Assignment 1
Instruction Selection and Register Allocation

15-411: Compiler Design
Frank Pfenning
Flávio Cruz, Maxime Serrano, Rokhini Prabhu, Tae Gyun Kim

Due Thursday, September 11, 2014 (23:59pm)

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
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 as necessary. The code generation for expressions is just as in Lecture 2, and thus 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
$\mathsf{assign}(x, e)$	$\mathsf{cogen}(x, e)$	
$\mathsf{return}(e)$	$\mathsf{cogen}(t, e), \mathsf{return}\ t$	(t new)
$\mathsf{seq}(s_1, s_2)$	$\mathsf{cogen}(s_1), \mathsf{cogen}(s_2)$	

This is a slight departure from lecture, where we posed a special return register r_{ret}. Here, the return instruction has the form $\mathsf{return}\ s$ for an operand s.

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

- cmpne $d\ s_1\ s_2$ assigns 1 to d if the value of s_1 is not equal to the value of s_2 and 0 otherwise.
- label l creates a jump target at the current place in the instruction sequence.
- jmp l continues execution at label l.
- jmpnz $l\ s$ jumps to label l if the value of s is not 0, otherwise continues with the next instruction.

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 7). The registers to be used are $r_0, \ldots, r_n$ 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 right shift instruction:

$$d \leftarrow s_1 \texttt{ >> } s_2$$

The language has two special conditions associated with instructions, of the kind that also arises for x86.

- In the shift instruction above, operand s_2 must be assigned to register r_0.
- In the return instruction return s, operand s must be also 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
jmpnz 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, if any? If necessary, give a modified version of the program that does not have this problem.

(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 modifications.