

Recitation 1: Back End

11 September

Lab 1

Designing a compiler for each of the labs is the heart of this course. The handouts for Lab 1 checkpoint and Lab 1 compiler are both linked on the course website at <https://www.cs.cmu.edu/~411/>, under assignments. Lab submissions are via Notolab at <https://notolab.cs.cmu.edu/>. In very short summary,

- Lab 1 checkpoint asks you to implement register allocation based on graph coloring algorithm given in class.
- Lab 1 asks you to implement the compiler for the language L1.

Read the handouts carefully and good luck have fun!

Tracing Through the Backend

In this recitation, we're going to discuss an example of the processing done by the compiler backend. Since you won't have to touch the frontend for Lab 1 (unless you opt to not use any starter code), we'll leave it for a future week. Here's the code and AST we'll use for the example:

```

1 int main() {
2   int x = 42;
3   int z;
4   if (x % 2 == 0) {
5     x++;
6     z = 1;
7   } else {
8     z = -1;
9   }
10  int y = 1;
11  while (y <= x - 1) {
12    y = x + y;
13  }
14  return z * y;
15 }

```

```

1 declare(x, seq(
2   assign(x, const(42)),
3   declare(z, seq(
4     if(compare(mod(x, const(2)), const
5       (0), EQ), seq(
6         incr(x),
7         assign(z, const(1))
8       ),
9     assign(z, neg(const(1)))
10    ),
11  declare(y, seq(
12    assign(y, const(1)), seq(
13      while(
14        compare(y, minus(x, 1),
15          LEQ),
16        assign(y, add(x, y))
17      ),
18      return(times(z,y))
19    )
20  ))

```

Code Generation

As discussed in lecture yesterday, we use the "maximal munch" algorithm to generate abstract 3-address assembly from IR. For each line in the IR, we recursively pattern-match as deep as possible into each sub-expression and generate lines of assembly at each step.

Recall that one formulation of the rules we used in lecture was the following:

e	\check{e}	\hat{e}	proviso
c	\cdot	c	
x	\cdot	x	
$e_1 \oplus e_2$	$\check{e}_1, \check{e}_2, t \leftarrow \hat{e}_1 \oplus \hat{e}_2$	t	(t fresh)

and for statements

s	\check{s}
$x = e$	$\check{e}, x \leftarrow \hat{e}$
return e	$\check{e}, r_{ret} \leftarrow \hat{e}, \text{ret}$

Checkpoint 0

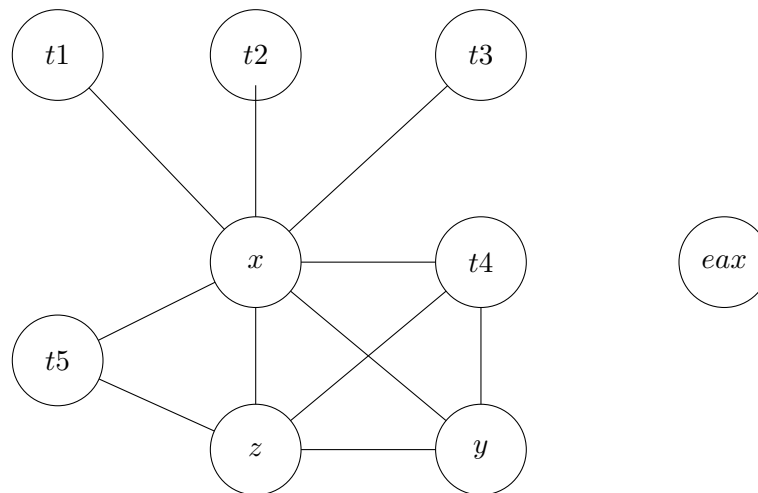
Translate the above AST into abstract 3-address assembly. You do not need to turn it into SSA form here. NOTE: answers may differ depending on the exact rules you're using in your head to generate instructions.

Checkpoint 1

Draw the control flow graph (CFG) based on the 3-address assembly. Think about where you need to put Φ functions if you turn it into SSA form.

Maximum Cardinality Search

From the last recitation, you already know how to perform liveness analysis and construct interference graph. For this program, the graph is given here (as exercise, you could construct the graph yourself):



In order to color the interference graph using the greedy algorithm, we need to decide on an order in which to process the vertices. We do this using the Maximum Cardinality Search algorithm. We first assign a weight of 0 to each vertex. Then, at each step, we:

- Choose a vertex with maximal weight from the working set
- Add it to our ordering and remove it from the working set
- Increment the weights of all of its neighbors

This algorithm produces an ordering which is optimal for chordal graphs.

Checkpoint 2

Use Maximum Cardinality Search to generate an ordering of the vertices in the example above. Break ties by choosing the vertex that is lexicographically first.

Greedy Graph Coloring

Once we have an ordering, we can assign registers to each of the temps in our program. Ignoring pre-colored vertices, such as `%eax`, we can color the temps by assigning the lowest register that is not assigned to any of the vertex's neighbors.

Checkpoint 3

Perform Greedy Graph Coloring on the interference graph from above to assign registers `%r1`, `%r2`, ... to the temps in the program. Then rewrite the abstract assembly using the new registers.

Lab 1 Tip: Spilling Temps

We can't fit all of our data in registers, so we spill into memory. But we need at least one operand in a register for most arithmetic operations. This is getting into the software engineering part of the course, but we will outline one strategy that you can use.

You will need to reserve a register, typically `%r11d`. Perform register allocation, then scan through your instructions looking for memory-memory operations. You then insert a `mov` from the destination to `%r11d`, perform the operation, then move `%r11d` back to memory.

In a functional language, you can implement this in a pass similar to code generation, where you case on instruction type and produce either a list with the input instruction, or a list with the moves into and out of `%r11d`.