CS 241 Week 11 Tutorial

Code Generation

Spring 2015

Summary

- Code generation tips
- Expression code generation
- Switch statement
- Test code generation

Code Generation Conventions and Tips

These are not mandatory, but following them might make it easier to write your compiler (most of which were discussed in class).

- All expressions should store their result in $3. Being consistent like this will make it easier to build up large expressions from the results of smaller ones.
- When evaluating expressions, store any intermediate results you need on the stack rather than in registers. This will guarantee that you don't run out of registers and accidentally overwrite important values.
- Define some helper functions that return bits of code that you have to output very frequently. For example, you may want helper functions (in your implementation language - i.e. Racket/C++) called "push(register)" and "pop(register)" to deal with using the stack.
- Store the value 4 in $4, and 1 in $11. You will use these constants a lot in the generated assembly code, so it is nice to always have them available. You may also want to store the addresses of the provided assembly procedures (print, new, etc.) as constants as well.
- Output comments along with your assembly code. It is difficult to output comments that are really meaningful, but as long as you can get a general idea of where each bit of code came from it will help with debugging.

Expression code generation

Recall that in C and C++ we have pre and post-increment operators (e.g. `++i` and `i++`). Suppose we add the following rules to the WLP4 grammar

```
factor -> PLUS PLUS lvalue
factor -> lvalue PLUS PLUS
```

For our purposes, we will assume that scanning, parsing, and semantic analysis all work out. Furthermore, as in A9 we will only consider the cases where lvalue derives an INT.

Write pseudocode that generates the correct MIPS output for each of these grammar rules.

Switch statement

C and C++ also have the switch statement. We will add a similar statement with slightly different syntax:

```
switch(expr) {
    case(expr) {
        statements
    }
    case(expr) {
        statements
    }
    ...
    default {
        statements
    }
}
```

Here, our case statements don't "fall through", so we don't need a break statement at the end of each case block. Also, the default case is not optional and must appear after all the other cases.

Again, suppose we add the following rules to the grammar and assume that scanning, parsing, and semantic analysis and done, write the pseudocode that generates the correct MIPS for the following grammar rules:

```
statement -> SWITCH LPAREN expr RPAREN LBRACE cases default RBRACE
cases -> cases case
cases ->
case -> CASE LPAREN expr RPAREN LBRACE statements RBRACE
default -> DEFAULT LBRACE statements RBRACE
```

What changes could we make to bring this more in line with the switch statement in C?

Test code generation

Generate code for the production rule:

```
test -> expr LE expr
```