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: ___*

`<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*