1 Optimization (Continued)

1.1 Optimization: Register Allocation

1.1.1 The Minimal Number of Locations

Consider the following program:

```
(let (b 4)
    (let (x 10)
        (let (i (if input
                    (let (z 11) (+ z b))
                    (let (y 9) (+ y 1))))
            (let (a (+ i 5))
                (+ a x)))))
```

To answer the second question, note that

- `i` and `x` need storage at the same time.
- `a` and `b` do not need storage at the same time[1].

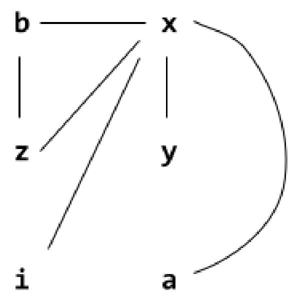
How many memory locations are needed? We'll look at the program from the *end* to the beginning.

- We first begin by looking at what variables are in use at the end. In this case, `a` and `x` are in use. The set of all variables in use is
$$\{a, x\}.$$
- We're going to go back "up" the program. When we get to a `let`-bindings, we're going to remove it from the set of variables that are in use right now. In the next level, we're *using* `i` and `x`, but we aren't using `a` here since `a` is being created. The set of all variables in use is
$$\{i, x\}.$$
- The `if`-expression is more interesting. We need to consider both branches of the `if`-expression. Note that, in this step, `i` is being created, so we don't have access to `i` yet.
  - Looking at the end of the "else" branch, at the body of the `let` binding, notice how `y` is being used. `x` is still around. The set of all variables in use is
  $$\{y, x\}.$$
  - Looking at the end of the "then" branch, at the body of the `let` binding, notice how `z` and `b`[2] are in use. As usual, `x` is still around. The set of all variables in use is
  $$\{z, b, x\}.$$
- At the `let`-binding for `i` (*not* in the body), we no longer have `z` or `y`, and `i` is being initialized here (so we aren't using `i` here). Thus, this gives us the variables in use
$$\{x, b\}.$$
- Moving "up" the program to the `let`-binding for `x`, we now only have the variables in use $\{x\}$.
- Finally, moving "up" the program to the `let`-binding for `b`, we have the variables in use $\emptyset$.

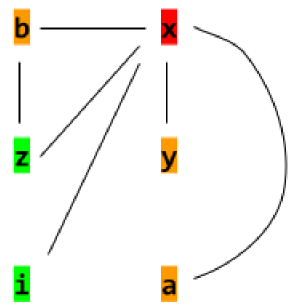
[1] Notice how we only use `b` once: in the `if`-expression. After that, we don't use `b` again.

[2] Even though `b` is defined at the top, this is the first time we're seeing `b` in use.

This information is telling us what variables need to be stored at the same time. Something we can do with this information is turn this into a **graph** where there's an edge between two variables *if* they're in use at the same time.



This is a graph where if there are two variables that had to be live at the same time, then there is an edge. How do we make it so we can have a set of locations where each variable can be assigned to a register that's different from all the things it conflicts with? This problem is known as **graph coloring**. The idea is that we want to find k colors assigning $1 \ldots k$ to each node such that k is minimal and no edge has the same index for both nodes. A coloring for this graph is



We only need 3 colors! In terms of what our compiler would output, we would end up with the environment

```
{x: 1, b: 2, y: 2, a: 2, z: 3, i: 3}
```

In other words, `x` gets abstract location 1, `b` gets abstract location 2, and so on. Note that this makes a few assumptions:

- All intermediates are carefully named and used (no useless temporaries).
- Assuming all temporaries are explicit, this could replace `depth(e)`. Note that this means *simple constants!*
- All variables are distictly named (although we can rename all non-distinct names if needed).

1.1.2 Algorithm

The algorithm for this process is as follows:

- Visit last, or innermost, expression first. THis means recurse, then working with result.
- Track set of variables we have seen used, then remove from set at the let-bindings.

So, going back to the example code, we have the following set of active variables.

```
(let (b 4)                                  ; {}
    (let (x 10)                             ; {b}
        (let (i (if input                   ; {b, x}
                    (let (z 11) (+ z b))    ; {z, b, x}
                    (let (y 9) (+ y 1))))   ; {y, x}
            (let (a (+ i 5))                ; {i, x}
                (+ a x)))))                 ; {a, x}
```

For each pair of active variables that appear at the same time, we draw an edge between them in the graph.