Contributor: Dennis George Degree: 3

Problem 2. Implement the recursive-descent parser of FCS section 11.6 constructing parse trees of the grammar of balanced parentheses (FCS figure 11.25; students are allowed to use directly the code of FCS section 11.6). After constructing a parse tree your algorithm should compute its height and list all labels in pre-order and post-order. Demonstrate with examples that your code operates properly.

All programming targets were achieved to the 3rd degree. Below is proof of program correctness.

The textbook used “()()” as the input in order to demonstrate the grammar of balanced parenthesis.

The code that was used in our solution came from the textbook in section 11.6 of FCS and is attached below. The program takes the input string as a command line argument.

```c
#include <stdlib.h>
#include <stddef.h>
#include <stdio.h>

#define FAILED NULL
typedef struct NODE* TREE;

struct NODE {
   char label;
   TREE leftmostChild, rightSibling;
};

TREE makeNode0(char x);
TREE makeNode1(char x, TREE t);
TREE makeNode4(char x, TREE t1, TREE t2, TREE t3, TREE t4);
TREE B();
TREE parseTree; /* holds the result of the parse */
char *nextTerminal; /* current position in input string */
int computeHeight(TREE tree);
void printPreOrder(TREE tree);
void printPostOrder(TREE tree);
void main(int argc, char *argv[])
{
   nextTerminal = argv[1]; /* in practice, a string
                              of terminals would be read from input */
   parseTree = B();
   if(parseTree!=NULL){
      int h = computeHeight(parseTree);
      printf("height: %d\n", h);
      printf("preorder:\n");
      printPreOrder(parseTree);
      printf("\n");
      printf("postorder:\n");
      printPostOrder(parseTree);
      printf("\n");
   } else
        printf("failed to make tree\n");

}
```

```c
int computeHeight(TREE tree)
{
    if(tree==NULL)
        return 0;
    else{
        int lDepth = computeHeight(tree->leftmostChild);
        int rDepth = computeHeight(tree->rightSibling);
        if(lDepth > rDepth)
            return lDepth+1;
        else return rDepth+1;
    }
}
void printPreOrder(TREE tree)
{
    if(tree!=NULL){
        printf("%c ", tree->label);
        printPreOrder(tree->leftmostChild);
        printPreOrder(tree->rightSibling);
    }
}
void printPostOrder(TREE tree)
{
    if(tree!=NULL){
        printPreOrder(tree->leftmostChild);
        printPreOrder(tree->rightSibling);
        printf("%c ", tree->label);
    }
}
TREE makeNode0(char x)
{
   TREE root;
   root = (TREE) malloc(sizeof(struct NODE));
   root->label = x;
   root->leftmostChild = NULL;
   root->rightSibling = NULL;
   return root;
}
TREE makeNode1(char x, TREE t)
{
   TREE root;
   root = makeNode0(x);
   root->leftmostChild = t;
   return root;
}
```

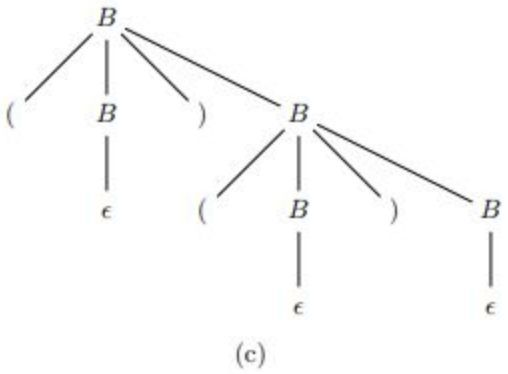
```
TREE makeNode4(char x, TREE t1, TREE t2, TREE t3, TREE t4)
{
   TREE root;
   root = makeNode1(x, t1);
   t1->rightSibling = t2;
   t2->rightSibling = t3;
   t3->rightSibling = t4;
   return root;
}

TREE B()
{
   TREE firstB, secondB;
   if(*nextTerminal == '(') /* follow production 2 */ {
      nextTerminal++;
      firstB = B();
      if(firstB != FAILED && *nextTerminal == ')') {
         nextTerminal++;
         secondB = B();
         if(secondB == FAILED)
            return FAILED;
         else
            return makeNode4('B',
                  makeNode0('('),
                  firstB,
                  makeNode0(')'),
                  secondB);
      }
      else /* first call to B failed */
         return FAILED;
   }
   else /* follow production 1 */
      return makeNode1('B', makeNode0('e'));
}
```

When running the program with “()()” as input, the following output is generated.

```
dennis: problem2 > ./a.out "()()"
height: 10
preorder:
B ( B e ) B ( B e ) B e
postorder:
( B e ) B ( B e ) B e B
```

Comparing this to the tree from the textbook, the program can be confirmed to be constructing correctly.



(c)

Fig. 11.29. Trees constructed by recursive calls to B.

Below are attached screenshots of more tests. By manually building and tracing the trees for the rest of the examples, we were able to confirm that the program was working as expected.

```
dennis: problem2 > ./a.out "(()"
failed to make tree
```

```
dennis: problem2 > ./a.out "(((()())))"
height: 14
preorder:
B ( B ( B ( B e ) B ( B e ) B e ) B e ) B e
postorder:
( B ( B ( B e ) B ( B e ) B e ) B e ) B e B
```

```
dennis: problem2 > ./a.out "()()()(()())()()(()()(()))"
height: 42
preorder:
B ( B e ) B ( B e ) B ( B e ) B ( B ( B e ) B ( B e ) B e ) B ( B e ) B ( B e ) B ( B ( B e ) B ( B e ) B ( B ( B e ) B e ) B e ) B e
postorder:
( B e ) B ( B e ) B ( B e ) B ( B ( B e ) B ( B e ) B e ) B ( B e ) B ( B e ) B ( B ( B e ) B ( B e ) B ( B ( B e ) B e ) B e ) B e B
```

```
dennis: problem2 > ./a.out "(((())()))"
height: 12
preorder:
B ( B ( B ( B e ) B e ) B ( B e ) B e ) B e
postorder:
( B ( B ( B e ) B e ) B ( B e ) B e ) B e B
```