15-411: Compiler Design — Spring 2026

Recitation 2: SSA — 30 January

Constructing the Dominator Tree

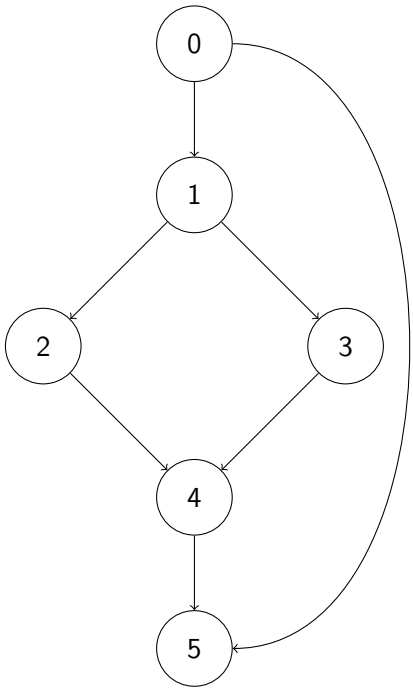
Before entering SSA, we need a dominator tree to effectively determine where Φ-functions should be inserted. Here are some important definitions:

- Node x **dominates** node y if every possible execution path from the CFG entry point to y includes x.
- Node x **strictly dominates** node y if x dominates y and $x \neq y$.
- The **immediate dominator** of a node x is the unique node that strictly dominates x but doesn't strictly dominate any other node that strictly dominates x.
- The **dominance frontier** of a node x is the set of nodes S such that for each node $y \in S$, x dominates a predecessor of y but x doesn't strictly dominate y. A more intuitive definition is the border of the CFG region that is dominated by x.

We construct the dominator tree by drawing an directed edge from node x to node y if x immediately dominates y.

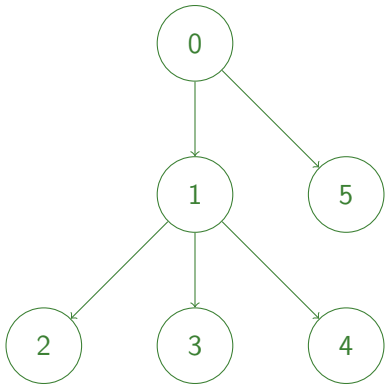
Checkpoint 0

Given the following CFG, draw its corresponding dominator tree. Then, list out the dominance frontiers for each node.



Solution:

Dominator tree:



dominance frontiers:

DF(0): {}
DF(1): {5}
DF(2): {4}
DF(3): {4}
DF(4): {5}
DF(5): {}

SSA Construction

To systematically construct SSA, we'll use the **Dominance-Frontier Criterion**: Whenever node x contains a definition of some variable a, then any node $z \in DF(x)$, z needs some Φ-function for a.

Applying this criterion, we insert Φ-functions where necessary:

- Gather all the defsites (basic block where variable has been defined) of every variable
- For each variable:
  - For each defsite:
    - For each node in DF(defsite):
      - If we don't have $\Phi()$ in node, put one in
      - If the node didn't previously define the variable, add this node to defsites

Note that the last step is necessary since the added Φ-function is considered a new definition for the variable.

Then, we need to rename variables so that each variable is defined once. To do this:

- Maintain a stack for each variable
- Define the helper routine rename_block(BB) as:
  - For each instruction in BB:
    - Replace each variable use by a non-Φ-function with topmost stack definition
    - Rename the destination and add the new name to its corresponding stack
  - For each Φ-function in direct successors of BB:

        * Replace each variable use with topmost stack definition
    - For each block BB_{child} immediately dominated by BB, run rename_block(BB_{child})
    - Restore original states of variable stacks (before the current basic block was entered)
- Run rename_block(main)

Checkpoint 1

Given the following abstract assembly, first add Φ-instructions to the abstract assembly above as necessary. Then, rename variables so that the program is in SSA form.

```
1  main:
2    x <- 1
3    if (x >= 2) then goto L1 else goto L2
4  L1:
5    x <- x + 2
6    goto L3
7  L2:
8    x <- 3
9    goto L3
10 L3:
11   %eax <- x
12   return
```

Solution: We observe that the abstract assembly has four basic blocks (main, L1, L2, L3). L1 and L2 both have a non-empty dominance frontier, both being {L3}. Adding phi functions, we get:

```
1  main:
2    x <- 1
3    if (x >= 2) then goto L1 else goto L2
4  L1:
5    x <- x + 2
6    goto L3
7  L2:
8    x <- 3
9    goto L3
10 L3:
11   x <- phi()
12   %eax <- x
13   return
```

Then, for renaming variables, we observe that we first visit the main block, followed by any ordering of the other three blocks. This gives us:

```
1  main:
2    x_0 <- 1
3    if (x_0 >= 2) then goto L1 else goto L2
4  L1:
5    x_1 <- x_0 + 2
6    goto L3
```

```
7  L2:
8    x_2 <- 3
9    goto L3
10 L3:
11   x_3 <- phi(x_1, x_2)
12   %eax <- x_3
13   return
```

SSA Deconstruction

At some point before converting to x86, we need to deconstruct SSA, since Φ-functions aren't executable machine instructions. To do this, we want to our IR to be in **conventional SSA** (where the variables in Φ-functions never interfere). This is a form that we naturally receive from SSA construction, but certain optimizations may make the SSA non-conventional.

To fix this, we introduce the idea of splitting critical edges. A critical edge is an edge $x \to y$ such that x has more than one successor and y has more than one predecessor. To split such a critical edge, we insert a new node z, then replace the previous edge with $x \to z$ and $z \to y$.

Once we do have conventional SSA, SSA deconstruction is easy (unless we have optimizations like copy propagation). For every Φ-function assignment $x_0 \leftarrow \Phi(B_1 : x_1, B_2 : x_2, \ldots, B_n : x_n)$ (where $B_i : x_i$ refers to the definition of x_i arriving from a direct predecessor B_i), we insert the move $x_0 \leftarrow x_i$ to the parallel copy PC_i in B_i.

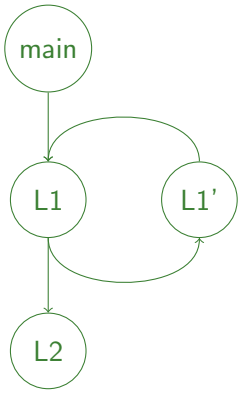
Once we have copy propagation, deconstructing SSA will become more involved. We'll need to generate a **location transfer graph** to identify a valid move ordering. If there are cycles in the graph, we'll need to break them. To do this, we can introduce a fresh temp to temporarily store the value of a temp in the cycle.

Checkpoint 2

Given the following abstract assembly, draw the CFG. Then, redraw the CFG, splitting any critical edges.

Solution:

Observe that the only critical edge in the CFG is L1's self-edge. Thus, after splitting, the CFG would look like so:



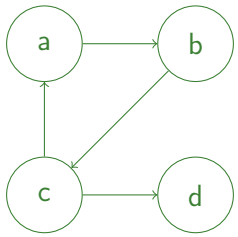

```
1  main:
2    x_0 <- 1
```

```
3   goto L1
4 L1:
5   x_2 <- phi(x_0, x_1)
6   x_1 <- x_2
7   if (true) then goto L1 else goto L2
8 L2:
9   %eax <- x_1
10  return
```

Checkpoint 3

Given the parallel copy PC($b \leftarrow a, c \leftarrow b, a \leftarrow c, d \leftarrow c$), draw a location transfer graph. Afterwards, convert it to sequential moves (breaking cycles as necessary).

Solution:



One possible solution is as follows (segmenting $b \leftarrow a$):

```
1 d <- c
2 a' <- a
3 a <- c
4 c <- b
5 b <- a'
```

SSA Resources

As Lab 2 onwards require that you implement SSA, these references may be helpful to read about SSA implementation intricacies in more detail:

- An alternate approach for SSA construction: We can use the approach described in this paper because C0 doesn't have gotos or breaks
- Paper that describes a dominance frontier construction algorithm
- SSA-based Compiler Design, a book that explains various aspects of SSA
- If you are planning on doing register allocation in SSA form:
  - Paper that describes register allocation on chordal graphs
  - Slides from a good presentation on SSA-based register allocation
  - Paper that describes an alternate approach to register allocation
  - Paper that provides useful information on deconstructing SSA after register allocation

Lab 2 Hints

- We **strong recommend** having register allocation and SSA working by the end of Lab 2.
- For the expression `if (a < 0) if (b < 0) x = 4 else x = 5`, x is not assigned if $a \geq 0$ (`else` binds to the most recent `if`).
- You have to add support for boolean variables now, and you will have to add support for pointers in Lab 4. Plan ahead when making design decisions to support different types in type checking and instruction selection.
- We suggest adding support for a `-O0` flag that disables register allocation and places all temps on the stack. Interference bugs fail in subtle, hard to understand ways.
- You can step through your programs with gdb. Place a breakpoint with `break _c0_main`, then use `step` to advance the program. If you're having trouble getting GDB working, see the Setup Guide on Piazza.