CMPU 331 · Compilers · Fall 2019

Assignment 4

Due Tuesday, December 3, 2019

This assignment covers intermediate code generation. You may discuss this assignment with other students and work on the problems together. However, your write-up should be your own individual work, and you should indicate in your submission who you worked with, if applicable. Assignments should be submitted through Moodle as a PDF by 11:59pm. (You can export a PDF from Google Docs, Word, LibreOffice, LaTeX, or any other text editor you prefer.)

Problem 1

Consider the following arithmetic expression:

$$(12 + 6) \cdot 5 - (20/(7 + 3)) + (4/2)$$

(a) Sketch out an AST tree for the expression. Keep it simple, similar to the example on Slide 5 of the lecture on "Abstract Syntax Trees and Tree Traversal", but use the operator symbols as the names of the operator AST nodes, and the integers as the names of the integer AST nodes.

(b) Tree traversal is a way of walking through every node of a tree data structure. *Visiting* a node means performing some action on that node, like a semantic analysis action (checking that all symbols have been declared), or a code generation action (writing out some code in the target language).

A *depth-first search* is a tree traversal algorithm that starts at the root of the tree and recursively walks the children in order. For example, this pseudocode `visit` function starts at the top node of the tree, then recursively visits the left child and right child of the top node, in order. Each child recursively visits its own children, and those children recursively visit their children. Finally, after recursively visiting both children, the action of each visit is to print out the name of the node.

```
visit(node) {
    visit(node.left_child)
    visit(node.right_child)
    print(node.name)
}
```

If you apply this simple pseudocode tree traversal to the AST tree in (a), what would it print out? Assume that the "name" of the nodes is simply an operator symbol or integer. Also assume that terminal nodes, which have no children, do nothing for the visit to the left and right child. Write a numbered list showing each line printed by the tree traversal.

Notice that each child is printed before its parent node, this is called *post-order traversal*, because the action for the parent happens after the action for the children.

(c) Update the AST tree in (a), and mark the operator nodes with the order they would be printed using the depth-first search in (b). Which operator is printed first, which second, ...? (Ignore the integer nodes for this step.)

(d) Repeat the tree traversal in (b), but this time, instead of printing the name of the node, generate code for the arithmetic operations. Starting at the top of the tree, visit the left and right children in order, then after visiting the children, write out the instruction for the operator node.

You will only need the following four instructions from LLVM's intermediate representation:

```
%result = add i32 %arg1, %arg2
%result = sub i32 %arg1, %arg2
%result = mul i32 %arg1, %arg2
%result = udiv i32 %arg1, %arg2
```

Recall that the arguments to each instruction, `%arg1` and `%arg2`, can be either registers or constants. If you're writing out the instruction for an operator that has two integer AST nodes as children, it will look like this:

```
%result = add i32 2, 10
```

But, if you're writing out the instruction for an operator that has another operator as a child, you will need to use the result register from the child as an argument to the parent:

```
%child = add i32 2, 10
%parent = mul i32 %child, 5
```

This is *post-order traversal* again. The instruction for each child node has to be written out before the parent node, so that the result of the child instruction can be used as an argument to the parent instruction.

Each `%result` register should have a unique name, but it doesn't matter what that name is. For your own sanity, you might want to pick some consistent naming pattern like `%r1`, `%r2`, `%r3`, etc. and make the numbers in your register names the same as the numbers you annotated on the AST tree in (c).

Ultimately, what you are building in this step is a sequence of instructions that correspond to the same calculation and produce the same result as the arithmetic expression above.

(e) To test the LLVM code you wrote in (d), download the LLVM template file https://www.cs.vassar.edu/~cs331/assignments/assignment4.ll, and insert your code as the first few lines of the `@main` function. In the call to `@printf`, replace the register named `%changeme` with whatever register name you gave to the final `%result` register in your sequence of instructions.

On the CS lab workstations, you can run the test file in the terminal and verify you get the correct result:

```
$ lli-6.0 assignment4.ll
Result: 90
```

(On most other machines, the command will be named simply `lli`, instead of `lli-6.0`.)

You don't need to submit anything for this step, but running your code through this test template is the best way to make sure it actually works correctly. Grading will be done using the same test template.

Optionally, if you find it easier to submit code through GitHub, you can get a copy of the same test template by following the Moodle link to "Assignment 4 on GitHub". Clone that assignment repository, and commit your LLVM code as an update to the file `assignment4.ll`.