Assignment 1
Instruction Selection and Register Allocation

15-411: Compiler Design
Alex Crichton (acrichto@andrew) and Ian Gillis (igillis@andrew)

Due: Tuesday, September 11, 2012 (1:30 pm)

Reminder: Assignments are individual assignments, not done in pairs. The work must be all your own.

You may hand in a handwritten solution or a printout of a typeset solution at the beginning of lecture on Tuesday, September 11. Please read the late policy for written assignments on the course web page. If you decide not to typeset your answers, make sure the text and pictures are legible and clear.

Problem 1 (30 points)

(a) Consecutive statements in a program can be represented in an AST by a `Seq` node that has two statements (possibly other `Seq`s) as children. For example, the program

```
int x = 5 + 3;
return x;
```

would be represented in an AST as

```
Define(Var x, Seq(Assign(Var x, Plus(Const 5, Const 3)), Return (Var x)))
```

Also notice that the variable `x` is defined for only a portion of the AST. This is achieved via a `Define` node which constitues a variable and a subtree which the variable is defined for.

Using this type of AST, write down (either as in the example or by drawing a real tree) the AST for the following program

```
int x = 9 * (5 - 2);
int y = (x + 2) / 4;
return x + y;
```

(b) When we expand the capabilities of a programming language, we also need to extend the AST to represent the new features. Write down the AST for the following program, choosing a reasonable AST representation of the "while", and "!=". Assume that the variables `x` and `y` are defined elsewhere, but notice that the variable `z` is only defined within the "while" loop.

```
while (x != 5) {
  int z = x * x;
  y += z;
  x = x + 1;
}
return y;
```

(c) Now you will perform instruction selection on the AST you created in part (a) into three-operand assembly language by using the patterns in the table below. As a sample, the example AST from part (a) would be translated (in a simplistic fashion) to

```
t0 <- 5
t1 <- 3
t2 <- t0 + t1
x  <- t2
t3 <- x
return t3
```

We aren't performing register allocation yet (that's for problem 2), so we will continue to refer to variables by their names and generate new temp variables ($t_0, ..., t_n$) as necessary. S_1 and S_2 refer to the first and second subtrees of an AST node. $S_n$instrs refers to the instructions generated for S_n, and $S_n$temp refers to a new temp variable created to hold the result of S_n in cases where S_n has one. Lastly, r is the temp where the result of an expression should be placed.

Pattern	Assembly
`Const` n	r `<-` n
`var` x	r `<-` x
`Plus(`S_1, S_2`)`	S_1`instrs,` S_2`instrs,` r `<-` S_1`temp + `S_2`temp`
`Minus(`S_1, S_2`)`	S_1`instrs,` S_2`instrs,` r `<-` S_1`temp - `S_2`temp`
`Times(`S_1, S_2`)`	S_1`instrs,` S_2`instrs,` r `<-` S_1`temp * `S_2`temp`
`Divide(`S_1, S_2`)`	S_1`instrs,` S_2`instrs,` r `<-` S_1`temp / `S_2`temp`
`Assign(Var` x`,` S_2`)`	S_2`instrs,` x `<-` S_2`result`
`Return` S_1	S_1`instrs,` `return` S_1`result`
`Seq(`S_1, S_2`)`	S_1`instrs,` S_2`instrs`

(d) Now perform instruction selection on the AST you created in part (b). To accomplish this, we will need to introduce "cmpne r x y", "label l", "jmp l", and "jmpnzero l x" into our assembly language, where l is always a number that identifies a label. "cmpne r x y" assigns 1 to r if the value of x is not equal to the value of y y and 0 to r otherwise. In the case of jmpnzero, control flow jumps to label l if the value of x is not 0. You will need to come up with your own patterns for generating instructions for "while" and "!=" , and you must write down these patterns in addition to the specific program.

Pattern	Assembly

Problem 2 (30 points)

In this question you will perform the register allocation algorithm discussed in class on a small assembly program which computes $\log_2(6x-2)+1$ (in the code given, the input x is hardcoded to be 7). The registers to be used are r_1,...,r_n so for the purposes of this question you have as many registers as you need (though the algorithm will still be trying to use as few as possible).

The language used is the assembly from problem 1 with an additional shfit instruction, used as in "$t_i \leftarrow t_j >> t_k$". As in x86 assembly, some instructions have special conditions associated with them. For the shift instruction in our language, t_k must be assigned to register r_0. Additionally, for "return t_i", t_i must always be assigned to register r_0.

```
t0 <- 7 // "input"
t1 <- 6
t2 <- t0 * t1
t3 <- 2
t4 <- t2 - t3
t5 <- 1
t6 <- 0
t7 <- 1
label 1
t4 <- t4 >> t5
t6 <- t6 + t7
jmpnzero 1 t4
return t6
```

(a) Compute the live variables at each instruction in the above program.

(b) Construct the interference graph for the program. If you don't want to actually draw a graph, you can just list the variables that each variable interferes with. You should also state whether the graph is chordal.

(c) What problem does the current program have for allocating registers? Give a modified version of the program that does not have this problem (try to make the smallest change that allows register allocation).

(d) Use the chordal graph coloring algorithm discussed in class to allocate registers for all the temps in the modified program. If you did part (c) correctly then your liveness analysis and interference graph should still be usable with slight modification.