(+ (100 50) 2)

Let's add local variables and **binary ops** to our compiler

```
expr := <number>
     |  (add1 <expr>)
     |  (let (<name> <expr>) <expr>)
     |  <name>
     |  (+ <expr> <expr>)
```

Result	Programs	Stack Layout	Assembly

```
((+ 1 2) 3)
```

```
(+ 1 (+ 2 (+ 3 4)))
```

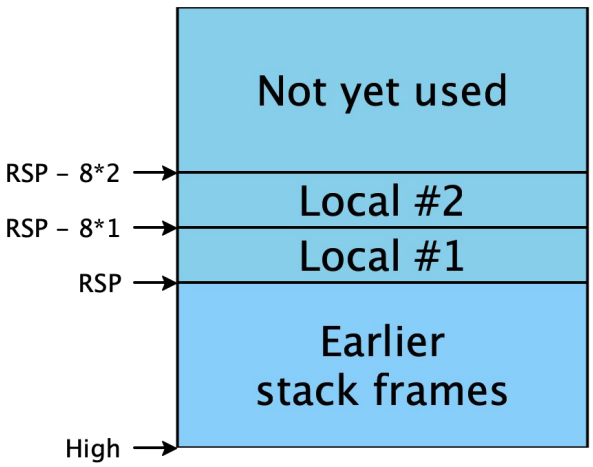
```
(let (x 10)
  (let (y 10)
     (+ x y)))
```

```
(let (x 10)
  (let (x (add1 x))
    (+ x 10)))
```

```
(+ (let (x 10) (add1 x))
   (let (y 7) (+ x y)))
```

```
(+ (let (x 10) (add1 x))
   (let (y 7) (+ 10 y)))
```

(+ e1 e2)



Let's add **local variables** and **binary ops** to our compiler

```
expr := <number>
      | (add1 <expr>)
      | (let (<name> <expr>) <expr>)
      | <name>
      | (+ <expr> <expr>)
```

(+ (100 50) 2)

What assembly is produced?

```
enum Expr {
  Num(i32),
  Add1(Box<Expr>),


}
```

```
enum Val {
  Reg(Reg),
  Imm(i32),


}
```

```
enum Reg {
  RAX,


}
```

```
enum Instr {
  IMov(Val, Val),
  IAdd(Val, Val),
  ISub(Val, Val),
  IMul(Val, Val),
}
```

```
fn compile(e:&Expr,env:&Env                ) -> String {
  match e {
    Expr::Num(n) => {
      format!("mov rax, {}", *n)
    }
    Expr::Add1(e1) => {
      let e1_code = compile(e1, env);
      format!("{e1_code}
               add rax, 1")
    }
    Expr::Var(x) => {
      let offset = env.get(x).unwrap();
      format!("mov rax, [rsp - 8*{offset}]")
    }
    Expr::Let(x, e1, e2) => {
      let e1_code = compile(e1, env, is);
      let new_env = env.update(x.clone(), offset);
      let e2_code = compile(e2, &new_env, is + 1);
        format!("{e1_code}
                 mov [rsp - 8*{is}], rax
                 {e2_code}")
    }
    Expr::Plus(e1, e2) => {









    }
}
```

What should these evaluate to? Why?

```
(let (x 5)
  (if (= x 10) (+ x 2) x))
```

```
(if 5 true false)
```

```
(+ 7 true)
```

```
(= true 1)
```

What should be in RAX after these are done evaluating? Why?

```
5
```

```
-3
```

```
true
```

```
false
```

```
(= 3 5)
```

```
(+ 4 7)
```

```
enum Expr {
  Num(i32),
  True, False,
  If(Box<Expr>, Box<Expr>, Box<Expr>)
  Eq(Box<Expr>, Box<Expr>)
  Add1(Box<Expr>),
  Plus(Box<Expr>, Box<Expr>),
  Let(String, Box<Expr>, Box<Expr>)
  Id(String),}

fn compile_expr(e : &Expr, si : i32, env : &HashMap<String, i32>) -> String {
  match e {
      Expr::Num(n) =>

      Expr::True =>

      Expr::False =>

      Expr::Add1(subexpr) => …,
      Expr::Plus(e1, e2) => …,
      Expr::Let(x, e, body) => …,

      Expr::If(cond, thn, els) => {









      }

      Expr::Eq(e1, e2) => {









      }
}
```

This is 64 bits: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000

This is 5: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0101

This is 5 shifted 1 to the left, AKA 10: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 1010

If we're OK with 63-bit numbers, can use LSB for *tag*

0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 1010 = 5

0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0001 = false

0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0011 = true

What does this mean for code generation?

What should we do the next time we need a new type? (string, heap-allocated object, etc.)

Condition Codes (that matter for us): Overflow, Sign, Zero
many instructions set these; arithmetic, shifting, etc. mov does not

cmp <reg>, <val> *compute <reg> - <val> and set condition codes (value in <reg> does not change)*
some cases to think about:

<reg> = -2^64, <val> = 1 Overflow: ___ Sign: ___ Zero: ___

<reg> = 0, <val> = 1 Overflow: ___ Sign: ___ Zero: ___

<reg> = 1, <val> = 0 Overflow: ___ Sign: ___ Zero: ___

<reg> = -1, <val> = -2 Overflow: ___ Sign: ___ Zero: ___

test <reg>, <val> *perform bitwise and on the two values, but don't change <reg>, and set condition codes as appropriate. Useful for mask checking.* test rax, 1 *will set Z to true if and only if the LSB is 1*

<label>: *set this line as a label for jumping to later*

jmp <label> *unconditionally jump to <label>*

jne <label> *jump to <label> if Zero is not set (last cmped values not equal)*

je <label> *jump to <label> if Zero is set (last cmped values are equal)*

jge <label> *jump to <label> if Overflow is the same as Sign (which corresponds to >= for last cmp)*

jle <label> *jump to <label> if Zero set or Overflow != Sign (which corresponds to <= for last cmp)*

shl <reg> *shift <reg> to the left by 1, filling in least-significant bit with zero*

sar <reg> *shift <reg> to the right by 1, filling in most-significant bit to preserve sign*

shr <reg> *shift <reg> to the right by 1, filling in most-significant bit with zero*