Assignment 1: Instruction Selection and Register Allocation

15-411: Compiler Design
Jan Hoffmann, Jonathan Burns, DeeDee Han, Anatol Liu, Alice Rao

Due Thursday, September 14, 2017 (11:59pm)

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

You should submit your assignment as a PDF on Gradescope. *The code for the Fall 2017 version of 15-411/611 is M2RKRD*. If this code does not work for you, or if you have any other trouble enrolling on Gradescope, please contact the course staff. Please read the late policy for written assignments on the course web page.

Problem 1 (20 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, and the available constructive nodes are `declare`, `seq`, `plus`, `const(n)`, `assign`, `mult`, `negate`, `mod`, and `return`. Be sure to parse the code according to the language's specified precedence rules (and evaluate according to mathematical order of operations).

```
int x;
x = 1 + 2 * 3 + (-9);
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 a potential 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 the following program. Note that we, other than in class, assume that the return instruction takes an operand to be returned as an argument.

```
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)
$\mathsf{seq}(s_1, s_2)$	$\mathsf{cogen}(s_1), \mathsf{cogen}(s_2)$	

Problem 2 (25 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 as you need (though the algorithm will still be trying to use as few as possible). The registers are named $r_0, \ldots, r_n$. 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 a simplicial elimination ordering. Then, using this ordering, 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.

(d) 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.

(e) Do the graph coloring algorithm like in 2(c) on this modified program. This time, allocate your registers in a way such that t_7 is assigned to the register with the highest possible number, and explain how you did this.

Problem 3 (10 points)

Recall that a *chordal* graph is a graph where every cycle of length 4 or larger contains a chord (an edge that connects to vertices on the cycle but is not part of the cycle).

(a) Write a program in three-address assembly that has a non-chordal interference graph and uses not more than 4 temps. Draw the interference graph.

(b) Rename the temps in your program so that you get an equivalent program that assigns every temp at most once (the resulting program is in SSA form). Draw the interference graph of the modified program. Is the graph chordal?

Problem 4 (5 points)

A relevant quote from Stephen Dolan:

> It is well-known that the x86 instruction set is baroque, overcomplicated, and redundantly redundant. We show just how much fluff it has by demonstrating that it remains Turing-complete when reduced to just one instruction.

Did you know that all of instruction selection is bogus? We're going to (partially) show that we don't need any instructions other than `mov` to implement an L1 compiler.

(a) Assuming x and y are registers containing two possibly equal values, and we'd like R to contain a 1 if they're equal and 0 if not. Write three lines of assembly that checks for equality using only `mov` instructions.

(b) So why aren't we just doing this instead? In 1 sentence, explain why it might not make sense to compile all code to only `mov` instructions.

Problem 5 (Book Prize)

This is a really hard problem. Feel free to think about this and email the instructor a proof if you think you've got a good solution. This problem will probably *not* help you with your compiler implementation or future written assignments. So it's absolutely okay to ignore it. However, the 411/611 student with the first correct solution will receive the instructor's favorite graph theory book as a prize.

Problem

Reproduce the following definitions from lecture: perfect elimination ordering; maximum cardinality search; chordal graph. Prove the following theorem.

> Performing a maximum cardinality search on a graph G produces a perfect elimination ordering for G if and only if G is chordal.

Rules

- The first correct solution emailed to the instructor wins
- There is no time limit
- The work must be your own
- You must prove all theorems (from lecture or elsewhere) that you use
- A solution is considered correct if you can convince two or more participants in the instructor's group meeting by giving a talk