15-411: Compiler Design — Spring 2026

Recitation 1: Instruction Selection Solutions — 23 Jan

[TA only: Remember to Announce]

- L1 Tests due Saturday 1/24. Submissions for coding assignments are through Gradescope.
- L1 Checkpoint due Saturday 1/24. L1 Compiler due Wednesday 1/28.
- Written 1 due Wednesday 1/28. Submissions are through Gradescope.
- Give a brief description of stages of the compiler required for L1 checkpoint and compiler.

Instruction Selection

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

Intermediate Representation

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

Our translation target is formulated as follow:

Source Operands	s	::=	$t \mid r \mid c$
Destination Operands	d	::=	$t \mid r$
Instructions	i	::=	$d \leftarrow s$
		$\mid$	$d \leftarrow s_1 \oplus s_2$
		$\mid$	if s then l_t else l_f
		$\mid$	goto l
		$\mid$	l :
		$\mid$	ret
Binop	$\oplus$	::=	$+ \mid - \mid * \mid /$
Condition Code	$?$	::=	$= \mid \neq \mid \ \ldots$
Programs	p	::=	$i_1; \ldots; i_n$

Recall that one formulation of maximal munch for expressions is the following:

$$\text{codegen}(e) = \langle \check{e}, \hat{e} \rangle$$

$\check{e}$: sequence of instructions generated from e
$\hat{e}$: destination operand storing the value of e

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 (exercise: fill in translation rules for `if` and `while`):

$$\text{codegen}(s) = \check{s}$$

s	$\check{s}$
$x = e$	$\check{e}, x \leftarrow \hat{e}$
return e	$\check{e}$, %eax $\leftarrow \hat{e}$, ret

Checkpoint 0

Fill in translation rules for `if` and `while` AST nodes.

Solution:

$$\begin{array}{rcll} \text{translate}(\text{if}(e,\ s_1,\ s_2)) & = & & \check{e}; \\ & & & \text{if } (\hat{e}) \text{ then } l_1 \text{ else } l_2\,; \\ & & l_1 \ : & \check{s}_1\,; \text{ goto } l_3\,; \\ & & l_2 \ : & \check{s}_2\,; \text{ goto } l_3\,; \\ & & l_3 \ : & \qquad\qquad (l_1, l_2, l_3 \text{ fresh}) \end{array}$$

$$\begin{array}{rcll} \text{translate}(\text{while}(e, s)) & = & & \text{goto } l_1 \\ & & l_1 \ : & \check{e} \\ & & & \text{if } (\hat{e}) \text{ then } l_2 \text{ else } l_3\,; \\ & & l_2 \ : & \check{s}\,; \text{ goto } l_1; \\ & & l_3 \ : & \qquad\qquad (l_1, l_2, l_3 \text{ fresh}) \end{array}$$

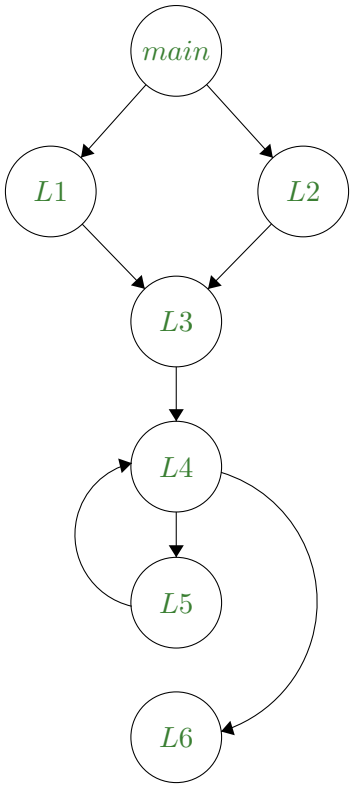
Checkpoint 1

Applying the translation rules, we derive the following abstract 3-address assembly:

```
1   main:
2     x <- 42
3     t1 <- 2
4     t2 <- x % t1
5     if (t2 == 0) then L1 else L2
6   L1:
7     x <- x + 1
8     z <- 1
9     goto L3
10  L2:
11    t3 <- 1
12    z <- t3 * -1
13    goto L3
14  L3:
15    y <- 1
16    goto L4
17  L4:
18    t4 <- x - 1
19    if (y <= t4) then L5 else L6
20  L5:
21    t5 <- x + y
22    y <- t5
23    goto L4
24  L6:
25    %eax <- z * y
26    return
```

Draw the control flow graph (CFG) based on the 3-address assembly and consider what modifications are necessary to convert the program into SSA form. Think about where you need to put Φ functions if you turn it into SSA form.

Solution:

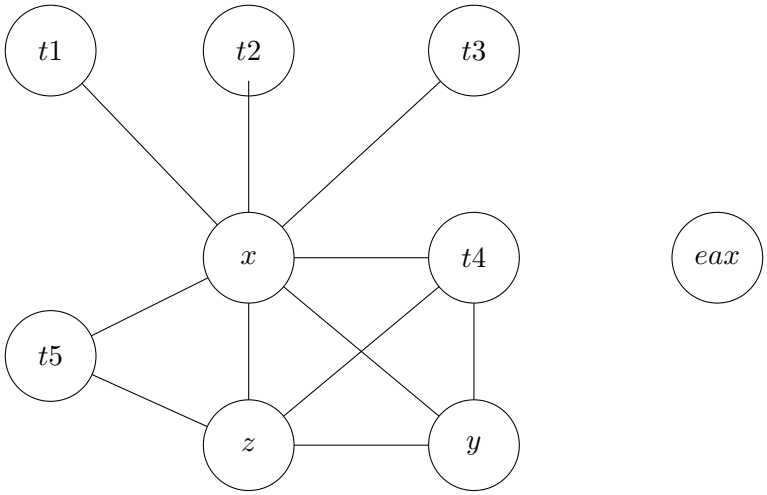



```
main:
  x0 <- 42
  t1 <- 2
  t2 <- x0 % t1
  if (t2 == 0) then L1 else L2
L1:
  x1 <- x0 + 1
  z0 <- 1
  goto L3
L2:
  t3 <- 1
  z1 <- t3 * -1
  goto L3
L3:
  x2 <- phi(x1, x0)
  z2 <- phi(z0, z1)
  y0 <- 1
  goto L4
L4:
  y1 <- phi(y0, y2)
  t4 <- x2 - 1
  if (y1 <= t4) then L5 else L6
L5:
  t5 <- x2 + y1
  y2 <- t5
  goto L4
```

```
27 L6:
28  %eax <- z2 * y1
29  return
```

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:

(a) Choose a vertex with maximal weight from the working set

(b) Add it to our ordering and remove it from the working set

(c) 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.

Solution: `%eax, t1, x, t2, t3, t4, y, z, t5`

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.

Solution:

Mapping:

```
t1 => %r1
x  => %r2
t2 => %r1
t3 => %r1
t4 => %r1
y => %r3
z => %r4
t5 => %r1
```

Program:

```
1  main:
2    %r2 <- 42
3    %r1 <- 2
4    %r1 <- %r2 % %r1
5    if (%r1 == 0) then L1 else L2
6  L1:
7    %r2 <- %r2 + 1
8    %r4 <- 1
9    goto L3
10 L2:
11   %r1 <- 1
12   %r4 <- %r1 * -1
13   goto L3
14 L3:
15   %r3 <- 1
16   goto L4
17 L4:
18   %r1 <- %r2 - 1
19   if (%r3 <= %r1) then L5 else L6
20 L5:
21   %r1 <- %r2 + %r3
22   %r3 <- %r1
23   goto L4
24 L6:
25   %eax <- %r4 * %r3
26   return
```

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`.

(Bonus) Best-Effort Coalescing

Unlike iterative register allocation, SSA-based register allocation perform coalescing after coloring. For a copy instruction where **x and y do not interfere**

$$\texttt{x} \leftarrow \texttt{y}$$

we can eliminate it by reassigning x and y to the same register. Let K be the number of machine registers available, S_x and S_y be the set of colors used in $\texttt{Nbr}(\texttt{x})$ and $\texttt{Nbr}(\texttt{y})$. Best-effort coalescing decides to merge x and y as node xy and choose color c as xy's color if $\texttt{c} < \texttt{K}$ and $\texttt{c} \notin S_x \cup S_y$. If this can be done, we replace all occurrence of x and y in the program with xy.

Checkpoint 4

Perform best-effort coalescing on the colored abstract 3-address assembly from previous checkpoint.

Solution: There is one move `y2 <- t5`. Since they don't interfere, they can be coalesced. `y2` got assigned register `%r3` and `t5` register `%r1`. Their joint neighborhood (`x`, `z`, `t4`) has consumed registers `%r1`, `%r2`, and `%r4`. We can still choose `%r3` for the coalesced `y2t5`

The new 3-address assembly looks like:

```
main:
  x0 <- 42
  t1 <- 2
  t2 <- x0 % t1
  if (t2 == 0) then L1 else L2
L1:
  x1 <- x0 + 1
  z0 <- 1
  goto L3
L2:
  t3 <- 1
  z1 <- t3 * -1
  goto L3
L3:
  x2 <- phi(x1, x0)
  z2 <- phi(z0, z1)
  y0 <- 1
  goto L4
L4:
  y1 <- phi(y0, y2t5)
  t4 <- x2 - 1
  if (y1 <= t4) then L5 else L6
L5:
  y2t5 <- x2 + y1
  // (removed) y2 <- t5
  goto L4
L6:
  %eax <- z2 * y1
  return
```

```
main:
  %r2 <- 42
  %r1 <- 2
  %r1 <- %r2 % %r1
  if (%r1 == 0) then L1 else L2
L1:
  %r2 <- %r2 + 1
  %r4 <- 1
  goto L3
L2:
  %r1 <- 1
  %r4 <- %r1 * -1
  goto L3
L3:
  %r3 <- 1
  goto L4
L4:
  %r1 <- %r2 - 1
  if (%r3 <= %r1) then L5 else L6
L5:
  %r3 <- %r2 + %r3
    // dest changed: %r1 -> %r3
  // (removed) %r3 <- %r1
  goto L4
L6:
  %eax <- %r4 * %r3
  return
```