Experiment No 3

Aim : Design recursive descent parser

Theory :

A **recursive descent parser** is a kind of top-down parser built from a set of mutually-recursive procedures (or a non-recursive equivalent) where each such procedure usually implements one of the production rules of the grammar. Thus the structure of the resulting program closely mirrors that of the grammar it recognizes.

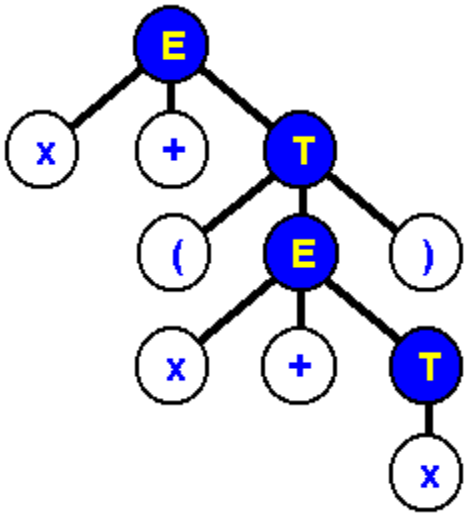
This parser attempts to verify that the syntax of the input stream is correct as it is read from left to right. A basic operation necessary for this involves reading characters from the input stream and matching then with terminals from the grammar that describes the syntax of the input. Our recursive descent parsers will look ahead one character and advance the input stream reading pointer when proper matches occur.

What a recursive descent parser actually does is to perform a depth-first search of the derivation tree for the string being parsed. This provides the 'descent' portion of the name. The 'recursive' portion comes from the parser's form, a collection of recursive procedures.

As our first example, consider the simple grammar

E ® x+T
T ® (E)
T ® x

and the derivation tree in figure 2 for the expression x+(x+x)



Derivation Tree for x+(x+x)

A recursive descent parser traverses the tree by first calling a procedure to recognize an E. This procedure reads an 'x' and a '+' and then calls a procedure to recognize a T. This would look like the following routine.

```
Procedure E()
Begin
If (input_symbol='x') then
next();
If (input_symbol='+') then
Next();
T();
Else
Errorhandler();
END
```

Procedure for E

Note that the 'next' looks ahead and always provides the next character that will be read from the input stream. This feature is essential if we wish our parsers to be able to predict what is due to arrive as input.

Note that 'errorhandler' is a procedure that notifies the user that a syntax error has been made and then possibly terminates execution.

In order to recognize a T, the parser must figure out which of the productions to execute. This is not difficult and is done in the procedure that appears below.

```
Procedure T()
Begin
Begin
If (input_symbol='(') then
next();
E();
If (input_symbol=')') then
next();
end
else If (input_symbol='x') then
next();
else
Errorhandler();
END
```

Procedure for T

In the above routine, the parser determines whether T had the form (E) or x. If not then the error routine was called, otherwise the appropriate terminals and nonterminals were recognized.

Algorithm:

1. Make grammar suitable for parsing i.e. remove left recursion(if required).

2. Write a function for each production with error handler.

3. Given input is said to be valid if input is scanned completely and no error function is called.

```
PS C:\Users\ACER\Documents\SPCC\EXP_3> cd "c:\Users\ACER\Documents\SPCC\EXP_3\" ; if ($?) { g++ main.cpp -o main } ; if ($?) { .\main }
Enter input
i*i
Input Accepted
PS C:\Users\ACER\Documents\SPCC\EXP_3> cd "c:\Users\ACER\Documents\SPCC\EXP_3\" ; if ($?) { g++ main.cpp -o main } ; if ($?) { .\main }
Enter input
i+(i*i)
Input Accepted
```

```
PS C:\Users\ACER\Documents\SPCC\EXP_3> cd "c:\Users\ACER\Documents\SPCC\EXP_3\" ; if ($?) { g++ main.cpp -o main } ; if ($?) { .\main }
Enter input
i/i
Input Accepted
PS C:\Users\ACER\Documents\SPCC\EXP_3> cd "c:\Users\ACER\Documents\SPCC\EXP_3\" ; if ($?) { g++ main.cpp -o main } ; if ($?) { .\main }
Enter input
(i*i
ERROR
Input Rejected
```

Conclusion:

1. What is left Recursion ? Write the rules for removing left recursion.
2. What is left factoring ? Write rules for eliminating left factoring.
3. Difference between top down and Bottom up parsing