Assignment 1: Backend

15-411/611: Course Staff

Due Tuesday, January 30, 2024 (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. If you have any trouble enrolling on Gradescope, please contact the course staff ASAP. Please read the late policy for written assignments on the course web page.

Problem 1: Code Generation (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("x", seq(assign("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 (see Figure 1), write down the AST for the following L1 program. Declarations that initialize the variable should be elaborated into a simple declaration followed by an assignment.

Be sure to parse the code according to L1's specified precedence rules, which are the usual mathematical order of operations.

```
int x;
x = 1 + 2 * 3 + (-9);
int y = (x + 2) / 4;
return x % y;
```

```
datatype stmt =
    declare of string * stmt
  | seq of stmt * stmt
  | assign of string * expr
  | return of expr
```

```
datatype expr =
    var of string
  | const of int
  | negate of expr
  | plus of expr * expr
  | mult of expr * expr
  | mod of expr * expr
  | div of expr * expr
```

Figure 1: AST datatypes

(b) When you extend from L1 to L2, you will need to extend the AST type to represent the new features. Write down a potential AST for the following program, extending the type in Figure 1 with 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. Use top-down code generation just as described in lecture, with no optimizations (see Figure 2).

s	$\texttt{cogen}(s)$
$\texttt{declare}(x, s)$	$\texttt{cogen}(s)$
$\texttt{assign}(x, e)$	$\texttt{cogen}(x, e)$
$\texttt{return}(e)$	$\texttt{cogen}(t, e)$ $\texttt{ret}\, t$
$\texttt{seq}(s_1, s_2)$	$\texttt{cogen}(s_1)$ $\texttt{cogen}(s_2)$

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

Figure 2: Top-down code generation rules for statements (l) and expressions (r). Temporaries (t, t_1, t_2) in each rule are assumed to be fresh.

Problem 2: Register Allocation (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).

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 $r_0, \ldots, r_n$.

```
L00: t0 <- 42 // "input"
L01: t1 <- 6
L02: t2 <- t0 * t1
L03: t3 <- 2
L04: t4 <- t2 - t3
L05: t5 <- 1
L06: t6 <- 0
L07: t7 <- 1

label .loop
L08: t4 <- t4 >> t5
L09: t6 <- t6 + t7
L10: branch t4 .loop .exit

label .exit
L11: ret t6
```

(a) Compute the live-out sets for each line in the above program. The definition of live-out is as follows:

- succ (l) = set of lines that could immediately follow l
- Uses (l) = set of variables whose value is used in line l
- Defs (l) = set of variables defined in line l
- LiveIn (l) = (LiveOut (l) − Defs (l)) ∪ Uses (l)
- LiveOut $(l) = \bigcup_{s \in \text{succ}(l)}$ LiveIn (s)

(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.

Recall that two variables x, y interfere if at some line l, $x \in$ Defs (l) and $y \in$ LiveOut (l).

(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. [1]

(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 1(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.

[1] These restrictions are *definitely not* foreshadowing the restrictions you may encounter in x86_64 assembly...

Problem 3: Chordal Graphs & SSA (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 (0 points, optional)

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 x86 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.