Today:

How can we write functions that use C (like +), but preserve Scheme's ability to manipulate them?

- "primitives"
- function pointers in C

---

We already have "special forms"

- if
- quote
- let

} handled completly individually by your code

eval

if

"if" ——————

"quote" ——————

"let" ——————

We have user defined functions
via lambda

eval

:

else if CONS-TYPE

apply fn crected lambda

---

How do we do functions like + ?

Two bad approaches:

(1) Define in Scheme yourself
what it means to add, from
1st principles

(2) Make it a special form
- add it the long list of special
cases in your code.
- Sort of works

Issues:

- that section of code in your interpreter gets really long
- also lose that functions are data
- e.g. lose ability to do

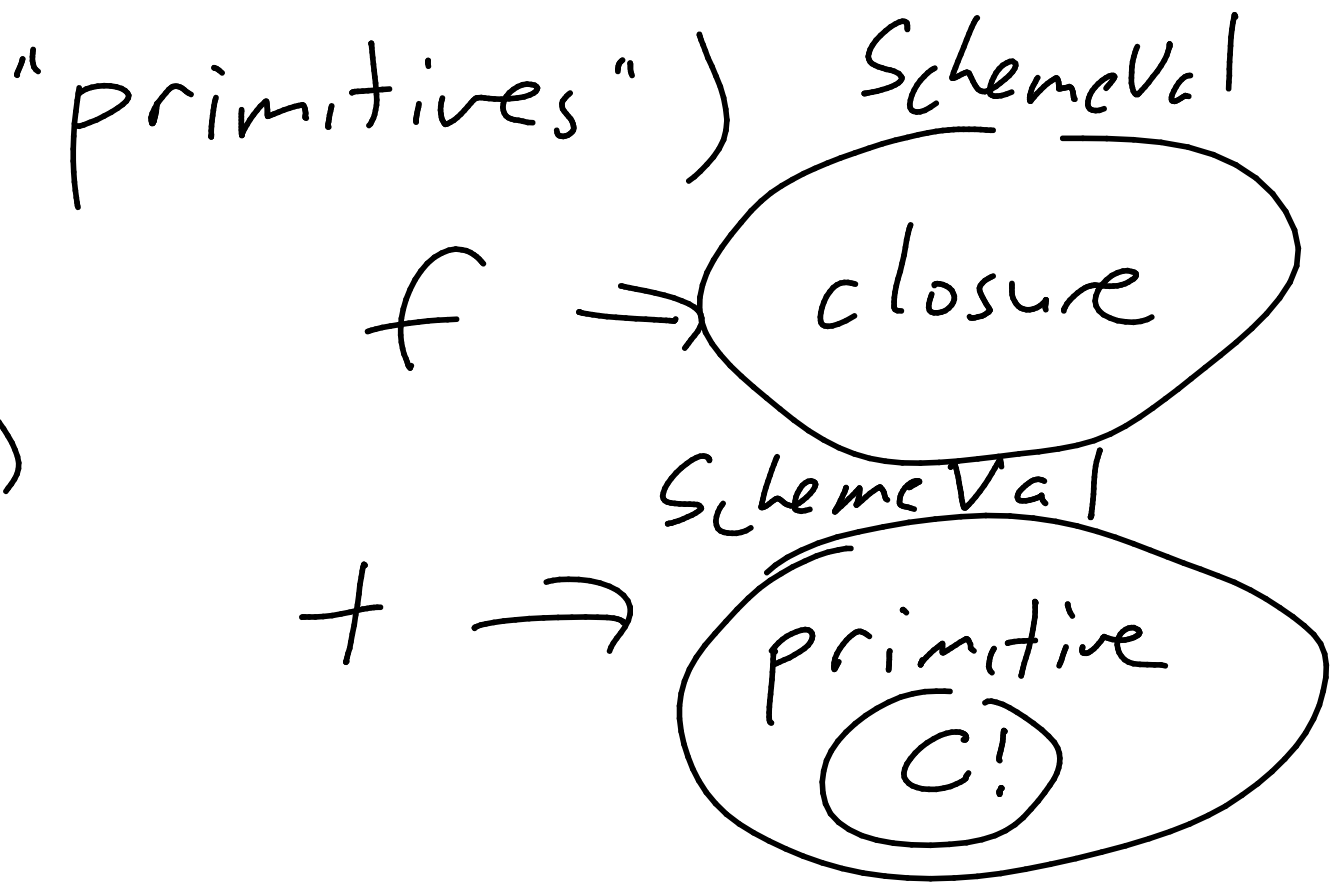
```
(define x +)
(x 3 5) → 8
```

---

How will we do it?

We're going to make C functions that Scheme variables can point to. ("primitives")

```
(define f
  (lambda ()
    3))
```



To do this, we need a C trick, which is being able to have a pointer to a function.

return type

param types

```
int doit(int (*f)(int, int),
         int a, int b) {
    return f(a, b);
}
```

f is a pointer

You will create functions like

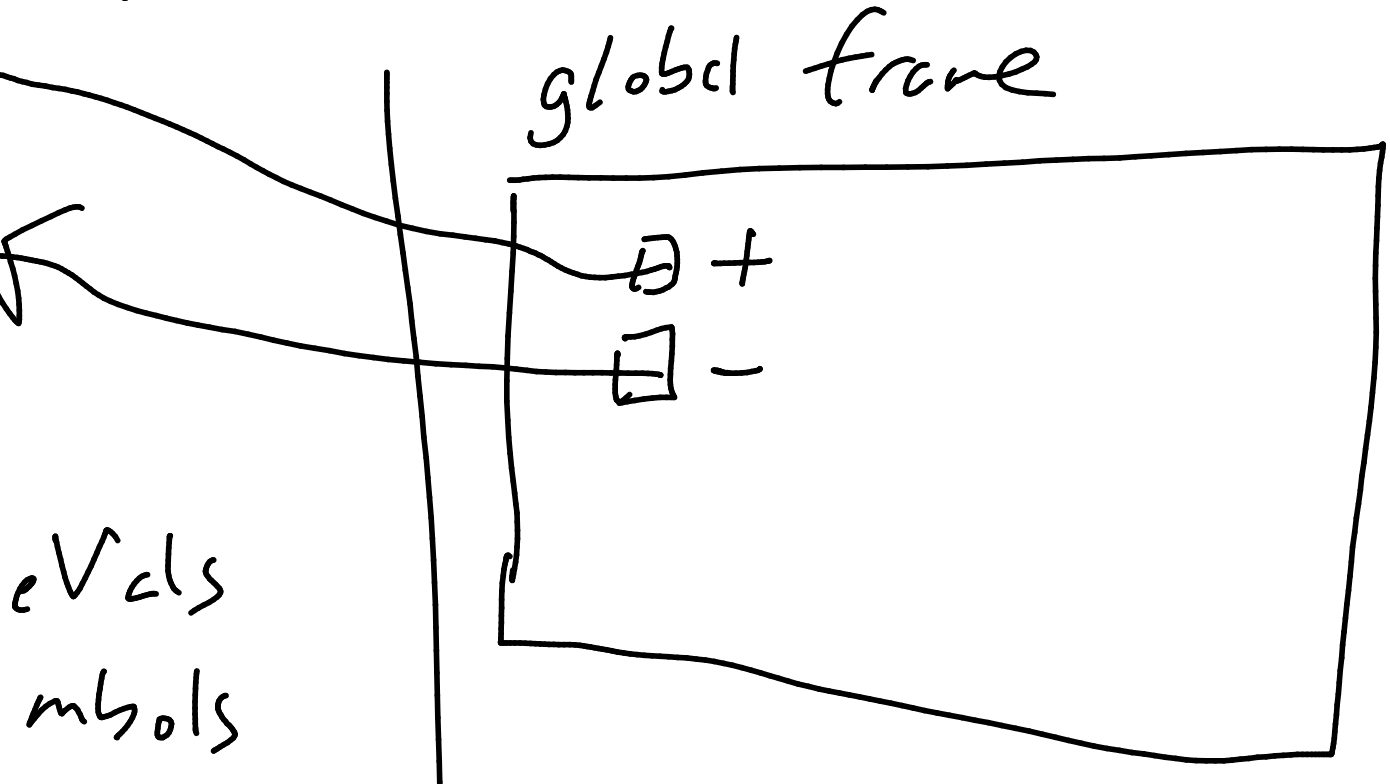
add

subtract

.

.

etc




wrap in SchemeVals
⇒ bind to symbols

Again, how is this different than just adding as another special form, special-cased in eval?

Special form version

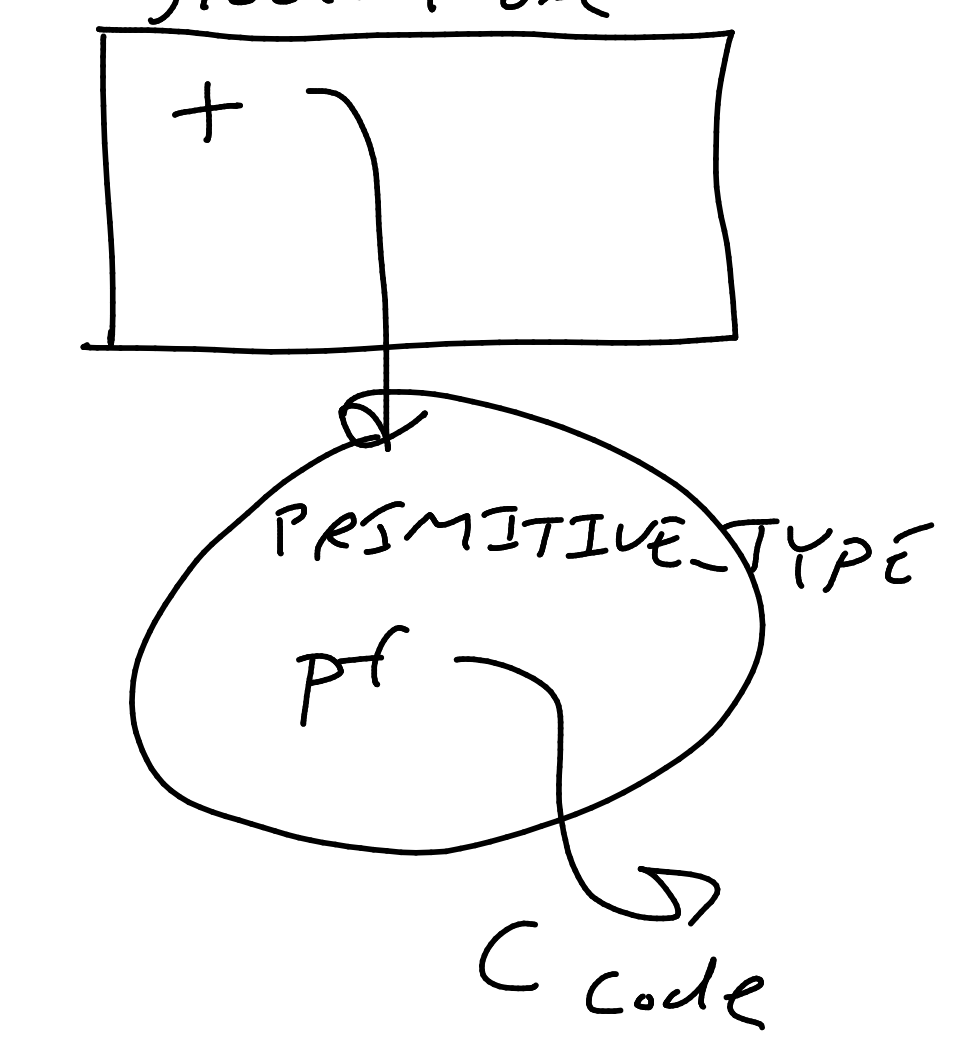
eval

if —

else if + {

answer = a + b

Primitive version




eval

if —

else

apply primitive ($f_1$ ...)

w/ the primitive version, I can do

```
(define x +)
(x 3 5) => 8
```

---

```
(lambda doit
  (lambda (f x y)
    (f x y)))
(doit + 3 5) → This works!
```

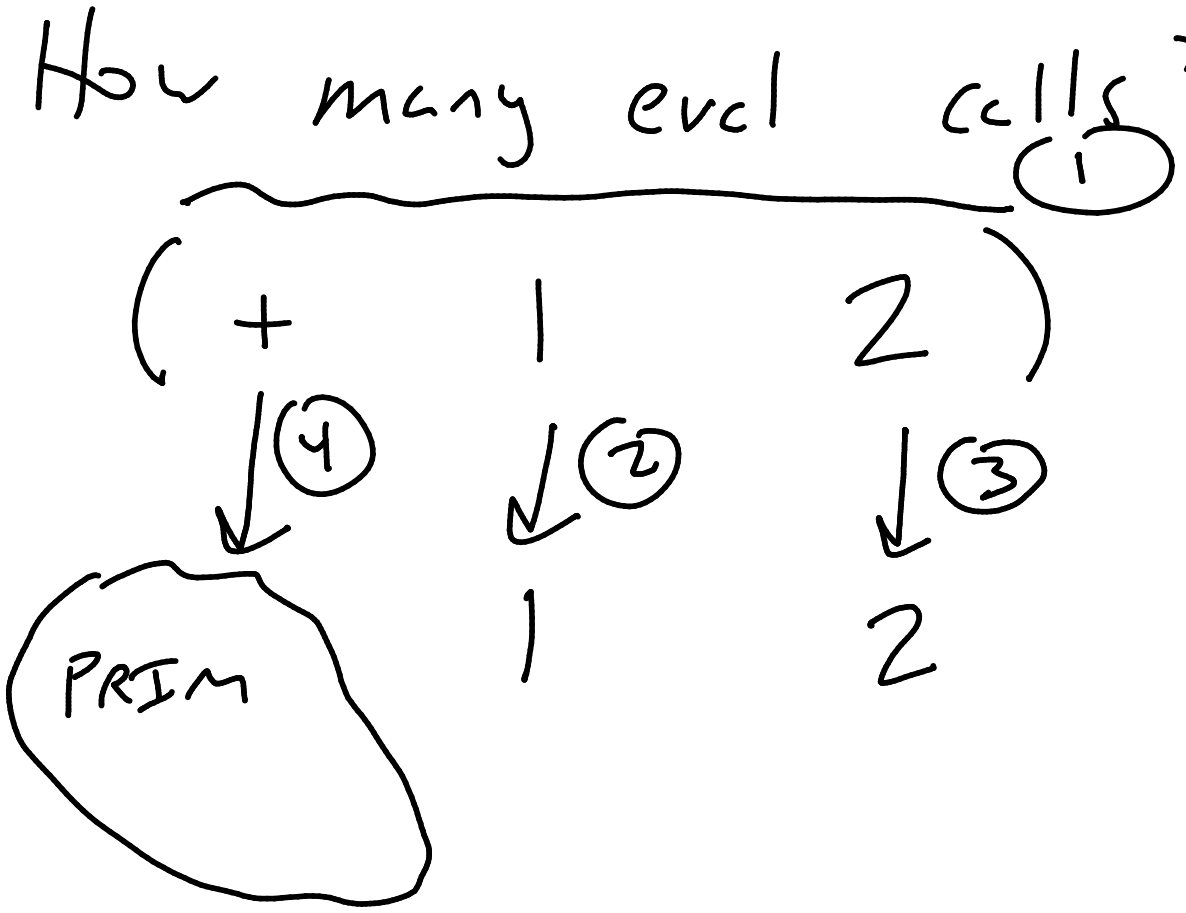
But

```
(doit if #t #f)
```

fails because `if` is not a symbol bound to a value

```
(doit quote #t #f)
(doit (quote #t) #f)
```

How many eval calls?




eval (1)

(if #t 3 4)