1 Optimization (Continued)

This section continues the previous section.

1.1 Optimization: Register Allocation

Let's consider the following code:

```
(let (n (+ 5 9))
    (let (m (+ 2 3))
        (let (x (+ n 1))
            (let (y (+ m 2))
                (+ x y)))))
```

The corresponding assembly[1], along with the corresponding code from the above, is shown below.

```
sub rsp, 40
  mov rax, 10
  mov [rsp + 0], rax    ; LHS of (+ 5 9)
  mov rax, 18
  add rax, [rsp + 0]

  mov [rsp + 0], rax    ; Variable n in (let (n ...))

  mov rax, 4
  mov [rsp + 8], rax    ; LHS of (+ 2 3)
  mov rax, 6
  add rax, [rsp + 8]

  mov [rsp + 8], rax    ; Variable m

  mov rax, [rsp + 0]    ; Variable n lookup
  mov [rsp + 16], rax   ; LHS of (+ n 1)
  mov rax, 2
  add rax, [rsp + 16]

  mov [rsp + 16], rax   ; Variable x

  mov rax, [rsp + 8]    ; Variable m lookup
  mov [rsp + 24], rax   ; LHS of (+ m 2)
  mov rax, 4
  add rax, [rsp + 24]

  mov [rsp + 24], rax   ; Variable y

  mov rax, [rsp + 16]   ; Variable x lookup
  mov [rsp + 32], rax
  mov rax, [rsp + 24]   ; Variable y lookup
  add rax, [rsp + 32]
add rsp, 40
```

One thing to notice immediately is that we reused some memory locations. One example is `[rsp + 8]`, which is where we stored both a temporary for addition and a value associated with a variable. We can generalize how many memory locations we ultimately *will* use by using the `depth` function. In particular, if `depth(expr)` $\leq$ Available Registers, then we can avoid memory entirely.

[1] With tag checks removed to make the assembly more concise.

There are two questions we should now consider.

1. (x86_64.) What registers should we use?

    We can use the registers `rbx`, `r12`, `r13`, `r14`, which are callee-saved registers. Note that we aren't using `r15` because this register is specifically the heap pointer.

2. (Design.) How should we implement this?

    We can create a `Loc` *enum* that holds either a register or a stack location (offset). Then, our environment can be represented by `HashMap<String, Loc>`.

    Suppose we have a list of registers that we can use. We can create a `get_loc` function which takes a stack index and returns the new location to be used; this might look something like

    ```
    let regs = [...];
    get_loc(si):
        if si < regs.size():
            return regs[si];
        else:
            return Stack(si - regs.len());
    ```

    Then, we can use this location to update the environment, like

    ```
    ...
    | ELet(x, val, body) => {
        env.update(x, get_loc(si));
    }
    ```

    Note that, while this is an *improvement* to how our program is compiled, this can still be made a *lot better*. Some other implementation notes to consider include:

    - We need to add code to save and restore registers in function definitions.
    - We need to compute stack size based on `depth - available registers`.

    Some improvements we could make to what we have so far include

    - Registers for outer bindings and stack for inner bindings.
    - Frequency matters.
    - Precompute registers and locations for all variables and temporaries across functions.
    - Are we using the minimal number of locations? (e.g., is the depth minimal?)

Remark: The register allocation algorithm we're talking about, which uses an idea similar to `depth`, is similar to the *Sethi-Ullman algorithm.*

1.1.1 The Minimal Number of Locations

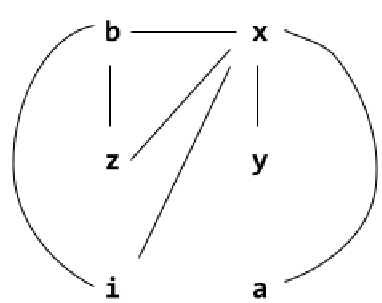
Consider the following program:

```
(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)))))
```

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 and do some unioning with the set of all variables in use.
  - Looking at the end of the "else" branch, at the body of the `let` binding, notice how `y` is being used. `x` and `i` are still around. The set of all variables in use is
  $$\{y, i, x\}.$$
  - 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` and `i` are still around. The set of all variables in use is
  $$\{z, b, i, x\}.$$
  - Here, we need to union the two branches. So, this gives us the variables in use
  $$\{z, b, i, x, y\}.$$
- 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 $\emptyset$.

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 is known as **graph coloring**.

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