Homework 10

Due: April 20, 11pm

For the final project, you generate code for expressions as discussed in class: the `codeGen` method for each kind of expression will generate code to evaluate the expression, leaving the value on the stack.

While this kind of code is easy to generate, storing intermediate values on the stack rather than in registers is inefficient. You might think we could instead require that the `codeGen` method for an expression should work as follows:

1. Each expression node's `codeGen` method would have 1 parameter: a register number N, which would be either 0 or 1.
2. For literals or identifiers, the `codeGen` method would simply load the appropriate value in register N.
3. For expressions involving non short-circuited binary operators (+, *, <, etc.) the `codeGen` method would call the `codeGen` method of the left child with argument 0 (which would generate code to evaluate the left expression, leaving the result in register 0), then call the `codeGen` method of the right child with argument 1 (which would generate code to evaluate the right expression, leaving the result in register 1), then perform the operation, leaving the result in the appropriate register.

Unfortunately, this approach does not always work.

Question 1:

Assume that the AST includes only expressions that involve non short-circuited binary operators, with literals or identifiers at the leaves (no unary operators, no dot-access expressions or function calls as operands). Also assume that in the generated code, all operands must be in registers (i.e., neither operand of a binary operator can be in a memory location, nor can it be a literal value).

Show that the approach described above does not always work by giving two examples:

a. An expression that cannot be evaluated using just two registers (without storing intermediate results on the stack), but can be evaluated using three registers.

b. An expression that cannot be evaluated using just three registers (without storing intermediate results on the stack), but can be evaluated using four registers.

For each example, give the expression, the abstract-syntax tree, and a pseudo-code version of the code to evaluate the expression (code that works, *not* the erroneous code that would be generated using the approach described above). For example, here is an expression that *can* be evaluated using just two registers, its AST, and its pseudo-code (***use the same kind of pseudo-code in your answer***):

Expression	AST	pseudo code

`a - b`	`  -`
	`/ \`
	`a  b`
		`load a into T0`
		`load b into T1`
		`T0 = T0 - T1`

Note that an expression's operands need not be evaluated left-to-right. For example, the following pseudo-code would also be OK for the expression `a - b`:

```
load b into T1    // evaluate the right operand first
load a into T0
T0 = T0 - T1
```

Question 2:

Use the same assumptions about the AST and the generated code as for Question 1, including the fact that an expression's operands can be evaluated either left-to-right or right-to-left. Also assume that all operators are non-commutative and non-associative (i.e., the expression `a op b` is not equivalent to the expression `b op a`, and the expression `(a op b) op c` is not equivalent to the expression `a op (b op c)`).

You are to give a recursive algorithm that works as follows:

- The input to the algorithm is an expression node in the AST (recall from Question 1 that we are assuming that we only have expressions involving non short-circuited binary operators).
- The output of the algorithm is the number of registers required to generate code for the *whole* expression.

  Note that generating code to evaluate a leaf node requires one register (into which the value of the identifier or literal is loaded).

  For non-leaf nodes, your algorithm should recursively calculate the number of registers required to generate code for the left operand (leaving the value in a register) and the number of registers required to generate code for the right operand (leaving the value in a register) and then determine the number of registers required for the entire expression represented by the node.

In other words, complete the following method:

```
int numRegisters(ASTNode node) {
    // add code to calculate and return the number of
    // registers required to generate code for the whole
    // expression (whose root is node)
}
```

You may use pseudo-code such as `node.leftChild`, `node.rightChild`, and `isLeaf( )`.

For example, for the expression `a - b`, the input to the algorithm is the root node of the expression tree. The algorithm will recursively determine that the number of registers required to generate code for each operand is 1 (because each operand is a leaf and thus requires one register). The output of the algorithm for this example should be 2.