Lecture 6

Top-Down Parsing :-

Top-down parsing can be viewed as an attempt to find a *leftmost derivation* for an input string. Equivalently, A top down parser, such as LL(1) parsing, move from the goal symbol to a string of terminal symbols. in the terminology of trees, this is moving from the root of the tree to a set of the leaves in the syntax tree for a program. in using full backup we are willing to attempt to create a syntax tree by following branches until the correct set of terminals is reached. in the worst possible case, that of trying to parse a string which is not in the language, all possible combinations are attempted before the failure to parse is recognized. the nature of top down parsing technique is characterized by:

1-Recursive-Descent Parsing : It is a general form of Top-Down Parsing that may involve " *Backtracking* ",that is ,making repeated scans of the input.

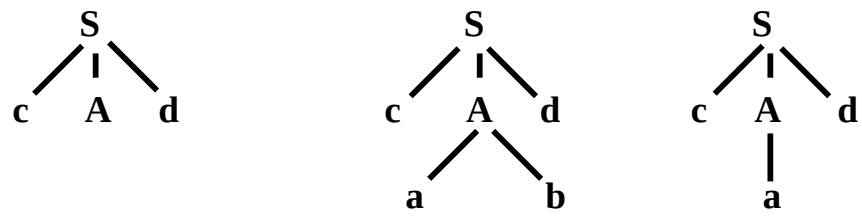
Example: consider the grammar

S ⟶ cAd

A ⟶ ab | a

Input : **cad**

Then the implementation of Recursive-Descent Parsing is:



- a - **- b -** **- c -**

2-Predictive parsing : In many cases, by carefully writing a grammar , eliminating *left-recursion* from it and *left-factoring* the resulting grammar, we can obtain agrammar that can be parsed by *recursive-descent parser* that needs no "*Backtracking*",i.e.,a **Predictive parser**.

2.1. Transition Diagrams for Predictive parsers

It is useful plan or flowchart for a predictive parser. There is one diagram for each *nonterminal*, the labels of edges are *tokens* and *nonterminals*.for example:

E→ E+T | T

T→ T*F | F // Original grammar

F→ (E) | id

Eliminate left-recursion and left factoring

E→ T E'

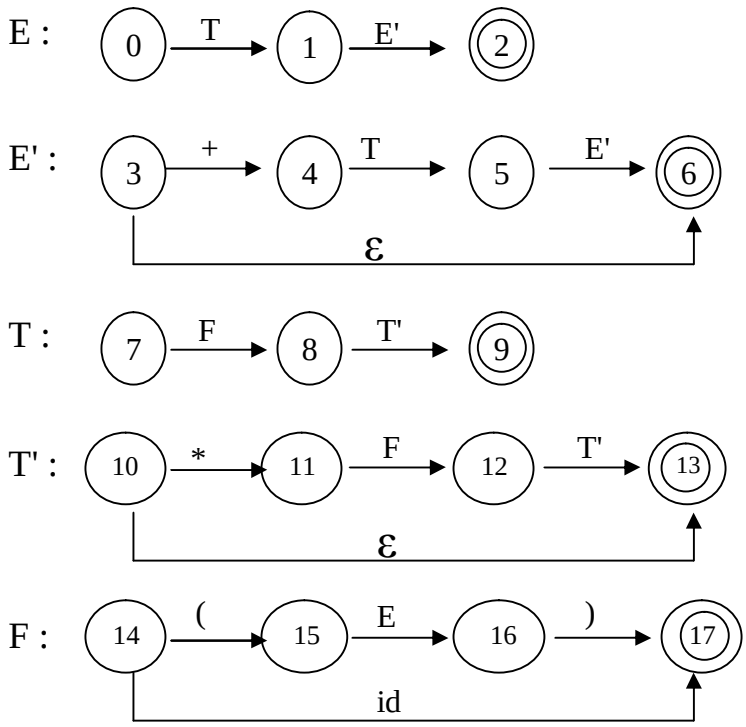
E' → +T E' | ϵ

T→ FT'

T'→ *F T' | ϵ

F→ (E) | id

Transition Diagrams



First & Follow :

- **First :** To compute First(X) for all grammar symbols apply the following rules until no more *terminal* or ε can be added to any First set **:**
  1. If x is *terminal*, then **FIRST(x)** is {x}.
  2. If x ⟶ ϵ is a production ,then add ϵ to FIRST(x).
  3. If x is *nonterminal* and x ⟶ y1y2 … yk is a production, then place *a* in FIRST(x) if for some *i* ,*a* is in FIRST(yi),and ϵ is in all of FIRST(y1)… FIRST(yi-1).
- **Follow :**To compute Follow(A) for all *nonterminals* apply the following rules until nothing can be added to any Follow set.
  1. Place $ in FOLLOW(S),where S is the start symbol.
  2. If there are a production A⟶ αBß, then everything in FIRST(ß)except for **ϵ** is placed in FOLLOW(B).
  3. If there are a production A ⟶αB ,or a production A⟶αBß where FIRST(ß) contains **ϵ**, then everything in FOLLOW(A) is in FOLLOW(B).

Example : suppose the following grammar

E⟶ T E'

E' ⟶ +T E' | ϵ

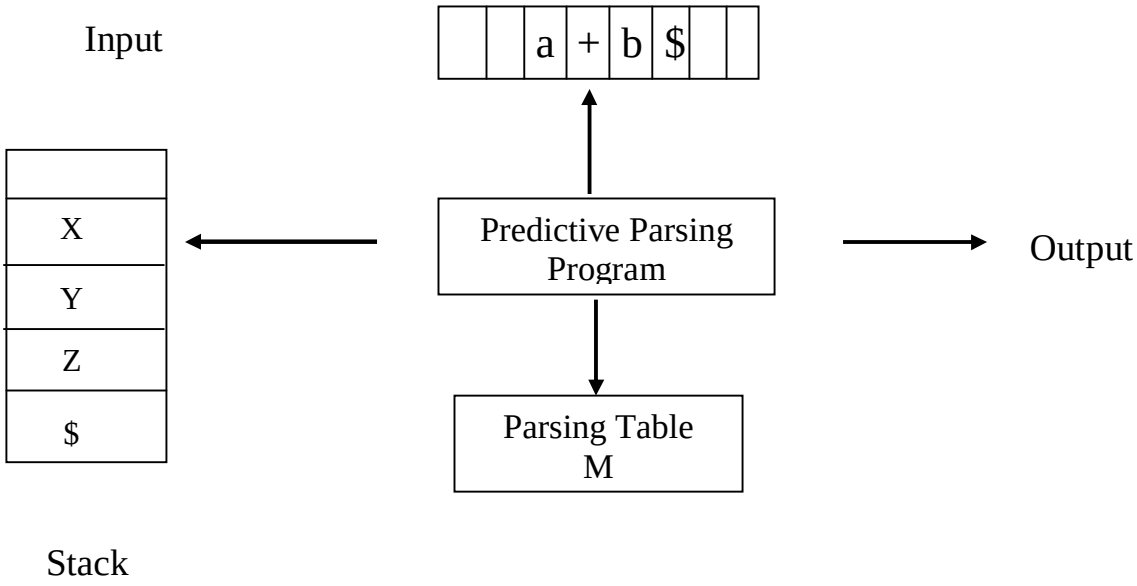
T⟶ FT'

T'⟶ *F T' | ϵ

F⟶ (E) | id

Nonterminals	First	Follow
E	**(, id**	**) , $**
E'	**+ ,** ϵ	**) , $**
T	**(, id**	**+ ,) , $**
T'	*** ,** ϵ	**+ ,) , $**
F	**(, id**	*** , + ,) , $**

2.2. Nonrecursive Predictive Parsing :-The nonrecursive parser in following figure lookup the production to be applied in a parsing table.



Model of a Nonrecursive Predictive Parsing

- **Construction of Predictive Parsing Table :**
  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 *b* in Follow(A).
  4. Make each undefined entry of M be error.

- **Predictive Parsing Program :**The parser is controlled by a program that behaves as follows:
  The program consider X- the symbol on top of the stack- and *a* – the current input symbol-. These two symbols determine the action of the parser. There are three possibilities :

1. If X = *a* = $, the parser halt, and 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 nonterminal , the program consults entry M[X,*a*] of the parsing table.If M[X,*a*]= { X⟶ UVW }the parser replaces X on top of stack by WVU (with U on top).

Example:

E→ E+T | T

T→ T*F | F // Original grammar

F→ (E) | id

Eliminate left-recursion and left factoring

E→ T E'

E' → +T E' | ϵ

T→ FT'

T'→ *F T' | ϵ

F→ (E) | id

Predictive Parsing Table M

Nonterminals	Input symbol					
	id	**+**	*****	**(**	**)**	**$**
E	TE'			TE'		
E'		+TE'			ϵ	ϵ
T	FT'			FT'		
T'		ϵ	*FT'		ϵ	ϵ
F	id			(E)		

Implement Predictive Parsing Program

stack	Input	output
$E	id+id*id$	
$E'T	id+id*id$	E ⟶ TE'
$E'T'F	id+id*id$	T ⟶ FT'
$E'T'id	id+id*id$	F ⟶ id
$E'T'	+id*id$	
$E'	+id*id$	T' ⟶ ϵ
$E'T+	+id*id$	E' ⟶ +TE'
$E'T	id*id$	
$E'T'F	id*id$	T ⟶ FT'
$E'T'id	id*id$	F ⟶ id
$E'T'	*id$	
$E'T'F*	*id$	T' ⟶ *FT'
$E'T'F	id$	
$E'T'id	id$	F ⟶ id
$E'T'	$	T' ⟶ ϵ
$E'	$	E' ⟶ ϵ
$	$	Accept

LL(1)Grammar :A grammar whose parsing table has no multiply-defined entries is said to be LL(1).

Example :- (H.W)

S ⟶ iEtSS' | a

S' ⟶ eS | ε

E ⟶ b