```
expr := <number> | <name> | true | false
      | (if <expr> <expr> <expr>)
      | (let (<name> <expr>) <expr>)
      | (+ <expr> <expr>)
      | (< <expr> <expr>)
      | (set <name> <expr>)
      | (fun (<name> : <t>) : <t> <expr>)
      | (<expr> <expr>)
```

```
type expr =
  | ENum of int | EBool of bool | EId of string
  | EIf of expr * expr * expr
  | ELet of string * expr * expr
  | EPlus of expr * expr
  | ELess of expr * expr
  | ESet of string * expr
  | EApp of expr * expr
  | EFun of string * typ * typ * expr
```

What's weird/interesting/wrong?

arg ret

```
t := Num | Bool | (<t> -> <t>)
```

```
and typ = TNum | TBool | TArrow of typ * typ
```

arg ret

```
prog := <expr>
```

```
type prog = expr
```

Defs?! Replaced by fun!

```
expr := <number> | <name> | true | false
      | (if <expr> <expr> <expr>)
      | (let (<name> <expr>) <expr>)
      | (+ <expr> <expr>)
      | (< <expr> <expr>)
      | (set <name> <expr>)
      | (<name> <expr> <expr>)
```

```
type expr =
  | ENum of int | EBool of bool | EId of string
  | EIf of expr * expr * expr
  | ELet of string * expr * expr
  | EPlus of expr * expr
  | ELess of expr * expr
  | ESet of string * expr
  | EApp of string * expr
```

```
def := (def <name> (<name> : <t>) : <t>
         <expr>)
```

```
type def =
  | DFun of string * string * typ * typ * expr
```

```
t := Num | Bool
```

```
type typ = TNum | TBool
```

```
prog := def ... <expr>
```

```
type prog = def list * expr
```

```
(let (twox (fun (x : Num) : Num (+ x x))
  (twox 10) )

(let (and (fun (b1 : Bool) : (Bool -> Bool)
            (fun (b2 : Bool) : Bool (if b1 b2 false))))
  ((and true) false))
```

closes over b1
remembers b1

and : Bool -> (Bool -> Bool)   type of and

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash (e_1\ e_2) : \tau_2}$$

$$\frac{(x,\tau)::\Gamma \vdash e : \tau_R}{\Gamma \vdash (\mathrm{fun}\ (x : \tau) : \tau_R\ \ e) : \tau \to \tau_R}$$

```
((fun (x:Num): Num (+ x 1)) 10)
```

Did we lose anything meaningful?

A: Yes

B: No

- Recursion?
- Mutual recursion?
- Program structure

```
(let (sum (fun (n:Num) : Num
             (if (< n 1) 0
                 (+ (sum (- n 1) n)))))
  (sum 10))
```

unbound id!

```
(letrec (sum ...) (sum 10))
```

$\Downarrow$ means

```
(let (sum (null (Num -> Num)))
  (set sum (fun ...))
  ⋮
  (sum 10))
```

```
(letrec (sum .... sum ...)
```

new error using sum before defined

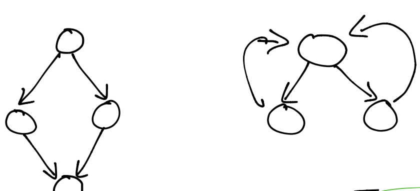
EFun(arg, _, _, e) ->
  let body_is = compile e [(arg, 3)]

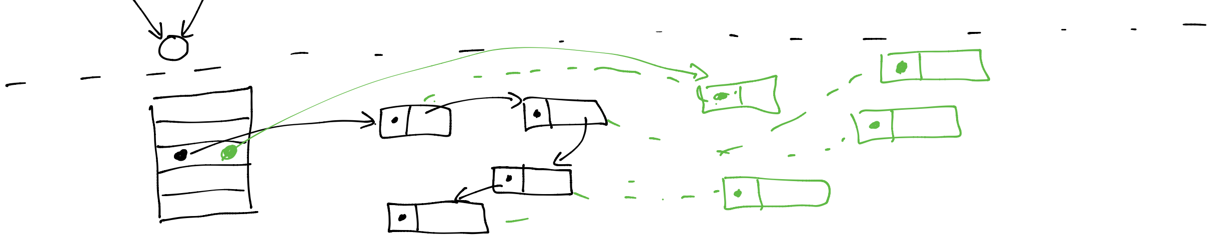
```
" jmp skip
  label:
    body_is
    ret
  skip:
  mov [r15], label
  mov [r15+8], unbound/free id1
  mov [r15+16], unbound/free id2
```

EApp(e_f, e_a) ->
  let f_is = compile e_f ... in

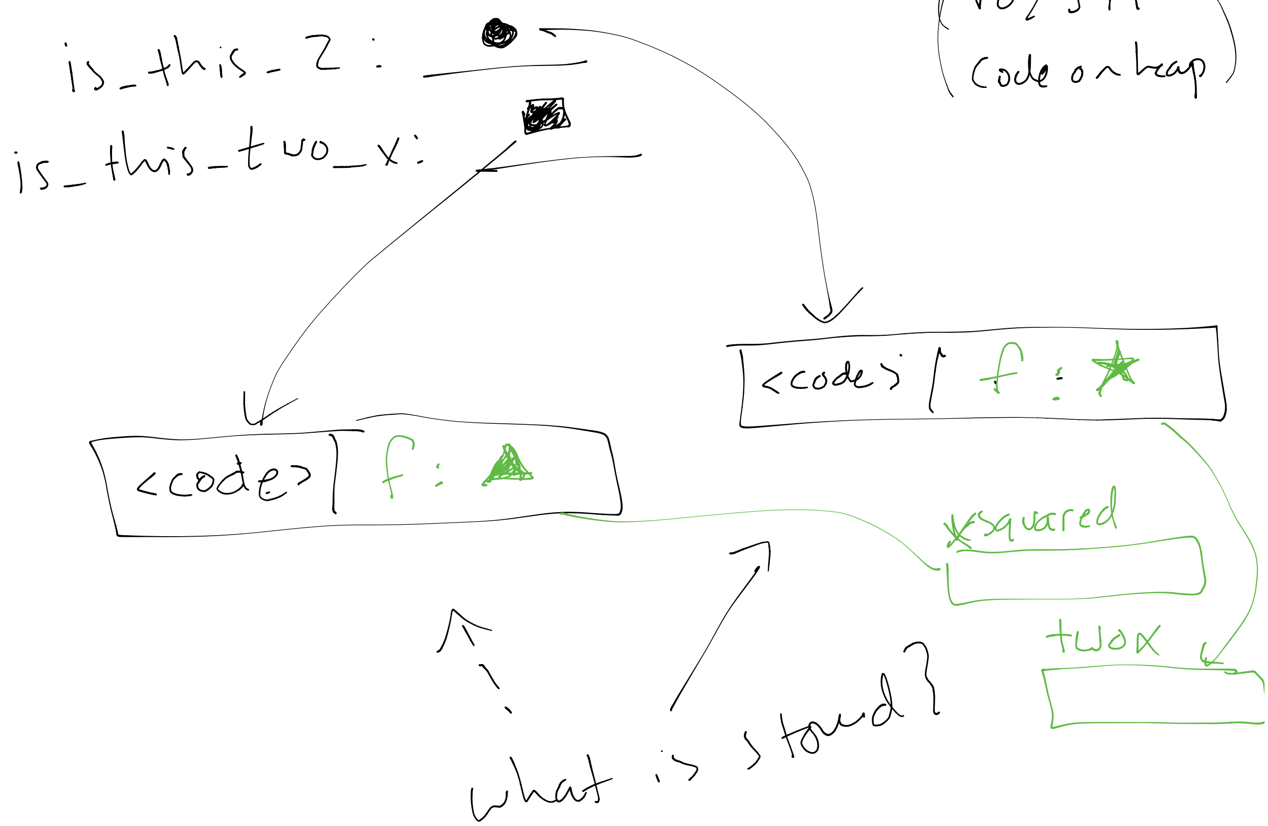
```
"  mov rbx, [rax]
   ... move args ...
   jmp rbx "
```

Checking for cycles is not the same as checking for visiting same node twice.





(V8/JIT
code on heap)



<code>:

isprime:

ret

```
def ddx(f):
  def fprime(x):
    return (f(x + 0.001) - f(x)) / 0.001
  return fprime
```

f is free
or unbound
relative to
fprime