Assignment No.10

AIM:

Implement the non-recursive predictive parser for the language.

THEORY:

Top down parsing

Top-down parsing can be viewed as an attempt to find a leftmost derivation for an input string. Equivalently, it can be viewed as an attempt to construct a parse tree for the input starting from the root and creating the nodes of the parse tree in preorder.

Following are the examples of top-down parser

- Recursive decent parser
- Predictive parser
- Non-recursive predictive parser

Non- Recursive Predictive parser

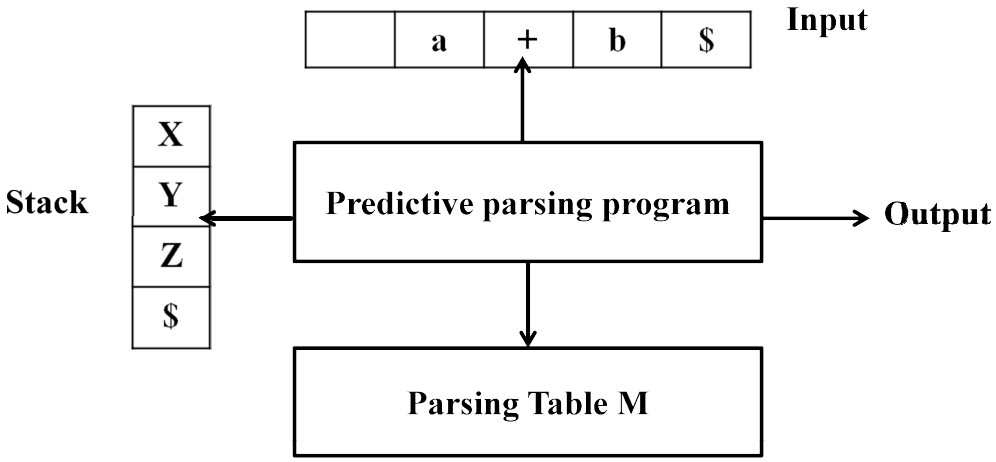

Figure 1: Model of non-recursive predictive parser

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

A table-driven predictive parser has an input buffer, a stack, a parsing table, and an output stream. The input buffer contains the string to be parsed, followed by $, a symbol used as a right end-marker to indicate the end of the input string. The stack contains a sequence of grammar symbols with $ on the bottom, indicating the bottom of the stack. Initially, the stack contains the start symbol of the grammar on top of $. The parsing table is a two dimensional array M[A, a], where capital A is a nonterminal, and small a is a terminal or the symbol $.

In parsing table, each row is a non-terminal symbol, each column is a terminal symbol or the special symbol $ and each entry holds a production rule.

The parser is controlled by a program that behaves as follows. The program considers 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 four possible parser actions.

1. If X = a = $ then the parser halts and announces successful completion of parsing
2. If X and a are the same terminal symbol (different from $) that is X = a which is not equal to $ then parser pops X from the stack, and advances the input pointer to the next input symbol,
3. If X is a non-terminal, the program consults the parsing table entry M[X,a]. If M[X,a] holds a production rule X→Y1Y2...Yk, it pops X from the stack and pushes Yk,Yk-1,...,Y1 into the stack. The parser also outputs the production rule X→Y1Y2...Yk to represent a step of the derivation.
4. If none of the above then error. All empty entries in the parsing table are errors and the parser calls an error recovery routine. If X is a terminal symbol different from a, this is also an error case.

Non- Recursive Predictive parser 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 input buffer and the start symbol S of grammar G on top of the stack, above $. The flowchart shown below uses the predictive parsing table M to produce a predictive parse for the input.

set ip to point to the first symbol of w;

```
repeat
        Let X be the top stack symbol and a the symbol pointed 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-i,... ,Y1 onto the stack, with Y1 on top;
                        output the production X -> Y1 Y2 • • • Yk;
                end
                else error()
until X=$ /* stack is empty */
```

Consider the grammar given below. A predictive parsing table for this grammar is shown in Fig. In parsing table, Blanks are error entries while non-blanks indicate a production with which to expand the top nonterminal on the stack. The input string is id+id.

With input id + id; the predictive parser makes the sequence of moves shown below. Initially, the parser is in a configuration with w$ that is id+id$ in the input buffer and the start symbol E of grammar G on top of the stack, above $. The input pointer points to the leftmost symbol of the string in the INPUT column. The parser traces out a Leftmost derivation for the Input, that is, the productions output are those of a leftmost derivation. The input symbols that have already been scanned, followed by the grammar symbol on the stack (from the top to bottom), make up the left-sentential forms in the derivation.

Finally stack as well as input contains the $ hence the parser halts and announces successful completion of parsing

Example 1:

E → T E'
E' → +T E' | ε
T → F T'
T' → *F T' | ε
F → id | (E)

Input String : id + id

	id	+	*	(	)
E	E → TE'			E → TE'	
E'		E' → +TE'			E' → ε
T	T → FT'			T → FT'	
T'		T' → ε	T' → *FT'		T' → ε

stack	input	output
$E	id+id$	E → TE'
$E'T	id+id$	T → FT'
$E' T'F	id+id$	F → id
$ E' T'id	id+id$	
$ E' T'	+id$	T' → ε
$ E'	+id$	E' → +TE'
$ E' T+	+id$	
$ E' T	id$	T → FT'
$ E' T' F	id$	F → id
$ E' T'id	id$	
$ E' T'	$	T' → ε
$ E'	$	E' → ε
$	$	accept

PROGRAM:

INPUT:

This parser has been made for the grammar given by

E --> Te
e --> +Te | N
T --> Ft
t --> *Ft | N
F --> (E) | i
E , F , T , t , e , N are nonterminals
* , + , i , (,) are terminals

OUTPUT:

Top down parsing using the TABLE DRIVEN/PREDICTIVE/LL(1) PARSER
==========

First of E is given by (i
First of e is given by +N
First of T is given by (i
First of t is given by *N
First of F is given by (i
First of i is given by i
First of + is given by +
First of * is given by *
First of (is given by (
First of) is given by)
First of $ is given by $

Follow of E is given by)$
Follow of e is given by)$
Follow of T is given by +)$
Follow of t is given by +)$
Follow of F is given by *+)$

Enter the string(Use i for an identifier) (i+i)*i

$E	(i+i)*i$
$eT	(i+i)*i$
$etF	(i+i)*i$
$et)E(	(i+i)*i$
$et)E	i+i)*i$
$et)eT	i+i)*i$
$et)etF	i+i)*i$
$et)eti	i+i)*i$
$et)et	+i)*i$
$et)e	+i)*i$
$et)eT+	+i)*i$
$et)eT	i)*i$
$et)etF	i)*i$
$et)eti	i)*i$
$et)et	)*i$
$et)e	)*i$
$et)	)*i$
$et	*i$
$etF*	*i$
$etF	i$
$eti	i$
$et	$
$E	(i+i)*i$
$eT	(i+i)*i$
$etF	(i+i)*i$
$et)E(	(i+i)*i$
$et)E	i+i)*i$
$et)eT	i+i)*i$
$et)etF	i+i)*i$
$et)eti	i+i)*i$
$et)et	+i)*i$
$et)e	+i)*i$
$et)eT+	+i)*i$

$et)eT	i)*i$
$et)etF	i)*i$
$et)eti	i)*i$
$et)et	)*i$
$et)e	)*i$
$et)	)*i$
$et	*i$
$etF*	*i$
$etF	i$
$eti	i$
$et	$
$E	(i+i)*i$
$eT	(i+i)*i$
$etF	(i+i)*i$
$et)E(	(i+i)*i$
$et)E	i+i)*i$
$et)eT	i+i)*i$
$et)etF	i+i)*i$
$et)eti	i+i)*i$
$et)et	+i)*i$
$et)e	+i)*i$
$et)eT+	+i)*i$
$et)eT	i)*i$
$et)etF	i)*i$
$et)eti	i)*i$
$et)et	)*i$
$et)e	)*i$
$et)	)*i$
$et	*i$
$etF*	*i$
$etF	i$
$eti	i$
$et	$
$e	$

Accepted

CONCLUSION:

Thus the non-recursive predictive parser for given grammar is implemented.

REFERENCES:

- Compilers - Principles, Techniques and Tools - A.V. Aho, R. Shethi and J. D. Ullman (Pearson Education)