Assignment 1
Instruction Selection and Register Allocation

15-411: Compiler Design

Due Thursday, September 17, 2015 (11:00pm)

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

Handin of your solutions is as a PDF file on Autolab. If this presents a significant hardship for you, please contact the course staff. Please read the late policy for written assignments on the course web page.

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

could be represented in an AST as

```
declare(var("x"), seq(assign(var("x"),
                             plus(const(5), const(3))),
                      return(var("x"))))
```

The variable `x` is declared for only a portion of the AST. This is achieved via a `declare` node, the first subtree of which is a variable, and the second a subtree which the variable is declared for (called the *scope* of the variable).

Using this type of AST, write down (either as in the example or by drawing a real tree) the AST for the following program. Variables initialized as part of a declaration should become a simple declaration followed by an assignment.

```
int x = (-9) + (5 * 13);
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 for `while` and `!=` (not equal). Assume that the variables `x` and `y` are declared elsewhere, but notice that the variable `z` is only declared 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
x <- t0 + t1
t3 <- x
ret 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 as necessary. The code generation for expressions is just as it was in lecture, and includes no optimizations:

e	$\mathsf{cogen}(d, e)$	proviso
$\mathsf{const}(c)$	$d \leftarrow c$	
$\mathsf{var}(x)$	$d \leftarrow x$	
$\mathsf{plus}(e_1, e_2)$	$\mathsf{cogen}(t_1, e_1), \mathsf{cogen}(t_2, e_2), d \leftarrow t_1 + t_2$	(t_1, t_2 new)
$\mathsf{times}(e_1, e_2)$	$\mathsf{cogen}(t_1, e_1), \mathsf{cogen}(t_2, e_2), d \leftarrow t_1 * t_2$	(t_1, t_2 new)
...	...	...

and similarly for other expressions. For statements:

s	$\mathsf{cogen}(s)$	proviso
$\mathsf{assign}(x, e)$	$\mathsf{cogen}(x, e)$	
$\mathsf{return}(e)$	$\mathsf{cogen}(t, e), \mathsf{ret}\, t$	(t new)
$\mathsf{seq}(s_1, s_2)$	$\mathsf{cogen}(s_1), \mathsf{cogen}(s_2)$	

(d) Now perform instruction selection on the AST you created in part (b). To accomplish this, we introduce new machine instructions.

- xor $d\ s_1\ s_2$ assigns the bitwise-xor of s_1 and s_2 to d.
  Usefully, this means that d will be zero exactly when s_1 and s_2 are equal.
- label l creates a jump target at the current place in the instruction sequence.
- goto l continues execution at label l.
- branch $s\ l_1\ l_2$ jumps to label l_1 if the value of s is nonzero,
  and otherwise, if s is zero, it jumps to label l_2.

Write down general patterns to generate code for `while` statements and `!=` expressions, using these new target instructions as you see fit. Then apply your general code generation patterns to the specific program in part (b).

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 42).

```
t0 <- 42 // "input"
t1 <- 6
t2 <- t0 * t1
t3 <- 2
t4 <- t2 - t3
t5 <- 1
t6 <- 0
t7 <- 1

label .loop
t4 <- t4 >> t5
t6 <- t6 + t7
branch t4 .loop .exit

label .exit
ret t6
```

The target of your compilation will be a three-address machine with as many registers, named $r_0, \ldots, r_n$, as you need (though the algorithm will still be trying to use as few as possible). The language also has a right shift instruction $d \leftarrow s_1$ >> s_2.

(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) Use the maximum cardinality search algorithm we described in lecture, starting from `t7`, to construct an elimination ordering.

(d) Using this ordering from part (c), use the greedy graph coloring described in class to assign registers $r_0, \ldots, r_n$ to temps.

Now we will add a restriction to our three-address assembly language: the register r_0 *must* be used as the return register (in other words, the operand of `ret` must be `r0`). Similarly, in the shift instruction $d \leftarrow s_1$ `>>` s_2, the same register r_0 *must* be to hold s_2, the magnitude of the shift.

(e) Why does this represent a problem for our sample program? Give a slightly modified but equivalent version of the program that does not have this problem.

(f) Redo the graph coloring algorithm on this modified program. This time, explain how to perform maximum cardinality search in such a way that t_7 is assigned to a register with the highest possible number.