Eastern Mediterranean University

CMSE318-CMPE410 Principles of Programming Languages
Spring 2020-2021

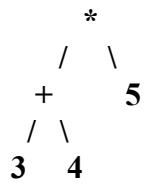
Assignment #4

Parser Application, part II

To be done in groups of two. Pick your partner!

This is a continuation of the previous assignment. Now, we are going to generate an abstract syntax tree of the input, and evaluate the tree to compute a value for the expression. We can consider the abstract syntax tree as the meaning of the expression that was parsed.

An abstract syntax tree is like a parse tree, except at the nodes we have operators, rather than non-terminals. For example, the abstract syntax tree for (3+4)*5 is given below. Note that there is no need for parenthesis in the tree.



Our original grammar remains unchanged.

G1:
G → E
E → E + T | E - T | T
T → T * F | T / F | F
F → (E) | M | N
M → a | b | c | d
N → 0 | 1 | 2 | 3

This grammar is not suitable for top down recursive descent parsing because of left recursion.

Removing left recursion, we get the grammar G2:

G2:
G → E

E → T R
R → + T R | - T R | ∈
T → F S
S → * F S | / F S | ∈
F → (E) | M | N
M → a | b | c | d
N → 0 | 1 | 2 | 3

"∈" means the empty production, i.e. if T → ∈, then T can derive the empty string.

Now, we can write recursive functions for parsing according to G2, generating an abstract syntax tree as a result of parsing, and evaluating the returned abstract syntax tree to compute the value of the expression. We shall assume that "a" is a constant with value 10, "b" is a constant with value 20, "c" is a constant with value 30 and "d" is a constant with value 40.

Below is the pseudo-code of the parser, tree printer and evaluator. We shall need a way to represent trees, hence the class "Node".

```
Class Node{
    char symbol;   // either an operator, or one of 0,1,2,3,a,b,c,d
    Node leftChild;  // will be NULL if node is a leaf
    Node rightChild;  // will be NULL if node is a leaf
}
```

```
Bool error = FALSE;
char next_token = '%';

void main(){
  Node theTree;
  /* open file for reading */
   ……
    theTree = G();
    if (not error){
        printTree(theTree);
        value = evaluate(theTree);
        print "The value is", value;
    }
  else print "input not parsed correctly"
}
```

```
/* G -> E */

Node G(){
  Node tree;

   lex();
   print ("G -> E");
   tree = E();
   if (next_token=='$'  and !(error))  {
          print "success";
         return tree;
   }
   else {
          print ("failure: unconsumed input=%s", unconsumed_input());
          return NULL;
   }
}



/* E -> T R */
Node E(){
       Node temp;

        if (error) return NULL;
        print ("E -> T R");

       temp = T();
       return R(temp);
}



/* R -> + T R | - T R | e */
Node R(Node tree){
    Node temp1, temp2;

    if (error) return NULL;

    if (next_token== '+')  {
       print ("R -> + T R");
       lex();
```

```
        temp1 = T();
        temp2 = R(temp1);
        return new Node('+', tree, temp2);
    }
 else if (next_token== '-')  {
       print ("R -> - T R");
       lex();
       temp1=T();
       temp2 =R(temp1);
       return new Node('-', tree, temp2);

    }
  else  {
      print ("R->e");
      return(tree);
    }
}



/* T -> F S */
Node T(){
   Node temp;

     if (error) return;
      print (" T -> F S");
      temp =  F();
      return   S(temp);
}



/*            S -> * F S | / F S | e          */
Node S(Node tree){
  Node temp1, temp2;

  if (error) return;
   if (next_token=='*') {
     print ("S -> * F S");
       lex();

       temp1=F();
       temp2=S(temp1);
       return new Node('*', tree, temp2);
   }
   else if (next_token=='/') {
```

```
    print "S -> / F S"
      lex();
      temp1=F();
      temp2=S(temp1);
      return new Node('/',tree,temp2);
   }
  else {
     print "S -> e";
     return(tree);
   }
}



/*        F -> ( E ) | N | M   */
Node F(){
   Node temp;

    if (error) return NULL;

   if (next_token=='(' )  {
      print "F->( E )";
      lex();
      temp=E();
      if (next_token == ')'  ){
         lex();
         return(temp);
      }
     else {  error=TRUE;
               print("error: unexptected token ", next_token);
               print("unconsumed_input ", unconusmed_input());
               return NULL;
     }
   }
   else  if  (next_token is one of 'a' or 'b' or 'c' or 'd') {
     print ("F->M");
     return (M());
   }
   else  if  (next_token is one of '0' or '1' or '2' or '3') {
     print ("F->N");
     return(N());
   }

   else {
        error=TRUE;
        print("error: unexptected token ", next_token);
```

```
        print("unconsumed_input ", unconusmed_input());
        return(NULL);

    }
}


/* M → a | b | c | d */

Node M(){
  char prev_token = next_token;

  if (error) return NULL;
  if  (next_token is one of  'a' or 'b' or 'c' or 'd') {
          print ("M->", next_token);
         lex();
         return new Node(prev_token, NULL,NULL);
  } else {
        error=TRUE;
        print("error: unexptected token ", next_token);
        print("unconsumed_input ", unconusmed_input());
        return(NULL);
  }
}

/* N → 0 | 1 | 2 | 3 */

Node N(){
  char prev_token = next_token;

  if (error) return NULL;

  if  (next_token is one of  '0' or '1' or '2' or '3') {
          print ("N->", next_token);
          lex();
          return new Node(prev_token, NULL,NULL);
  } else {
        error=TRUE;
        print("error: unexptected token ", next_token);
        print("unconsumed_input ", unconusmed_input());
        return(NULL)
  }
}
```

```
// print tree in postfix notation
void printTree(Node tree){
     if (tree==NULL) return;
    printTree(tree.leftChild);
    printTree(tree.rightChild);
    print(tree.symbol);
}


// compute the value of the expression
int evaluate(Node tree){
     if (tree==NULL) return -1;
     if (tree.symbol='a') return 10;
     if (tree.symbol='b') return 20;
     if (tree.symbol='c') return 30;
     if (tree.symbol='d') return 40;
     if (tree.symbol= one of 0,1,2,3) return the value of tree.symbol;
     if (tree.symbol= '+') return (evaluate(tree.leftChild) + evaluate(tree.rightChild));
     if (tree.symbol= '-') return (evaluate(tree.leftChild) - evaluate(tree.rightChild));
     if (tree.symbol= '*') return (evaluate(tree.leftChild) * evaluate(tree.rightChild));
     if (tree.symbol= '/') return (evaluate(tree.leftChild) / evaluate(tree.rightChild));
}
```

OK, now that the parser is mostly written for you, here is what you will do:

Write a Python program (based on the pseudo-code given above) to parse expressions as defined by the grammar above, print the abstract syntax tree and compute its value. The input should be in a file. You should have global variables *error* (of type *Boolean*), and *next_token* (of type *char*). Define a function *lex()* that gets the next character from the file and places it inside *next_token*. *lex()* should skip any white spaces, such as newlines or the space character. The function *unconusmed_input()* should return the remaining input in the file. The last character in the file should always be $. Define functions (hint: class methods) *G()*, *E()*, *R()*, *T()*, *F()* and *N()*. Inside *main()*, open the file containing the expression and call *G()*.

What to hand in:

Zip the following and upload to Moodle.

1. **A report containing**
    a. **a description of the problem (what your program does)**

    b. **description of your program (how it does what it does)**
    c. **tutorial on running your program**
    d. **5 sample runs that parse correctly (i.e. 5 different inputs that produce no error)**
    e. **5 sample runs that produce errors (5 different inputs that produce errors)**
2. **Commented source code**

The filename of your uploaded file should be of the format:
studentID1_studentID2_CMSE318_CMPE410_Assignment4.zip