MODULE 2

(Part I)

Syntax Analysis: Review of Context-Free Grammars - Derivation trees and Parse Trees, Ambiguity.

Top-Down Parsing: Recursive Descent parsing, Predictive parsing, LL(1) Grammars

SYNTAX ANALYSIS

Syntax analysis or parsing is the second phase of a compiler.

A lexical analyzer can identify tokens with the help of regular expressions and pattern rules. But a lexical analyzer cannot check the syntax of a given sentence due to the limitations of the regular expressions.

Regular expressions cannot check balancing tokens, such as parenthesis. Therefore, this phase uses context-free grammar (CFG), which is recognized by push-down automata.

REVIEW OF CONTEXT FREE GRAMMARS

A context-free grammar (grammar for short) consists of terminals, nonterminals, a start symbol, and productions.

1. **Terminals** are the basic symbols from which strings are formed. The term "token name" is a synonym for "terminal" and frequently we will use the word "token" for terminal when it is clear that we are talking about just the token name.
2. **Nonterminals** are syntactic variables that denote sets of strings. The nonterminals define sets of strings that help define the language generated by the grammar. They also impose a hierarchical structure on the language that is useful for Both syntax analysis and translation.
3. In a grammar, one nonterminal is distinguished as the **start symbol**, and the set of strings it denotes is the language generated by the grammar. Conventionally, the productions for the start symbol are listed first.
4. The **productions** of a grammar specify the manner in which the terminals and nonterminals can be combined to form strings. Each production consists of:
    a. A nonterminal called the head or left side of the production; this production defines some of the strings denoted by the head.
    b. The symbol →. Sometimes : : = has been used in place of the arrow.
    c. A body or right side consisting of zero or more terminals and nonterminals.

EXAMPLE:

The grammar with the following productions defines simple arithmetic expression:

$$\begin{array}{rcl}
expression & \rightarrow & expression + term \\
expression & \rightarrow & expression - term \\
expression & \rightarrow & term \\
term & \rightarrow & term * factor \\
term & \rightarrow & term \ / \ factor \\
term & \rightarrow & factor \\
factor & \rightarrow & (\ expression\ ) \\
factor & \rightarrow & \mathbf{id}
\end{array}$$

In this grammar, the **terminal symbols** are : id + - * / ( )

The **nonterminal symbols** are : expression, term, factor

Start symbol : expression

Notational Conventions

To avoid always having to state that "these are the terminals," "these are the nonterminals," and so on, the following notational conventions for grammars will be used.

- These symbols are terminals:
  - **a.** Lowercase letters early in the alphabet, such as a, b, c.
  - **b.** Operator symbols such as +, *, and so on.
  - **c.** Punctuation symbols such as parentheses, comma, and so on.
  - **d.** The digits 0, 1, . . . , 9.
  - **e.** Boldface strings such as id or if, each of which represents a single terminal symbol.
- These symbols are nonterminals:
  - **a.** Uppercase letters early in the alphabet, such as A, B, C.
  - **b.** The letter S, which, when it appears, is usually the start symbol.
  - **c.** Lowercase, italic names such as expr or stmt.
  - **d.** When discussing programming constructs, uppercase letters may be used to represent nonterminals for the constructs. For example, nonterminals for expressions, terms, and factors are often represented by E, T, and F, respectively.
- Uppercase letters late in the alphabet, such as X, Y, Z, represent grammar symbols; that is, either nonterminals or terminals.
- Lowercase letters late in the alphabet , chiefly u, v, ... ,z, represent (possibly empty) strings of terminals.
- Lowercase Greek letters α, β, γ for example, represent (possibly empty) strings of grammar symbols.
- A set of productions A → α1 , A → α2 , ... , A → αk with a common head A (call them A-productions) , may be written A → α1| α2|.....| αk · We call α1, α2,.., αn the alternatives for A.
- Unless stated otherwise, the head of the first production is the start symbol.

Using these conventions, the grammar for arithmetic expression can be rewritten as:

E→ E + T | E - T | T

T→ T * F | T / F | F

F→ (E) | id

DERIVATION TREES AND PARSE TREES

The construction of a parse tree can be made precise by taking a derivational view, in which productions are treated as rewriting rules.

Beginning with the start symbol, each rewriting step replaces a nonterminal by the body of one of its productions.

For example , consider the following grammar , with a single nonterminal E:

E → E + E | E * E | - E | (E) | id

The production E → - E signifies that if E denotes an expression, then – E must also denote an expression. The replacement of a single E by - E will be described by writing E => -E which is read, "E derives - E."

The production E -+ (E) can be applied to replace any instance of E in any string of grammar symbols by (E) , e.g., **E * E => (E) * E or E * E => E * (E)**

We can take a single E and repeatedly apply productions in any order to get a sequence of replacements. For example, **E => - E => - (E) => - (id)**

We call such a sequence of replacements a derivation of - (id) from E. This derivation provides a proof that the string - (id) is one particular instance of an expression.

Derivation Order

1. $S \to AB$ 2. $A \to aaA$ 4. $B \to Bb$

3. $A \to \lambda$ 5. $B \to \lambda$

Leftmost derivation:

$$S \overset{1}{\Rightarrow} AB \overset{2}{\Rightarrow} aaAB \overset{3}{\Rightarrow} aaB \overset{4}{\Rightarrow} aaBb \overset{5}{\Rightarrow} aab$$

Rightmost derivation:

$$S \overset{1}{\Rightarrow} AB \overset{4}{\Rightarrow} ABb \overset{5}{\Rightarrow} Ab \overset{2}{\Rightarrow} aaAb \overset{3}{\Rightarrow} aab$$

Leftmost And Rightmost Derivation Of A String

- **Leftmost derivation** – A leftmost derivation is obtained by applying production to the leftmost variable in each step.
- **Rightmost derivation** – A rightmost derivation is obtained by applying production to the rightmost variable in each step.

Example

Let any set of production rules in a CFG be

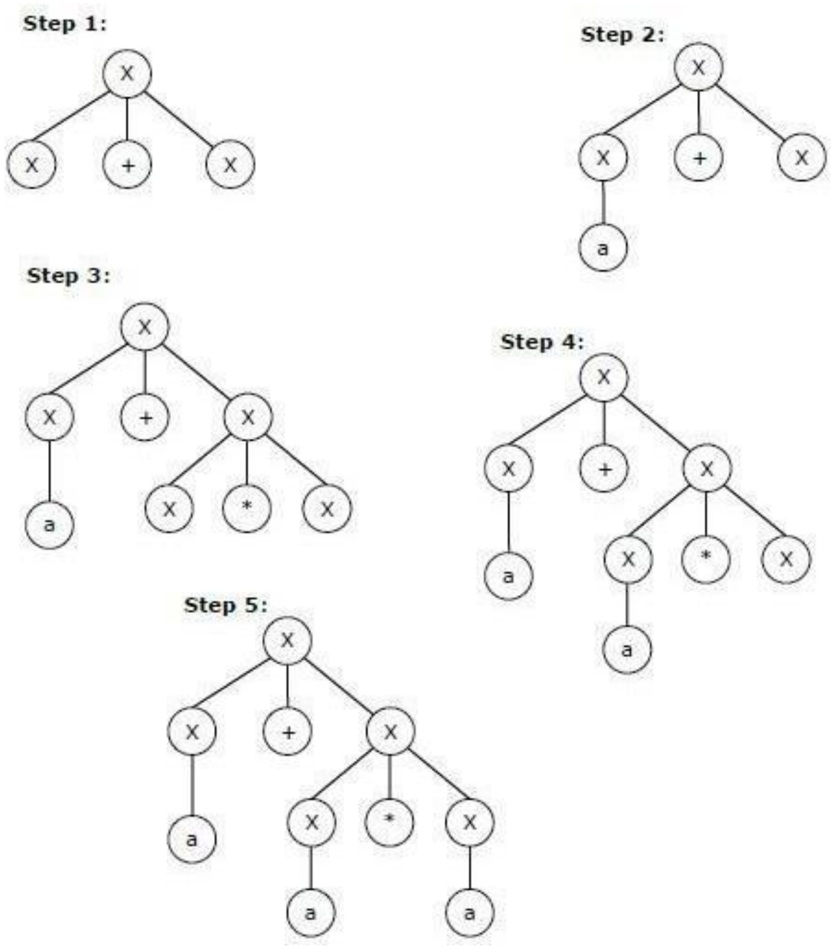
X → X+X | X*X |X| a

over an alphabet {a}.

The leftmost derivation for the string **"a+a*a"** may be –

X → X+X → a+X → a + X*X → a+a*X → a+a*a

The stepwise derivation of the above string is shown as below –



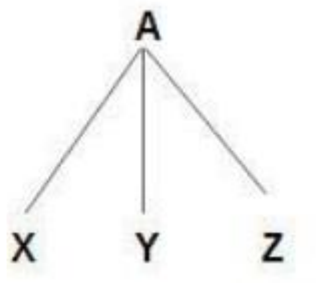
The rightmost derivation for the above string "a+a*a" may be

X → X*X → X*a → X+X*a → X+a*a → a+a*a

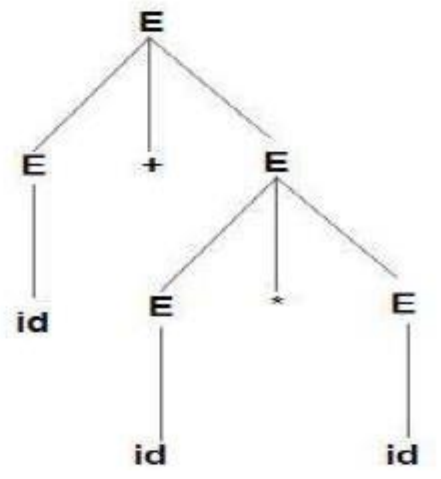
Parse Tree

- Parse tree is a hierarchical structure which represents the derivation of the grammar to yield input strings.
- Simply it is the graphical representation of derivations.

- Root node of parse tree has the start symbol of the given grammar from where the derivation proceeds.
- Leaves of parse tree are labeled by non-terminals or terminals.
- Each interior node is labeled by some non terminals.
- If **A → xyz** is a production, then the parse tree will have **A** as interior node whose children are **x, y** and **z** from its left to right.



- Construct parse tree for **E → E + E / E * E /id**



Yield of Parse Tree

- The leaves of the parse tree are labeled by nanterminals or terminals and read from left to right, they constitute a sentential form, called the yield or frontier of the tree.
- Figure above represents the parse tree for the string **id+ id*id**. The string **id + id * id**, is the yield of parse tree depicted in Figure.

AMBIGUITY

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

- 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

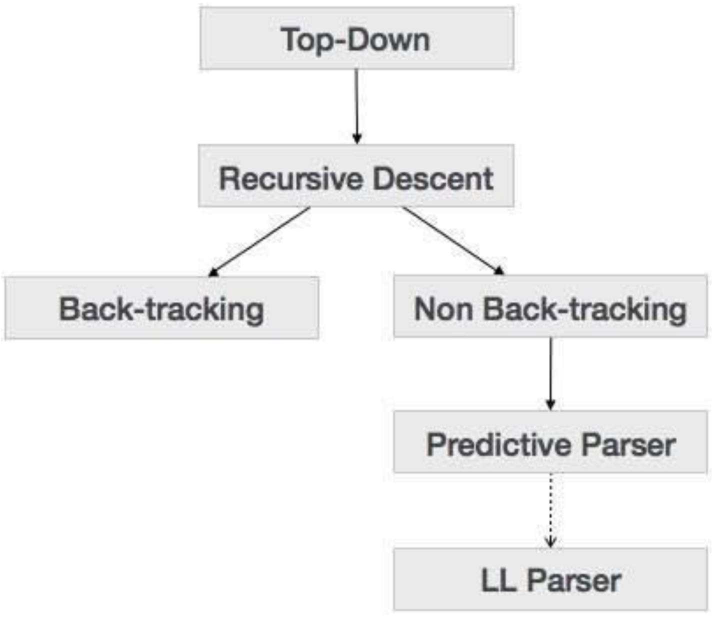
Consider very simple sentence id+ id * id.

```
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
```

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).



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

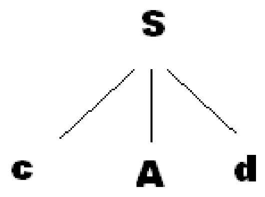
Consider the grammar:

S → cAd

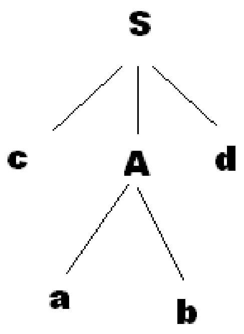
A → ab | a

and the input string w = cad.

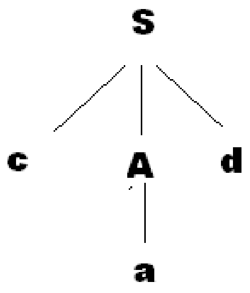
- To construct a parse tree for this string top down, we initially create a tree consisting of a single node labelled **S.**
- An input pointer points to **c**, the first symbol of w. **S** has only one production, so we use it to expand **S** and obtain the tree as:



- 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).

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

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

  where no β_i begins with an **A**. Then we replace the **A**-productions by

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

  $$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A' \mid \varepsilon$$

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

  $$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$$

- are two A-productions, and the input begins with a non-empty string derived from α we do not know whether to expand **A** to $\alpha\beta_1$ or $\alpha\beta_2$.
- However, we may defer the decision by expanding **A** to α**B**. Then, after seeing the input derived from α, we may expand **B** to β_1 or β_2 .
- The left factored original expression becomes:

  $$A \rightarrow \alpha B$$
  $$B \rightarrow \beta_1 \mid \beta_2$$

- 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. Stack
2. Parsing Table
3. Input Buffer
4. Parsing

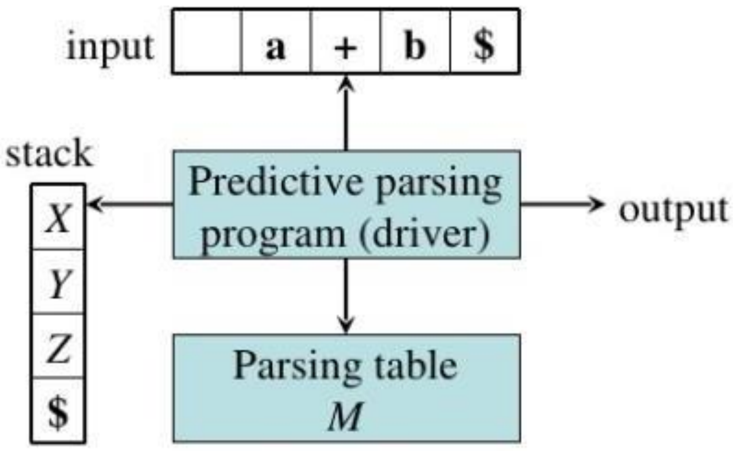


Figure : ***Model of a nonrecursive predictive parser***

- **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** $\neq$ **$** , 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_l onto the stack, with Y_l on top;
            output the production X → Y_1 Y_2 .......... Y_k
        end
        else error ()
until X = S /* stack is empty */
```

EXAMPLE

Consider Grammar:

$E \rightarrow T E'$

$E' \rightarrow +T E' \mid \epsilon$

$T \rightarrow F T'$

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

$F \rightarrow (E) \mid id$

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 $\alpha \overset{*}{\Rightarrow} \epsilon$ 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**$\rightarrow$ **ϵ** then add **ϵ** to FIRST(**X**)
  3. If **X** is a non terminal and **X**$\rightarrow$**$Y_1Y_2Y_3 \ldots Y_n$**, then put '**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}**).

EXAMPLE

Consider Grammar:

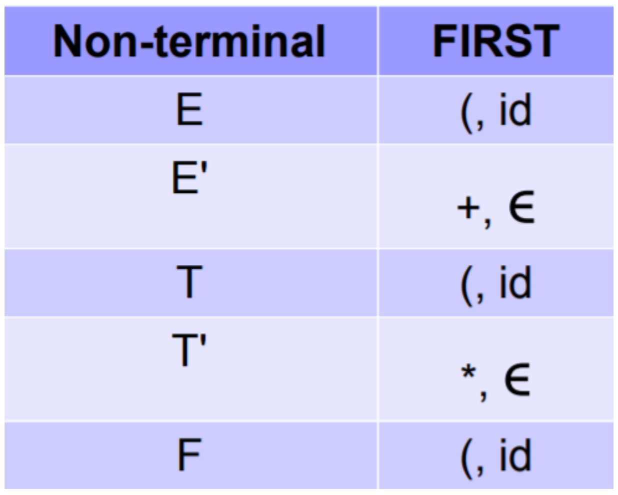
E $\rightarrow$ T E'

E' $\rightarrow$ +T E' | ϵ

T $\rightarrow$ F T'

T' $\rightarrow$ * F T' | ϵ

F $\rightarrow$ (E) | id



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

FOLLOW

- FOLLOW is defined only for non-terminals of the grammar **G**.
- It can be defined as the set of terminals of grammar **G** , which can immediately follow the non-terminal in a production rule from start symbol.
- In other words, if **A** is a nonterminal, then FOLLOW(**A**) is the set of terminals '**a**' that can appear immediately to the right of **A** in some sentential form.
- Rules to Compute Follow Set
  1. If S is the start symbol, then add $ to the FOLLOW(S).
  2. If there is a production rule **A→ αBβ** then everything in FIRST(**β**) except for **ϵ** is placed in FOLLOW(**B**).
  3. If there is a production **A→ αB** , or a production **A→αBβ** where FIRST(**β**) contains **ϵ** then everything in FOLLOW(**A**) is in FOLLOW(**B**).

EXAMPLE

Consider Grammar:

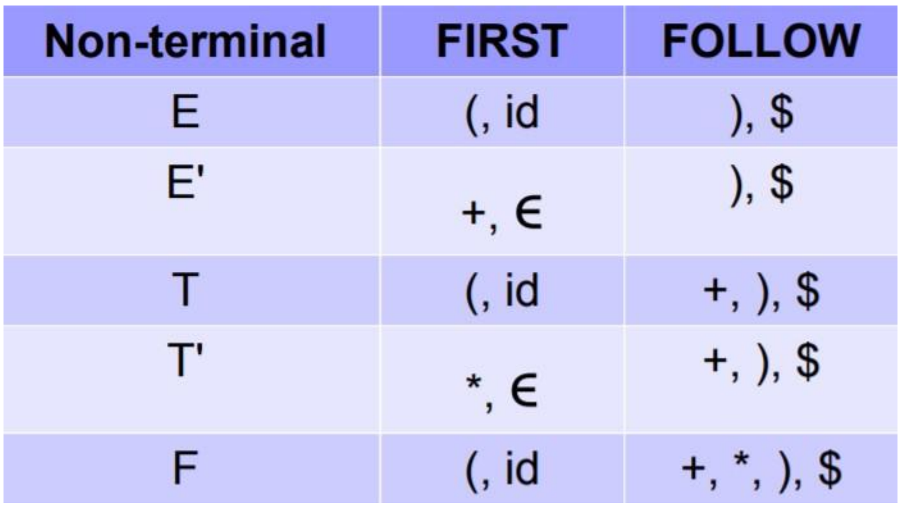
E → T E'

E' → +T E' | Є

T → F T'

T' → * F T' | Є

F → (E) | id



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

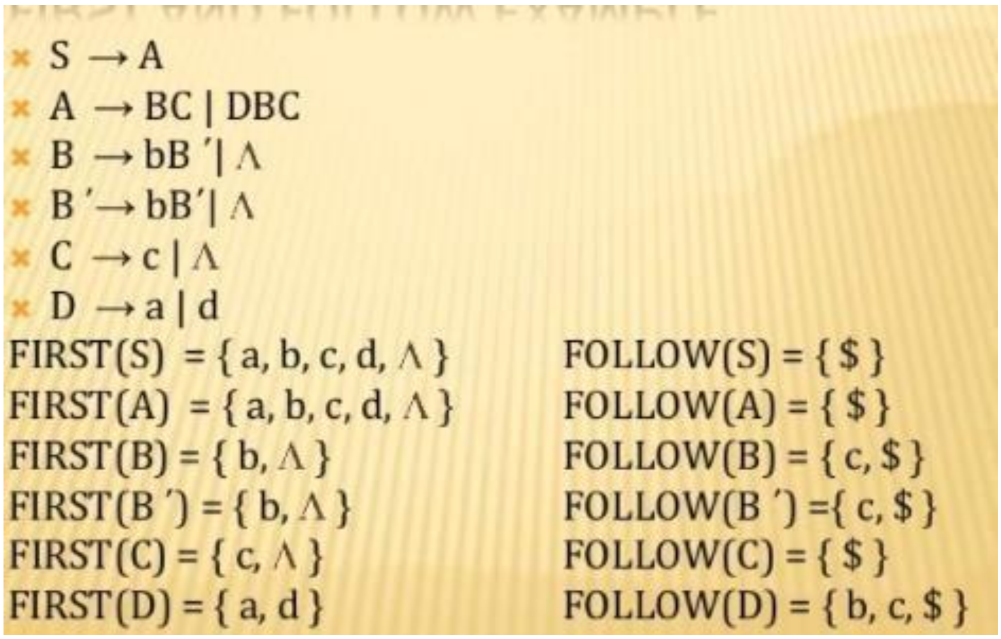
EXAMPLE

1. $S \to A\ a$
2. $A \to B\ D$
3. $B \to b$
4. $B \to \epsilon$
5. $D \to d$
6. $D \to \epsilon$

First(S) = {b, d, a}
First(A) = {b, d, ϵ}
First(B) = {b, ϵ}
First(D) = {d, ϵ}

Follow(S) = {\$}
Follow(A) = {a}
Follow(B) = {d, a}
Follow(D) = {a}

EXAMPLE



- $S \to A$
- $A \to BC \mid DBC$
- $B \to bB' \mid \Lambda$
- $B' \to bB' \mid \Lambda$
- $C \to c \mid \Lambda$
- $D \to a \mid d$

FIRST(S) = { a, b, c, d, Λ }	FOLLOW(S) = { \$ }
FIRST(A) = { a, b, c, d, Λ }	FOLLOW(A) = { \$ }
FIRST(B) = { b, Λ }	FOLLOW(B) = { c, \$ }
FIRST(B ′) = { b, Λ }	FOLLOW(B ′) ={ c, \$ }
FIRST(C) = { c, Λ }	FOLLOW(C) = { \$ }
FIRST(D) = { a, d }	FOLLOW(D) = { b, c, \$ }

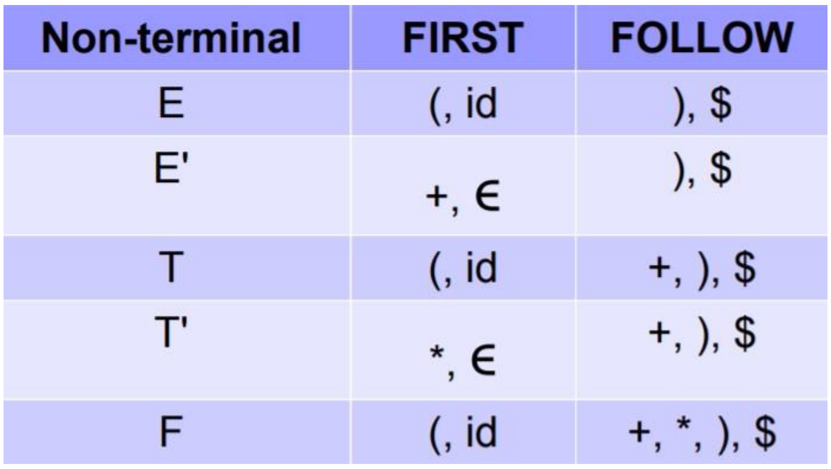
Algorithm To Construct A Predictive Parsing Table.

INPUT: Grammar **G.**

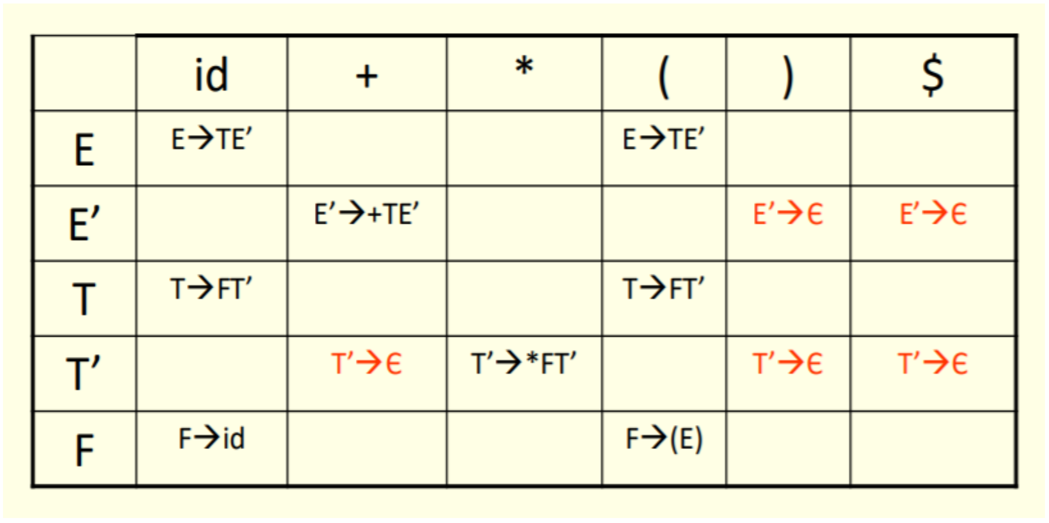
OUTPUT: Parsing table ***M***.

METHOD

1. For each production **A** →α of the grammar, do steps 2 and 3.
2. For each terminal **a** in FIRST(α), add ***A*** →α to **M|A, a|**
3. If ε is in FIRST(α), add **A**→ α to **M|A, b |** for each terminal *b* in FOLLOW (**A**). If ε is in FIRST (α)j and **$** is in FOLLOW(**A**), add A→α to ***M |A, $|***
4. Make each undefined entry of *M* be error



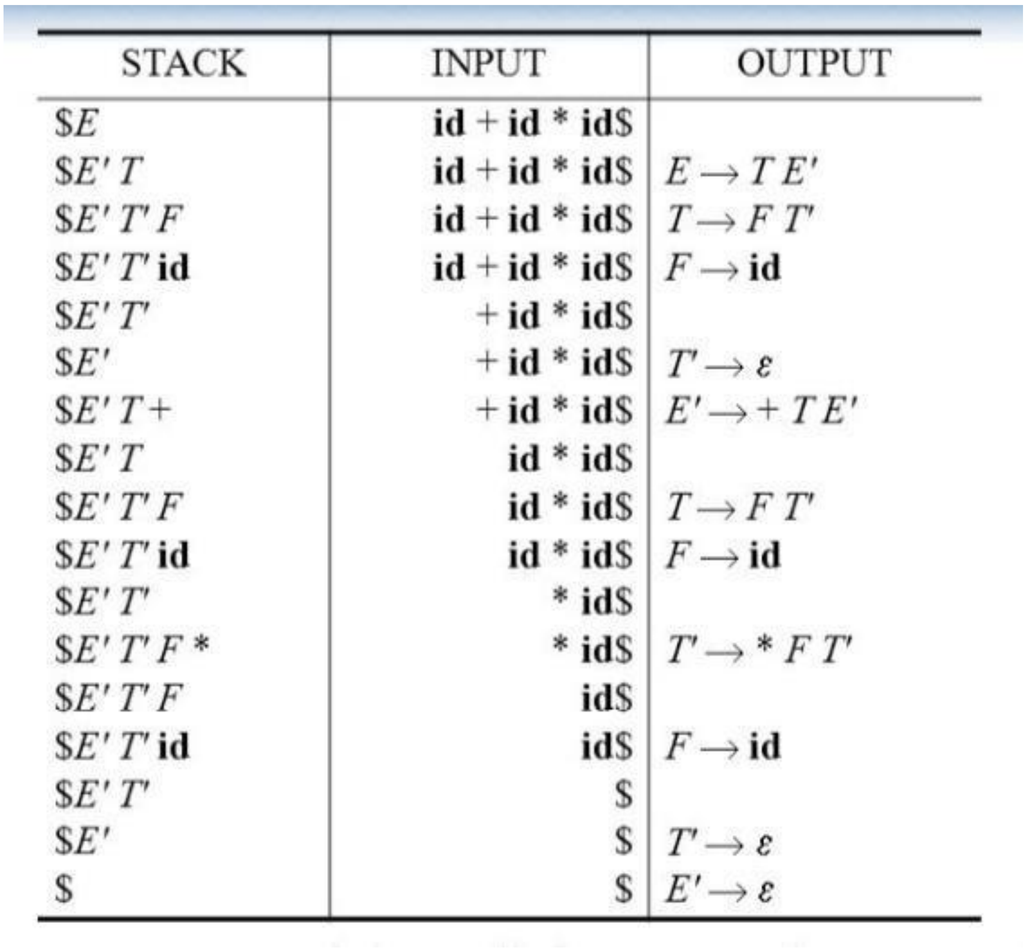
Non-terminal	FIRST	FOLLOW
E	(, id	), $
E'	+, ∈	), $
T	(, id	+,), $
T'	*, ∈	+,), $
F	(, id	+, *,), $



	id	+	*	(	)	$
E	E→TE’			E→TE’		
E’		E’→+TE’			E’→∈	E’→∈
T	T→FT’			T→FT’		
T’		T’→∈	T’→*FT’		T’→∈	T’→∈
F	F→id			F→(E)		

Parsing Table

Blank entries are error states. For example, E cannot derive a string starting with ‘+’



STACK	INPUT	OUTPUT
$\$E$	$\textbf{id} + \textbf{id} * \textbf{id}\$$	
$\$E'T$	$\textbf{id} + \textbf{id} * \textbf{id}\$$	$E \rightarrow T E'$
$\$E'T'F$	$\textbf{id} + \textbf{id} * \textbf{id}\$$	$T \rightarrow F T'$
$\$E'T'\textbf{id}$	$\textbf{id} + \textbf{id} * \textbf{id}\$$	$F \rightarrow \textbf{id}$
$\$E'T'$	$+ \textbf{id} * \textbf{id}\$$	
$\$E'$	$+ \textbf{id} * \textbf{id}\$$	$T' \rightarrow \varepsilon$
$\$E'T+$	$+ \textbf{id} * \textbf{id}\$$	$E' \rightarrow + T E'$
$\$E'T$	$\textbf{id} * \textbf{id}\$$	
$\$E'T'F$	$\textbf{id} * \textbf{id}\$$	$T \rightarrow F T'$
$\$E'T'\textbf{id}$	$\textbf{id} * \textbf{id}\$$	$F \rightarrow \textbf{id}$
$\$E'T'$	$* \textbf{id}\$$	
$\$E'T'F*$	$* \textbf{id}\$$	$T' \rightarrow * F T'$
$\$E'T'F$	$\textbf{id}\$$	
$\$E'T'\textbf{id}$	$\textbf{id}\$$	$F \rightarrow \textbf{id}$
$\$E'T'$	$\$$	
$\$E'$	$\$$	$T' \rightarrow \varepsilon$
$\$$	$\$$	$E' \rightarrow \varepsilon$

Moves made by predictive parser for the input **id+id*id**

LL(1)GRAMMERS

- LL(l) grammars are the class of grammars from Which the predictive parsers can be constructed automatically.
- A context-free grammar G = (VT, VN, P, S) whose parsing table has no multiple entries is said to be LL(1).
- In the name LL(1),
  - the first L stands for scanning the input from left to right,
  - the second L stands for producing a leftmost derivation,
  - and the 1 stands for using one input symbol of lookahead at each step to make parsing action decision.
- A language is said to be LL(1) if it can be generated by a LL(1) grammar. It can be shown that LL(1) grammars are
  - not ambiguous and

- not left-recursive

EXAMPLE

Consider the following grammar

S → i E t S S' | a

S' → eS | ϵ

E → b

Non-terminal	FIRST	FOLLOW
S	i, a	e, $
S'	e, ∈	e, $
E	b	t

Non-terminal	i	t	a	e	b	$
S	S → i E t S S'		S → a			
S'				S' → e S S' → ∈		S' → ∈
E					E → b	

This is not LL(1) grammar

****