15-411: Compiler Design Spring 2024

Recitation 1: Register Allocation Solutions — 26 January

Liveness Analysis

The first step in assigning registers is to determine which temps interfere with each other. If a temp is defined on line ℓ, it interferes with all variables in liveout(ℓ).

While we used inductive definitions and inference rules to define liveness in lecture, the actual implementation of liveness in compilers usually does not involve that. Instead, we will use the following rules to construct live-in and live-out sets:

$$\text{LiveIn}(\ell) = \text{Uses}(\ell) \cup (\text{LiveOut}(\ell) - \text{Defs}(\ell))$$

$$\text{LiveOut}(\ell) = \bigcup_{s \in \text{succ}(\ell)} \text{LiveIn}(s)$$

Note that for straight-line code (Lab 1), you will only need to use the first rule, and you will only need to apply one backward pass across a program.

The second rule becomes necessary on programs with branching (Lab 2). If there are cycles in the control flow group, i.e. loops, then solving for liveness often requires repeated application of the rules until reaching a fixed point. We will discuss further this in future lectures and recitations.

The following is an program in 3-address abstract assembly. Although there is branching, there are no loops, so you should be able to compute all live-in/out sets by applying the rules in one backward pass.

```
 1  main:
 2    x <- 42
 3    t1 <- 2
 4    t2 <- x % t1
 5    if (t2 == 0) then goto L1 else goto 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    %eax <- z * x
16    return
```

Checkpoint 0

Analyze the above program to determine the live-out and live-in sets at each of the lines. Then draw the interference graph.

Solution:

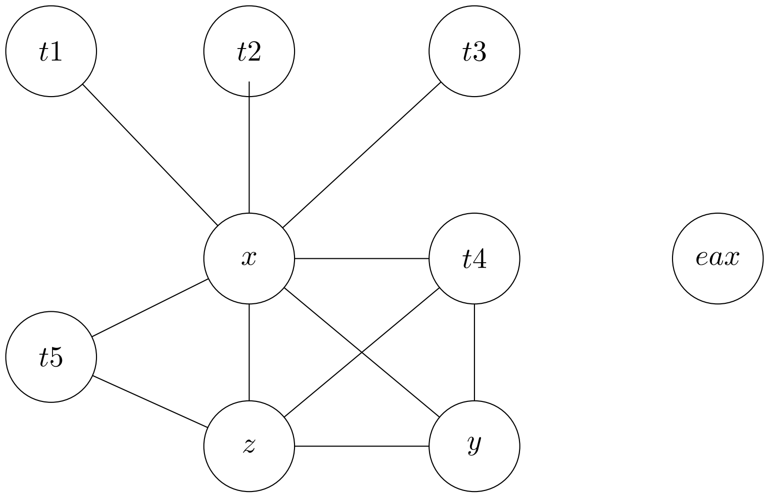
```
l.2:  def: x, use: none, liveout: x, livein: none
l.3:  def: t1, use: none, liveout: x, t1, livein: x
```

```
l.4:  def: t2, use: x, t1, liveout: t2, x, livein: x,t1
l.5:  def: none, use: t2, liveout: x, livein: t2,x
l.7:  def: x, use: x, liveout: x, livein: x
l.8:  def: z, use: none, liveout: z,x, livein: x
l.11: def: t3, use: none, liveout: t3, x, livein: x
l.12: def: z, use: t3, liveout: z,x, livein = t3, x
l.15: def: %eax, uses: z,x, liveout: none, livein: z,x
```

The interference graph is a star with `x` in the middle, and `%eax` is by itself.

Maximum Cardinality Search

Once we have used liveness information to construct the interference graph, we can color it, i.e. replace the temps with concrete registers or slots in stack memory (see Spilling Temps below). For the next few checkpoints, we will work with the following interference graph:



We will use the greedy algorithm to color in the interference graph. However, if we just pick an arbitrary order in which to color in the vertices, we may end up with very suboptimal colorings. Instead, we will use 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 1

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 2

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:

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

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 to be a register for most arithmetic operations.

One strategy of getting around this is 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 memory destination to `%r11d`, perform the operation using `%r11d`, then move the operation output in `%r11d` back to memory. In a functional language, you can implement this by casing on instruction type and produce either a singleton list with the input instruction, or a list with the moves into and out of `%r11d`.

Of course, the downside to such a strategy is that you have one fewer register to use during register allocation. If there are only a few memory-memory operations, then preemptively reserving `%r11d` may actually lead to worse results.

There are alternative strategies that do not have this issue – we encourage you to think of them and implement them into your compiler!