Let's add (+ <expr> <expr>) to the compiler, and (let (x <expr>) <expr>)

```
(*
expr := <number>
    | (add1 <expr>)
    | (+ <expr> <expr>)
    | (let (<name> <expr>) <expr>)
    | <name>
*)
enum Expr {
  Num(i32),
  Add1(Box<Expr>),
  Plus(Box<Expr>, Box<Expr>),
}

fn compile_expr(e : &Expr, si: i32 ) -> String {
  match e {
      Expr::Num(n) => format!("mov rax, {}", *n),
      Expr::Add1(subexpr) => {
          compile_expr(subexpr, si) + "\nadd rax, 1"
      },
      Expr::Plus(e1, e2) => {
         let e1_instrs = compile_expr(e1, si);
         let e2_instrs = compile_expr(e2, si + 1);
         let stack_offset = si * 8;
         format!("
            {e1_instrs}
            mov [rsp - {stack_offset}], rax
            {e2_instrs}
            add rax, [rsp - {stack_offset}]
         ")
      }
      Expr::Let(x, e, body) => {
         let e_instrs = comp_e(e, si);
         let b_instrs = comp_e(body, si+1, env.update(x, si));
         format!("
            {e_instrs}
            mov [rsp - {si*8}], rax
            {b_instrs}
         ")
      }
  }
}
```

Let("x", ...)

Let("y", ...)

Id("x")

Id("y")

env: HashMap<String, int>

Let(String, Box<Expr>, Box<Expr>)

Id(String)

(+ 4 (+ 100 20)) assume si starts at 2

What assembly is produced?

```
mov rax, 4
mov [rsp – 16], rax
mov rax, 100
mov [rsp – 24], rax
you fill in the rest as review!
```

mov rax, 20
add rax, [rsp - 24]
add rax, [rsp - 16]

```
(let (x 10)
  (let (y 10)
    (+ x y)))
```

What assembly should we produce?

mov rax, 10
mov [rsp-16], rax (X)

mov rax, 10
mov [rsp-24], rax (Y)
mov rax, [rsp-16] (X)
mov [rsp-32], rax
mov rax, [rsp-24] (Y)
add rax, [rsp-32]

Let's agree on what each of these programs should evaluate to…

```
(let (x 10)
  (let (y 10)
    (+ x y)))
```
⇒ 20

```
(+ (let (x 10) (add1 x))
   (let (y 7) (+ x y)))
```
⇒ (+ (add1 10) (let ...)) ⇒ (+ 11 (let ...)) ⇒ (+ 11 (+ x 7))

```
(let (x (let (y 10) (add1 y)))
  (add1 x))
```
11

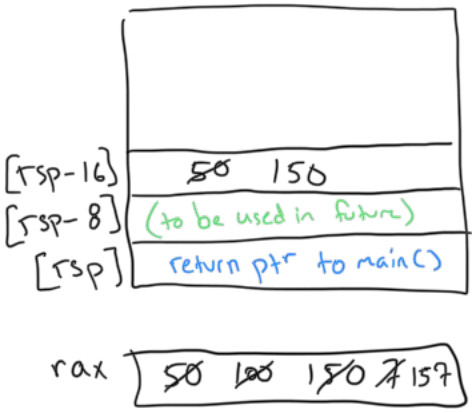
(let (x 11) (add1 x))
12

⇒ (+ 11 error!

```
(let (x 10)
  (let (x (add1 x))
    (+ x 10)))
```
error
⇒ 21
~~⇒ 20~~

(+ (+ 50 100) 7)

```
mov rax, 50
mov [rsp – 16], rax
mov rax, 100
add rax, [rsp – 16]
mov [rsp – 16], rax
mov rax, 7
add rax, [rsp – 16]
```



mov <reg>, <value> *move <value> into reg, <value> could be a constant, another reg, or memory location*

mov [<reg> + <offset>], <value> *move <value> into memory at address [<reg> + <offset>], value could be const or reg*
*(value **cannot** be another mem location)*

add <reg>, <value> *add <value> to the value in <reg> and store the result in <reg>*
value could be const, register, or a memory location

Rust Immutable Data Structures: https://docs.rs/im/latest/im/

Module im::hashmap

An unordered map.
An immutable hash map using hash array mapped tries.
Most operations on this map are O($\log_x$ n) for a suitably high *x* that it should be nearly O(1) for most maps. Because of this, it's a great choice for a generic map as long as you don't mind that keys will need to implement Hash and Eq.

pub fn update(&self, k: K, v: V) -> Self
Construct a new hash map by inserting a key/value mapping into a map.
If the map already has a mapping for the given key, the previous value is overwritten.
Time: O(log n)
Examples

```
let map = hashmap!{};
assert_eq!(
  map.update(123, "123"),
  hashmap!{123 => "123"} );
```

pub fn get<BK>(&self, key: &BK) -> Option<&V> where
BK: Hash + Eq + ?Sized,
K: Borrow<BK>,
Get the value for a key from a hash map.
Time: O(log n)
Examples

```
let map = hashmap!{123 => "lol"};
assert_eq!( map.get(&123), Some(&"lol") );
```