CS 241 Week 11 Tutorial Solutions

Code Generation

Spring 2015

Expression code generation

Immediately, we must first recall what these two operators do. Pre-increment will increment the value of ID by 1 and return that *new* value. Post-increment will also increment the value of ID by 1 but will return the *old* value.

Pre-increment is easier so let's do that first.

```
void genCode(tree t)
    ...
    if(t.rule is "factor -> PLUS PLUS lvalue"){
        // generate code to put the lvalue in $3
        // You'll do this in your assignment
        genCode(t.children[2])
        // $3 contains an lvalue, which means we need to fetch the actual value
        // from memory
        // We're making the assumption that the full address and not just an offset
        // Otherwise we need to do something like: add/sub $3, $3, $29
        lw $5, 0($3)
        // Add one to the value
        add $5, $5, $11
        // Save the new value
        sw $5, 0($3)
        // Return new value
        add $3, $5, $0
    }
}
```

Note that we return the actual value and not an lvalue since we have no rules that reduce factor to lvalue (e.g. `factor -> lvalue`). This also means we cannot nest pre and post-increment (e.g. `++(++i)`). Similarly, we cannot return an lvalue as we have no grammar rules to support `expr -> expr + lvalue`. We could add additional rules to facilitate this but that is left as an exercise to you.

Post-increment is "harder" only in the sense that we must keep a copy of the old value and return that instead.

```
void genCode(tree t)
    ...
    if(t.rule is "factor -> lvalue PLUS PLUS"){
        // generate code to put the lvalue in $3
        // You'll do this in your assignment
        genCode(t.children[0])
        // $3 contains an lvalue, which means we need to fetch the actual value
        // from memory
        // We're making the assumption that the full address and not just an offset
```

```
        // Otherwise we need to do something like: add/sub $3, $3, $29
        lw $5, 0($3)
        // Copy the old value
        add $6, $5, $0
        // Add one to the current value
        add $5, $5, $11
        // Save the new value
        sw $5, 0($3)
        // Return old value
        add $3, $6, $0
    }
}
```

As we can see, post-increment is not that much more expensive than pre-increment for integers. It's one instruction more but this gets compounded as we create more complex data structures.

Switch statement

Here the individual case statements need to be able to jump to a label that the switch statement generated, so we need a way to pass the label IDs down the tree. We will do this by creating a parentLabelId field which the parent passed down the tree to its children.

```
void genCode(tree t)
    ...
    if(t.rule is "statement -> SWITCH LPAREN expr RPAREN LBRACE cases default RBRACE"){
        x = genLabelID()
        // Evaluate the expr and push it onto the stack
        genCode(t.children[2])
        push($3)
        // Pass the label ID we generated to the children
        c.children[5].parentLabelId = x
        // Generate all the case statements
        genCode(t.children[5])
        // Generate code for the default case
        genCode(t.children[6])
        endSwitch + x:
    }
    if(t.rule is "cases -> cases case"){
        // Generate the code for the case statements
        // Pass on the parentLabelId
        t.children[0].parentLabelId = t.parentLabelId
        genCode(t.children[0])
        t.children[1].parentLabelId = t.parentLabelId
        genCode(t.children[1])
    }
    if(t.rule is "cases -> "){
    }
    if(t.rule is "case -> CASE LPAREN expr RPAREN LBRACE statements RBRACE"){
        x = genLabelID()
        // Evaluate the expr
        genCode(t.children[2])
        // Pop the switch statement expression from the stack
        pop($5)
        // Compare the case statement and switch statement expressions
        bne $3, $5, endLabel + x
```

```
        genCode(t.children[5])
        jr endSwitch + t.parentLabelId
        endLabel + x:
    }
    if(t.rule is "default -> DEFAULT LBRACE statements RBRACE") {
        genCode(t.children[2])
    }
}
```

There are a number of changes that would need to be made in order to bring this closer to the C switch statement.

1. The grammar must be updated to remove braces around the statements and to make the default case optional.
2. The keyword `break` should be added, along with the grammar rule `statement -> BREAK SEMI` .
3. The code generation procedure must be reworked. All of the conditional branch code should be generated up front, with a branch to the end of the switch only if there is no default case and none of the case conditions are true, or when there is a break statement. The simplest way to accomplish this is with two passes over a switch statement: one to generate all of the condition-checking code, and one to generate the case bodies.

Test code generation

Our immediate instinct to generate code for the less than or equal to test is:

- Generate code for left expression and push result onto stack
- Generate code for right expression and pop previous result from stack into $5.
  - $3 contains right-value, $5 contains left-value
- Check if $5 is less than $3
- If it is, return 1
- Otherwise, check if they are equal
- If they are equal, return 1
- Otherwise, return 0

What is the problem with this code? Well, it's slightly ineffecient. We use more instructions than necessary. Let's see:

```
slt $6, $5, $3
bne $0, $6, returnOne
beq $5, $3, returnOne
add $3, $0, $0
beq $0, $0, end
returnOne:
add $3, $11, $0
```

Is there a better way? Yes. What does X <= Y mean? It evalues to **true** if X is not greater than Y. Equvailently, it returns **true** if Y is not *less than* X, i.e. $!(Y < X)$. But that's effectively swapping the arguments in our slt expression.

```
slt $6, $3, $5
sub $3, $11, $6 ; if $6 = 1 then $3 = 0 => y < x
```

So we've managed to signficantly reduce our generated code from our initial version by the use of some fancy negation techniques.