Eric Roberts
CS106B

Handout #47
May 17, 2009

Section #7—Trees

For problems 1, 2, and 3, assume that `nodeT` is defined as follows:

```
struct nodeT {
    string key;
    nodeT *left, *right;
};
```

1. Tracing the binary tree insertion algorithm

Diagram the tree that results if you insert the strings `"one"`, `"two"`, `"three"`, `"four"`, `"five"`, `"six"`, `"seven"`, `"eight"`, `"nine"`, and `"ten"` into an empty tree in that order. Once you have finished, answer the following questions about your diagram:

1a. What is the height of the resulting tree?

1b. Which nodes are leaves?

1c. Which nodes, if any, are out of balance?

1d. Which key comparisons would be made to find the string `"seven"` in the tree?

2. Calculating the height of a binary tree (Chapter 13, exercise 6, page 485)

Write a function

```
int Height(nodeT *tree);
```

that takes a binary search tree and returns its height.

3. Checking whether a tree is balanced (Chapter 13, exercise 7, page 485)

Write a function

```
bool IsBalanced(nodeT *tree);
```

that determines whether a given tree is balanced according to the definition in the section on "Balanced trees." To solve this problem, all you really need to do is translate the definition of a balanced tree more or less directly into code. If you do so, however, the resulting implementation is likely to be quite inefficient because it has to make several passes over the tree. The real challenge in this problem is to implement the `IsBalanced` function so that it determines the result without looking at any node more than once.

4. Changing the interpreter into a compiler (Chapter 14, exercise 6, page 522)

Although the interpreter program that appears in this chapter is considerably easier to implement than a complete compiler, it is possible to get a sense of how a compiler works by defining one for a simplified computer system called a *stack machine*. A stack machine performs operations on an internal stack, which is maintained by the hardware, in much the same fashion as the calculator described in Chapter 4. For the purposes of this problem, you should assume that the stack machine can execute the operations shown in Figure 1 at the top of the next page.

Write a function

```
void Compile(ifstream & in, ofstream & out);
```

Figure 1. Stack machine instructions

LOAD #*n*	Pushes the constant *n* on the stack.
LOAD *var*	Pushes the value of the variable *var* on the stack.
STORE *var*	Stores the top stack value in *var* without actually popping it.
DISPLAY	Pops the stack and displays the result.
ADD **SUB** **MUL** **DIV**	These instructions pop the top two values from the stack and apply the indicated operation, pushing the final result back on the stack. The top value is the right operand, the next one down is the left.

that reads expressions from `in` and writes to `out` a sequence of instructions for the stack-machine that have the same effect as evaluating each of the expressions in the input file and displaying their result. For example, if the file opened as `in` contains

```
x = 7
y = 5
2 * x + 3 * y
```

calling `Compile(in, out)` should write the following code to `out`:

```
LOAD #7
STORE x
DISPLAY
LOAD #5
STORE y
DISPLAY
LOAD #2
LOAD x
MUL
LOAD #3
LOAD y
MUL
ADD
DISPLAY
```

5. Constant folding (Chapter 14, exercise 7, page 523)

After it parses an expression, a commercial compiler typically looks for ways to simplify that expression so that it can be computed more efficiently. This process is called **optimization.** One common technique used in the optimization process is **constant folding,** which consists of identifying subexpressions that are composed entirely of constants and replacing them with their value. For example, if a compiler encountered the expression

```
days = 24 * 60 * 60 * sec
```

there would be no point in generating code to perform the first two multiplications when the program was executed. The value of the subexpression `24 * 60 * 60` is constant and might as well be replaced by its value (86400) before the compiler actually starts to generate code.

Write a function `FoldConstants(exp)` that takes an `expressionT` and returns a new `expressionT` in which any subexpressions that are entirely composed of constants are replaced by a new constant node containing the computed value.