2.2 CONCEPT OF PARSING

Parsing:

It is the process of analyzing a continuous stream of input in order to determine its grammatical structure with respect to a given formal grammar.

Parse tree:

Graphical representation of a derivation or deduction is called a parse tree. Each interior node of the parse tree is a non-terminal; the children of the node can be terminals or non-terminals.

Types of parsing:

1. Top down parsing
2. Bottom up parsing

- Top–down parsing : A parser can start with the start symbol and try to transform it to the input string.

Example : LL Parsers.

- Bottom–up parsing : A parser can start with input and attempt to rewrite it into the start symbol.

Example : LR Parsers.

2.2.1 TOP-DOWN PARSING

It can be viewed as an attempt to find a left-most derivation for an input string or an attempt to construct a parse tree for the input starting from the root to the leaves.

Types of top-down parsing :

1. Recursive descent parsing
2. Predictive parsing

2.2.2 Recursive Descent Parsing

· Recursive descent parsing is one of the top-down parsing techniques that uses a set of recursive

procedures to scan its input.

· This parsing method may involve backtracking, that is, making repeated scans of the input.

Example for backtracking :

Consider the grammar G : S → cAd

A → ab |a and the input string w=cad.

The parse tree can be constructed using the following top-down approach :

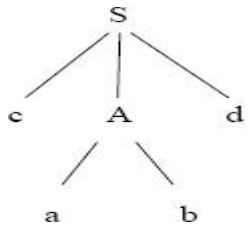
Step 1:

Initially create a tree with single node labeled S. An input pointer points to 'c', the first symbol of w.

Expand the tree with the production of S.

Step 2:

The leftmost leaf 'c' matches the first symbol of w, so advance the input pointer to the second symbol of w 'a' and consider the next leaf 'A'. Expand A using the first alternative.

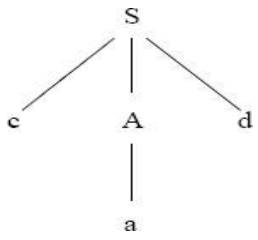


Step 3:

The second symbol 'a' of w also matches with second leaf of tree. So advance the input pointer to third symbol of w 'd'. But the third leaf of tree is b which does not match with the input symbol d. Hence discard the chosen production and reset the pointer to second position. This is called **backtracking**.

Step 4:

Now try the second alternative for A.



Now we can halt and announce the successful completion of parsing.

Example for recursive descent parsing:

A left-recursive grammar can cause a recursive-descent parser to go into an infinite loop.

Hence, **elimination of left-recursion** must be done before parsing.

Consider the grammar for arithmetic expressions

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

After eliminating the left-recursion the grammar becomes,

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

Now we can write the procedure for grammar as follows:

```
Procedure E()
Begin
    T();
    EPRIME();
End
Procedure EPRIME()
Begin
If input_symbol='+' then
ADVANCE();
T();
EPRIME();
End
Precedure T()
begin
```

```
F( );
TPRIME( );
end
ProcedureTPRIME( )
begin
If input_symbol='*' then
ADVANCE( );
F( );
TPRIME( );
end
ProcedureF( )
begin
If input-symbol='id' then
ADVANCE( );
else if input-symbol='(' then
ADVANCE( );
E( );
else if input-symbol=')' then
ADVANCE( );
end
else ERROR( );
```

Stack implementation:

To recognize input id+id*id :

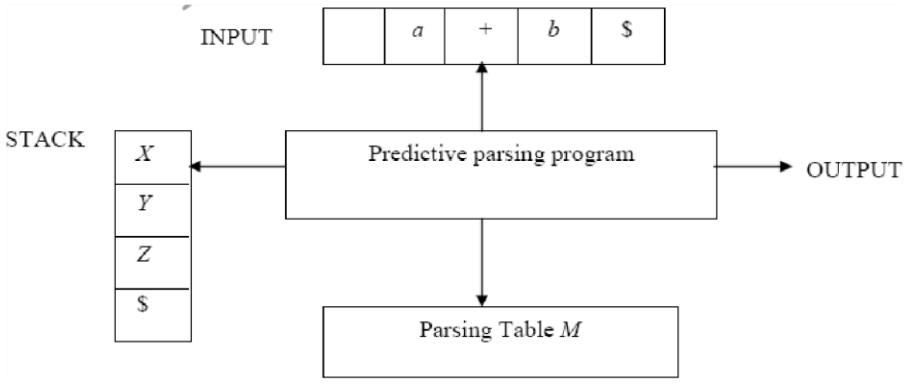
PROCEDURE	INPUT STRING
E( )	**id**+id*id
T( )	**id**+id*id
F( )	**id**+id*id
ADVANCE( )	id+id*id

TPRIME( )	id+id*id
EPRIME( )	id+id*id
ADVANCE( )	id+**id***id
T( )	id+**id***id
F( )	id+**id***id
ADVANCE( )	id+id*id
TPRIME( )	id+id*id
ADVANCE( )	id+id***id**
F( )	id+id***id**
ADVANCE( )	id+id***id**
TPRIME( )	id+id***id**

2.2.3 Predictive Parsing

Predictive parsing is a special case of recursive descent parsing where no backtracking is required. The key problem of predictive parsing is to determine the production to be applied for a non-terminal in case of alternatives.

Non-recursive predictive parser



The table-driven predictive parser has an input buffer, stack, a parsing table and output an stream.

Input buffer: It consists of strings to be parsed, followed by $ to indicate the end of the input string.

Stack: It contains a sequence of grammar symbols preceded by $ to indicate the bottom of the stack. Initially, the stack contains the start symbol on top of $.

2.2.4 Parsing Table:

It is a two-dimensional array M[A, a], where 'A' is a non-terminal and 'a' is a terminal.

Predictive parsing program:

The parser is controlled by a program that considers X, the symbol on top of stack, and a, the current input symbol. These two symbols determine the parser action. 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 non-terminal , the program consults entry M[X, a] of the parsing table M. This entry will either be an X-production of the grammar or an error entry.

If M[X, a] = {X → UVW},the parser replaces X on top of the stack by WVU.

If M[X, a] = **error**, the parser calls an error recovery routine.

2.2.5 Algorithm for nonrecursive predictive parsing:

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 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 as follows:

```
set ip to point to the first symbol of w$;
repeat let X be the top 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 non-terminal */
                if M[X, a] = X→ Y1Y2 … Yk then begin pop X from
                the stack; push Yk, Yk-1,…,Y1 onto the stack, with
                Y1 on top; output the production X → Y1 Y2 . . . Yk
                end
                else error()

until X = $     /* stack is empty */
```

2.2.6 Predictive parsing table construction:

The construction of a predictive parser is aided by two functions associated with a grammar G : 1. FIRST

2. FOLLOW

Rules for first( ):

1. If X is terminal, then FIRST(X) is {X}.

2. If X → is a production, then add to FIRST(X).

3. If X is non-terminal and X→ aa is a production then add a to FIRST(X).

4. If X is non-terminal and X → $Y_1Y_2 \ldots Y_k$ is a production, then place a in FIRST(X) if for some i, a is in FIRST(Y_i), and is in all of FIRST(Y_1),…,FIRST(Y_{i-1}); that is, $Y_1,\ldots Y_{i-1}$ => . If is in FIRST(Y_j) for all j=1,2,..,k, then add to FIRST(X).

Rules for follow( ):

1. If S is a start symbol, then FOLLOW(S) contains $.

2. If there is a production A → αBβ, then everything in FIRST(β) except is placed in follow(B).

3. If there is a production A → αB, or a production A → αBβ where FIRST(β) contains , then everything in FOLLOW(A) is in FOLLOW(B).

2.2.7 Algorithm for construction of predictive parsing table:

Input :

Gramma

r 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 is in FIRST(α), add A→ α to M[A, b] for each terminal b in FOLLOW(A). If is in FIRST(α) and $ is in FOLLOW(A) , add A→ α to M[A, $].

4. Make each undefined entry of M be error.

Example:

Consider the following grammar :

E → E+T |T

T → T*F |F

F → (E) |id

After eliminating left-recursion the grammar is

E → TE’

E’ → +TE’ |

T → FT’

T’ → *FT’ |

F

→

(E

)

|id

Fi

rs

t(

) :

FIRST(E) = { (, id}

FIRST(E’) ={+ , }

FIRST(T) = { (, id}

FIR

ST(

T’) =

{*, }

FIRS

T(F)

= { (

, id }

Foll

ow(

):

FOLLOW(E) = { $,) }

FOLLOW(E’) = { $,) }

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

FOLLOW(T’) = { +, $,) }

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

Predictive parsing table :

NON-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)		

Stack implementation:

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'	$	
$E'	$	T' → ε
$	$	E' → ε

2.2.8 LL(1) grammar:

The parsing table entries are single entries. So each location has not more than one entry.

This type of grammar is called LL(1) grammar.

Example:

Consider this following grammar:

S→ iEtS | iEtSeS | a

E→ b

After eliminating left factoring, we have

S→ iEtSS' | a

S'→ eS |

E→ b

To construct a parsing table, we need FIRST() and FOLLOW() for all the non-terminals.

FIRST(S) = { i, a }

FIRST(S') = {e, }

FIRST(E) = { b}

FOLLOW(S) = { $,e }

FOLLOW(S') = { $,e }

FOLLOW(E) = {t}

Parsing table:

NON-TERMINAL	a	b	e	i	t	$
S	S → a			S → iEtSS'		
S'			S' → eS S' → ε			S' → ε
E		E → b				

Since there are more than one production, the grammar is not LL(1) grammar.

Actions performed in predictive parsing:

1. Shift
2. Reduce
3. Accept
4. Error

Implementation of predictive parser:

1. Elimination of left recursion, left factoring and ambiguous grammar.
2. Construct FIRST() and FOLLOW() for all non-terminals.
3. Construct predictive parsing table.
4. Parse the given input string using stack and parsing table.