Department of Computer Engineering

Academic Term : Jan-May 23-24

Class : T.E. (Computer)
Subject Name : System Programming and Compiler Construction
Subject Code : (CPC601)

Practical No:	3
Title:	Design recursive descent parser.
Date of Performance:	
Date of Submission:	
Roll No:	9601
Name of the Student:	Ivan Dsouza

Evaluation:

Sr. No	Rubric	Grade
1	Time Line (2)	
2	Output(3)	
3	Code optimization (2)	
4	Postlab (3)	

Signature of the Teacher :

FR. CONCEICAO RODRIGUES COLLEGE OF ENGINEERING

System Programming and Compiler Construction

VI Semester (Computer) Academic Year: 22-23

Experiment No 3

- *Ivan Dsouza*
- *9601*
- *T.E. Comps A (Batch C)*

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.

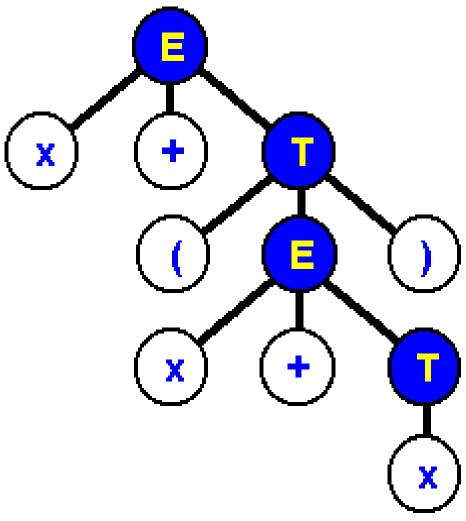
This parser attempts to verify that the syntax of the input stream is correct as it is read from left to right. A basic operation necessary for this involves reading characters from the input stream and matching then with terminals from the grammar that describes the syntax of the input. Our recursive 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
```

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.

Conclusion: Recursive Descent Parser was designed and performed. Also the corresponding test cases were run to test its functioning.

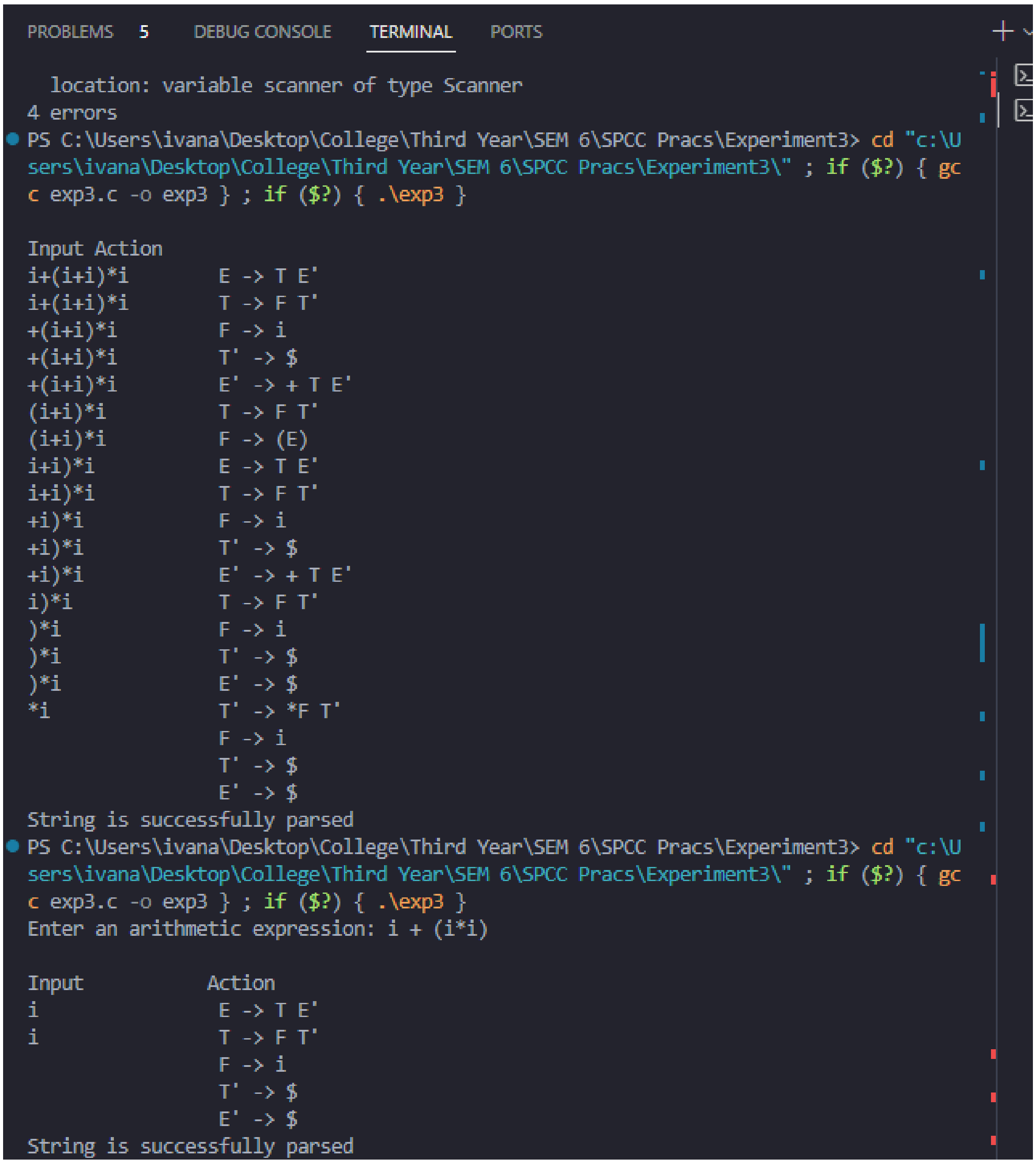
1. What is left Recursion ? Write the rules for removing left recursion.
2. What is left factoring ? Write rules for eliminating left factoring.
3. Difference between top down and Bottom up parsing

Source Code:

```
#include <stdio.h>
#include <string.h>
#define SUCCESS 1
#define FAILED 0

int E(), Edash(), T(), Tdash(), F();

const char *pt;
char grammar[64];

int main() {
    printf("Enter an arithmetic expression: ");
    scanf("%s", grammar);
    pt = grammar;
    puts("");
    puts("Input\t\tAction");
```

```c
    if (E() && *pt == '\0') {
        puts("String is successfully parsed");
        return 0;
    } else {
        puts("Error in parsing String");
        return 1;
    }
}

int E() {
    printf("%-16s E -> T E'\n", pt);
    if (T()) {
        if (Edash())
            return SUCCESS;
        else
            return FAILED;
    } else
        return FAILED;
}

int Edash() {
    if (*pt == '+') {
        printf("%-16s E' -> + T E'\n", pt);
        pt++;
        if (T()) {
            if (Edash())
                return SUCCESS;
            else
                return FAILED;
        } else
            return FAILED;
    } else
        printf("%-16s E' -> $\n", pt);
    return SUCCESS;
}

int T() {
    printf("%-16s T -> F T'\n", pt);
    if (F()) {
        if (Tdash())
            return SUCCESS;
        else
            return FAILED;
    } else
        return FAILED;
}

int Tdash() {
    if (*pt == '*') {
        printf("%-16s T' -> *F T'\n", pt);
        pt++;
        if (F()) {
            if (Tdash())
                return SUCCESS;
            else
                return FAILED;
        } else
            return FAILED;
    } else {
        printf("%-16s T' -> $\n", pt);
        return SUCCESS;
    }
}

int F() {
    if (*pt == '(') {
        printf("%-16s F -> (E)\n", pt);
        pt++;
        if (E()) {
            if (*pt == ')') {
                pt++;
                return SUCCESS;
            } else
                return FAILED;
        } else
            return FAILED;
    } else if (*pt == 'i') {
        pt++;
        printf("%-16s F -> i\n", pt);
        return SUCCESS;
    } else
        return FAILED;
}
```

Output:



PROBLEMS 5   DEBUG CONSOLE   TERMINAL   PORTS

```
  location: variable scanner of type Scanner
4 errors
PS C:\Users\ivana\Desktop\College\Third Year\SEM 6\SPCC Pracs\Experiment3> cd "c:\U
sers\ivana\Desktop\College\Third Year\SEM 6\SPCC Pracs\Experiment3\" ; if ($?) { gc
c exp3.c -o exp3 } ; if ($?) { .\exp3 }

Input Action
i+(i+i)*i        E -> T E'
i+(i+i)*i        T -> F T'
+(i+i)*i         F -> i
+(i+i)*i         T' -> $
+(i+i)*i         E' -> + T E'
(i+i)*i          T -> F T'
(i+i)*i          F -> (E)
i+i)*i           E -> T E'
i+i)*i           T -> F T'
+i)*i            F -> i
+i)*i            T' -> $
+i)*i            E' -> + T E'
i)*i             T -> F T'
)*i              F -> i
)*i              T' -> $
)*i              E' -> $
*i               T' -> *F T'
                 F -> i
                 T' -> $
                 E' -> $
String is successfully parsed
PS C:\Users\ivana\Desktop\College\Third Year\SEM 6\SPCC Pracs\Experiment3> cd "c:\U
sers\ivana\Desktop\College\Third Year\SEM 6\SPCC Pracs\Experiment3\" ; if ($?) { gc
c exp3.c -o exp3 } ; if ($?) { .\exp3 }
Enter an arithmetic expression: i + (i*i)

Input           Action
i                E -> T E'
i                T -> F T'
                 F -> i
                 T' -> $
                 E' -> $
String is successfully parsed
```

Postlab:

IVAN DSOUZA
9601
T.E. COMPS A
BATCH C

EXPERIMENT 3 POSTLAB

Ans1) Left Recursion occurs in a grammar when a non-terminal A can derive a string that starts with itself, leading to infinite recursion.

~~Replace~~
To eliminate left recursion

i) Replace productions of the form $A \rightarrow A\alpha | B$ with $A \rightarrow BA'$ where A' is a new non-terminal

ii) Add production $A' \rightarrow \alpha A' | \epsilon$, where α does not start with A.

Ans2) Left factoring is a technique to eliminate common prefixes in the production of a grammar.

To left factor:

i) Identify common prefixes in the productions and factor them out

ii) ~~Factor~~ Factor out the prefix: Create a new rule with the common prefix as the RHS and a new non-terminal on the LHS.

iii) Rewrite the original production: Replace the common prefix in each original production with the new non-terminal

3) Difference between Top-Down and BottomUp parsing.

Top Down	Bottom up
i) Starts from the root (NT) and expands it downwards, matching input token	i) Starts from the leaves (terminals) and builds upwards, constructing a parse tree.
ii) Can be faster for smaller grammars and deterministic inputs. ~~[illegible]~~	ii) More efficient for ambiguous grammars and complex input
iii) Requires storing of stack of non-terminals and ~~potentially [illegible]~~ potential choices.	iii) ~~[illegible]~~ Less memory intensive

iv) Good for languages hierarchi

v) May back measure ineffecien

terminal or

: Replace
ent original
on-terminal

n and

ttom up

ts from the
s (terminals)
ards upwards,
uting a parse

ferent
uous grammars
mplex inputs.

mmars
ive

iv) Good for context free languages with clear hierarchical structures	iv) Good for ambiguous grammars and complex input handling.
v) May backtrack upon mismatch, potentially inefficient.	v) Handles errors more ~~eff~~ gracefully