Let's add **input** and **booleans** and **assignments** and **loops** to our compiler

```
enum Expr {
  Num(i32),
  Add1(Box<Expr>),
  Plus(Box<Expr>, Box<Expr>),
  Let(String, Box<Expr>, Box<Expr>)
  Id(String),
  Input,
  True, False,
  If(Box<Expr>, Box<Expr>, Box<Expr>)
  Eq(Box<Expr>, Box<Expr>)
  Set(String, Box<Expr>),
  Block(Vec<Expr>),
  Loop(Box<Expr>),
  Break(Box<Expr>),
}
```

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)
```

```
fn snek_print(val: i64) -> i64 {
  println!("{val}");
  return val;
}

fn parse_arg(v: &Vec<String>) -> i64 {
  if v.len() < 2 {
    return 1; // default
  }
  s.parse::<i64>().unwrap()
}

fn main() {
  let args: Vec<String> = env::args().collect();
  let input = parse_arg(&args);

  let i: i64 = unsafe { our_code_starts_here(input) };
  snek_print(i);
}
```

tt/ff/eq

set!/block

loop/break

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*