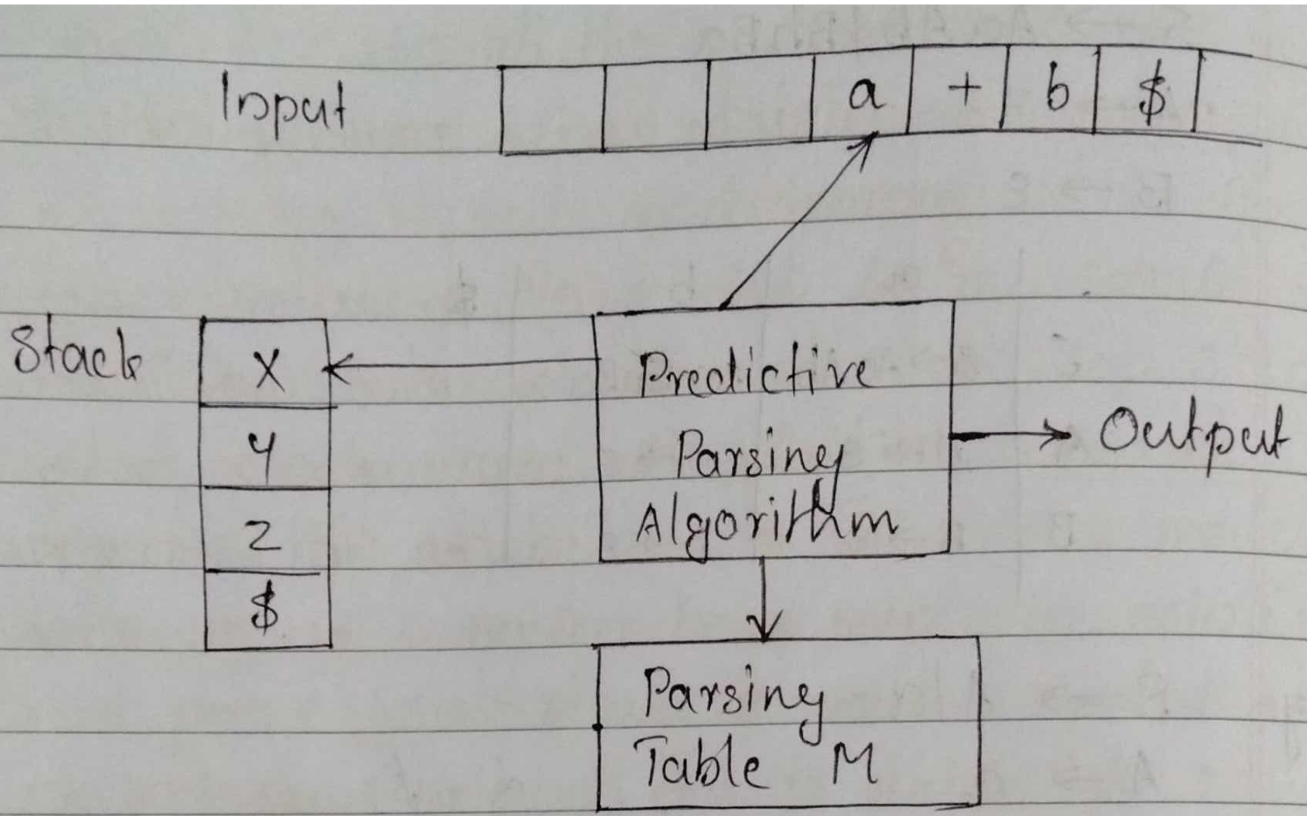


Fig: Model of a predic table-driven predictive Parser

- The symbol $ is reused to mark the bottom of the stack, which initially contain the start symbol of the grammar on top of $.

- The parser is controlled by a program that considers X, the symbol on top of the stack, and a the current i/p symbol. If X is a non-terminal the parser chooses an X- pdn by consulting entry M[X,a] of the parsing table M. otherwise, it checks for a match between the terminal X and current i/p symbol a.

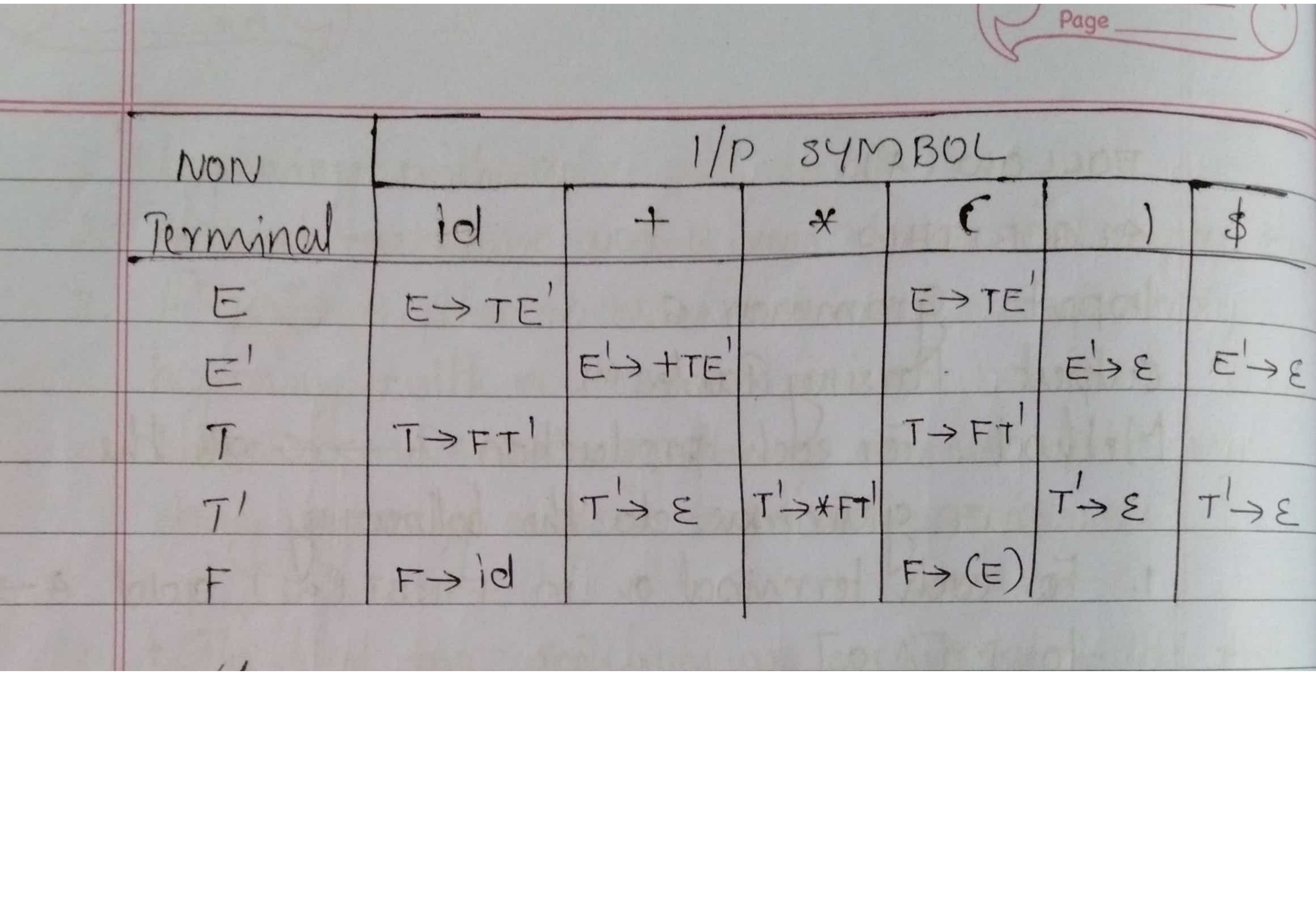
Algorithm:

INPUT: A string w and a parsing table M for grammar G.

OUTPUT: If w is in L(G), a leftmost derivation of w; otherwise, an error indication.

METHOD: Initially, the parser is in a configuration with w$ in the I/p buffer and the start symbol S of G on top of the stack, above $.

```
set ip to point to the first symbol of w; (ie, a)
set X ~~to~~ the top of stack symbol;
while ($X \neq \$$) /* stack not empty */
{
   If ($X = a$) pop the stack and advance ip;
   else if (X is a terminal) error();
   else if (M[X, a] is an error entry) error();
   else if ($M[X, a] = X \rightarrow Y_1 Y_2 \cdots Y_k$)
   {
       output the production $X \rightarrow Y_1 Y_2 \cdots Y_k$ ;
       pop the stack;
       push $Y_k, Y_{k-1} \cdots Y_1$ on to the stack with
            $Y_1$ on top;
   }
   set X to the top of stack symbol;
}
```



NON	I/P SYMBOL					
Terminal	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)		

classmate
Date
Page

eg: Consider the following grammar

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

Input : id + id * id

The moves corresponds to the LMD

$$E \underset{lm}{\Rightarrow} TE' \underset{lm}{\Rightarrow} FT'E' \Rightarrow id\,T'E' \Rightarrow id\,E' \Rightarrow id+TE' \Rightarrow .$$

Matched	Stack	Input	Action
	$E\$$	$id+id*id\$$	
	$TE'\$$	$id+id*id\$$	output $E \rightarrow TE'$
	$FT'E'\$$	$id+id*id\$$	output $T \rightarrow FT'$
	$idT'E'\$$	$id+id*id\$$	output $F \rightarrow id$
id	$T'E'\$$	$+id*id\$$	match id
id	$E'\$$	$+id*id\$$	output $T' \rightarrow \varepsilon$
id	$+TE'\$$	$+id*id\$$	output $E' \rightarrow +TE'$
id+	$TE'\$$	$id*id\$$	match +
id+	$FT'E'\$$	$id*id\$$	output $T \rightarrow FT'$
id+	$idT'E'\$$	$id*id\$$	output $F \rightarrow id$
id+id	$T'E'\$$	$*id\$$	match id
id+id	$*FT'E'\$$	$*id\$$	output $T' \rightarrow *FT'$
id+id*	$FT'E'\$$	$id\$$	match *
id+id*	$idT'E'\$$	$id\$$	output $F \rightarrow id$

id + id * id	$T'E'\$$	\$	match id
id + id * id	$E'\$$	\$	output $T' \to \varepsilon$
id + id * id	\$	\$	output $E' \to \varepsilon$.

- An error is detected during predictive parsing when the terminal on top of the stack does not match the next i/p symbol or when non-terminal A is on top of the stack, a is the next i/p symbol and M [A, a] is error (i.e, parsing table entry is empty).

FIRST and FOLLOW:

- The construction ab both top-down and bottom-up parsers is aided by two functions, FIRST and FOLLOW, associated with grammar G.

- During top-down parsing, FIRST and FOLLOW allows us to choose which pdn to apply, based on the next I/p symbol.

- Define FIRST (α), where α is any string ab grammar symbols to be the set ab terminals that begin strings derived from α.

eg: If $\alpha \overset{*}{\Rightarrow} \epsilon$, then ϵ is also in FIRST (α)

If $S \Rightarrow \alpha A a \beta \Rightarrow \alpha c \gamma a \beta$,
then c is in FIRST (A)

- Consider two A-pdns, $A \rightarrow \alpha | \beta$, where FIRST ($\alpha$) and FIRST ($\beta$) are disjoint sets. we can

Then choose between these A pdns by looking at the next i/p symbol a, since a can be in atmost one of FIRST(α) and FIRST(β), not both.

For instance, if a is in FIRST(β) choose the pdn $A \rightarrow \beta$.

To compute FIRST(X) for all grammar symbol X, apply the following rules until no more terminals or ϵ can be added to any FIRST set.

1. If X is a terminal, then FIRST(X) = {X}.
2. If X is a non-terminal and $X \rightarrow Y_1 Y_2 \cdots Y_k$ is a pdn for some $k \geq 1$, then place a in FIRST(X), if for some i, a is in FIRST(Y_i), and ϵ is in all of FIRST(Y_1) ... FIRST(Y_{i-1})

   ie, $Y_1 \cdots Y_{i-1} \overset{*}{\Rightarrow} \epsilon$

   If ϵ is in FIRST(Y_j) for all $j = 1, 2 \cdots k$, then ~~and~~ add ϵ to FIRST(X)

   If Y_1 does not derive ϵ, then we add nothing more to FIRST(X), but if $Y_1 \overset{*}{\Rightarrow} \epsilon$, then we add FIRST($Y_2$) and so on.
3. If $X \rightarrow \epsilon$ is a pdn, then add ϵ to FIRST(X).

Eg: Consider the following grammar

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) | id$

FIRST(E) = FIRST(T) = FIRST(E) = { C, id}

FIRST (E') = { +, ε}

FIRST (T') = {*, ε)

- Define FOLLOW(A), for a nonterminal A, to be the set of terminals a that can appear immediat to the right of A in some sentential form.

eg: If $S \overset{*}{\Rightarrow} \alpha A a \beta$, then ~~A~~ a is in FOLLOW(A)

- If A is the rightmost symbol in some sententia form, then $ is in FOLLOW(A). ($ is a special end marker symbol that is assumed not to be a symbol of any grammar).

- To compute FOLLOW(A), for all nonterminals A, apply the following rules until nothing can be added to any FOLLOW set.

1. Place $ in 'FOLLOW(S), where S is the start symbol, and $ is the i/p right end marker.

2. If there is a pdn $A \rightarrow \alpha B \beta$, then everything

in FIRST(β), except ε is in FOLLOW(B).

3. If there is a pdn $A \rightarrow \alpha B$, or a pdn $A \rightarrow \alpha B \beta$ where FIRST(β) contains ε, then everything in FOLLOW(A) is in FOLLOW(B).

eg: Consider the following grammar.

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

FOLLOW(E) = FOLLOW(E') = {), \$ }.

FOLLOW(T) = FOLLOW(T') = { +,), \$ }

FOLLOW(F) = { *, +,), \$ }.

Egs: 1)

	FIRST	FOLLOW
$S \rightarrow ABCDE$	{a, b, c}	{ \$ }
$A \rightarrow a/\varepsilon$	{a, ε}	{b, c}
$B \rightarrow b/\varepsilon$	{b, ε}	{c}
$C \rightarrow c$	{c}	{d, e, \$}
$D \rightarrow d/\varepsilon$	{d, ε}	{e, \$}
$E \rightarrow e/\varepsilon$	{e, ε}	{ \$ }.

// whenever a variable is at the right end of a pdn and if nothing after it then the ~~follow~~ FOLLOW of that variable is FOLLOW of left hand side.

whenever a variable after which everything goes to ε, for eg. in FOLLOW (C), D and E are going to ε, in this case C is coming to the right end, so FOLLOW of right end is equal to the FOLLOW of left end //

		FIRST	FOLLOW
2)	$S \rightarrow Bb \mid Cd$	{a, b, c, d}.	{$}
	$B \rightarrow aB \mid \varepsilon$	{a, ε}	{b}
	$C \rightarrow cC \mid \varepsilon$	{c, ε}	{d}.

		FIRST	FOLLOW
3.	$E \rightarrow TE'$	{id, (}	{$,)}
	$E' \rightarrow +TE' \mid \varepsilon$	{+, ε}.	{$,)}
	$T \rightarrow FT'$	{id, (}	{+, $,)}
	$T' \rightarrow *FT' \mid \varepsilon$	{*, ε}	{+, $,)}
	$F \rightarrow id \mid (E)$	{id, (}	{*, +, $,)}.

// E is the start symbol, so FOLLOW(E) always contains $. and E is also in the r.h.s so FOLLOW (E) = {$,)}.

FOLLOW(E') - E' is at the right end so FOLLOW (E') = FOLLOW (E).

~~Fow~~ FOLLOW (T) = FIRST (E')

FIRST(E') contains ε, so replace E' with ε now T is at the right end so FOLLOW (T) contains FOLLOW (E)

ie, FOLLOW (T) = {+, \$,)}. //

4).

	FIRST	FOLLOW
$S \rightarrow ACB \mid CbB \mid Ba$	{d, g, h, ε, b, a}	{\$}
$A \rightarrow da \mid BC$	{d, g, h, ε}	{h, g, \$}
$B \rightarrow g \mid \epsilon$	{g, ε}	{\$, a, h, g}
$C \rightarrow h \mid \epsilon$	{h, ε}	{g, \$, b, h}

are disjoint sets.

Construction of a Predictive Parsing Table:

This algm collects information from FIRST and FOLLOW sets into a predictive parsing table M[A, a], a two dimensional array

- where A is a non-terminal and a is a terminal or the symbol $\$$, the i/p endmarker

The algm is based on the following idea.

The production $A \rightarrow \alpha$ is chosen, if the next i/p symbol a is in FIRST(α). The only complication occurs when $\alpha \Rightarrow \varepsilon$, or more generally $\alpha \overset{*}{\Rightarrow} \varepsilon$ In this case, we should again choose $A \rightarrow \alpha$ if the current i/p symbol is in FOLLOW(A). or if the $\$$ on the i/p has been reached and $\$$ is in

FOLLOW (A).

ALGORITHM:

Input: Grammar G

output: Parsing Table M.

Method: For each production $A \rightarrow \alpha$ ob the grammar, do the bollowing

1. For each terminal a in FIRST (α), add $A \rightarrow \alpha$ to M[A,a]
2. If ε is in FIRST(α), then for each terminal b in FOLLOW(A), add $A \rightarrow \alpha$ to M[A,b]. If ε is in FIRST (α) and \$ is in FOLLOW(A), add $A \rightarrow \alpha$ to M[A, \$] as well.

If abter performing the above, there is no production at all in M[A,a], then set M[A,a] to error. (normally represent by an empty string in the table.

Eg: For the expression grammar

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) / id.$

string: id + id * id.

NON	I/P SYMBOL					
Terminal	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)		

	FIRST	FOLLOW
E→TE'	{id, (}	{$,)}
E'→+TE'/ε	{+, ε}	{$,)}
T→FT'	{id, (}	{+,), $}
T'→*FT'/ε	{*, ε}	{+,), $}
F→id | (E)	{id, (}	{*, +,), $}

$. is included because there is a chance for there is a $ in the input string.

E'→ε place the production in the follow of rhs.

Eg: (id E') $

Apply ε production to E' if the next symbol is)

ie, (id)

LL(1) Grammar.

Predictive parsers (i.e, recursive descent parsers needing no backtracking) can be constructed for a class of grammars called LL(1).

LL(1)

1st L stands for — Scanning the i/p from left to right

second L stands for — Producing a leftmost derivation

1 for using — one i/p symbol of lookahead at each step to make parsing action decisions.

A grammar G is LL(1) if and only if whenever $A \longrightarrow \alpha / \beta$ are two distinct productions of G, the following conditions hold:

1. For no terminal 'a' do both α and β derive strings

beginning with a.

2. At most one of α and β can derive the empty str

3. If $\beta \overset{*}{\Longrightarrow} \varepsilon$, then α does not derive any string beginning with a terminal in FOLLOW(A). Likewise, if $\alpha \overset{*}{\Longrightarrow} \varepsilon$, then β does not derive any string beginning with a terminal in FOLLOW(A).

- The first two conditions are equivalent to the stmt that FIRST (α) and FIRST (β) are disjoint sets
- The third condition is equivalent to stating that if ε is in FIRST (β), then FIRST (α) and FOLLOW(A) are disjoint sets.

1. Recursive Descent Parsing:

- A recursive descent parsing program consists of a set of procedures, one for each non-terminal.

Execution begins with the procedure for the start symbol, which halts and announces success if its procedure body scans the entire i/p string.

- general recursive descent may require backtracking (ie, it may require repeated scans

classmate
Date
Page

over the i/p).

A typical procedure for a nonterminal in a top-down parser

```
    void A() {
1)      choose an A-pdn, A → X1 X2 ..... Xk ;
2)      for (i=1 to k) {
3)          if (Xi is a nonterminal)
4)              call procedure Xi();
5)          else if (Xi equals the current i/p symbol a)
6)              advance the i/p to the next symbol;
7)          else /* an error has occurred */;
        }
    }
```

To allow backtracking, the above code need to be modified. It is not possible to choose a unique A-pdn at line 1, so several pdns have to be tried out in some order. Then failure at line 7 is not ultimate failure, but suggest that we need to return to line(1) and try another A-pdn. Only if there are no more A-pdns to try, we say that an error has been found. So inorder to try another A-pdn, i/p pointer has to be reset to the point where it was when it reached line 1. Thus a local variable is needed to store the i/p pointer for further use.

Modify the above pgm code to indicate

include the bollowing:

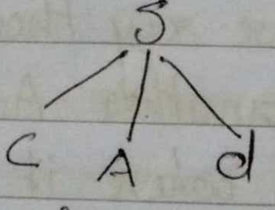
7) else if A has any alternative pdns
8) choose any one alternative path bor A and reset i/p pointer.
9) else error();

Eg: Consider the grammar

$S \rightarrow cAd$
$A \rightarrow ab|a$

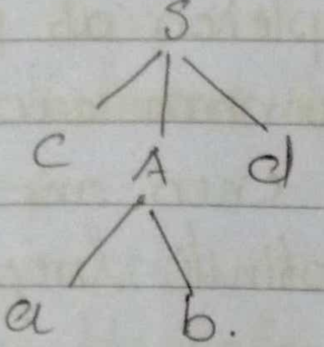
construct a parse tree bor the i/p string $w = cad$.

Ans: To construct a parse tree top-down bor the i/p string $\omega = cad$, begin with a tree consisting ab a single node labelled S, and the i/p pointer pointing to c, the birst symbol ab ω. S has only one pdn, so it is used to expand S.
Thus the parse tree becomes



The lebtmost leab, labeled 'c' matches the birst symbol ab input ω, so i/p pointer is advanced by 1 position and thus it now points to 'a', the second symbol of ω.

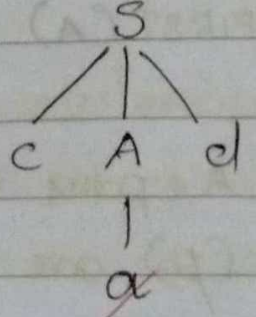
- The next leaf is A when the 1st alternative A → ab is choosen, the tree becomes



A match is obtained for the second symbol 'a' so the i/p pointer is advanced by 1. and hence, it points to 'd' is compared with the next leaf, labeled 'b'. Since 'b' does not match with 'd', we must go back to A to see whether there is another alternative for A that has not been tried.

In going back to A, we must reset the i/p pointer to position 2. (The position it had, when it first came to A) which means that the procedure for A must store the i/p pointer in a local variable.

The 2nd alternative for A produces the tree



The leaf 'a' matches 2nd symbol of w and leaf 'd' matches the 3rd symbol. Since we have produced a parse tree for w, we halt and announce successful completion of parsing.

A left recursive grammar can cause a recursive descent parser even one with back-tracking, to go into an infinite loop.