6/3/19

IB ::= True | False

true : IB
False : IB
} is type of bool

Function = input and product output

neg : IB → IB

neg true = False

neg false = True

$|IB|^{|IB|} = 2^2 = 4$

How many function are there for boolean to boolean?

$2^2 = 4 \rightarrow 4^2 = 16$

and : IB → IB → IB

and true true = true

and _ _ = false

true	false
false	true
false	false

xor : 𝔹 → 𝔹 → 𝔹

xor true true = false

xor false false = false

xor false true = true

Natural number :

ℕ := zero
| succ ℕ

zero : ℕ

one = succ zero

two = succ one or succ (succ zero)

three = succ two

(+) : ℕ → ℕ → ℕ

(succ x) + y = succ (x + y)

zero + y = y

succ zero + succ (succ zero) ≡ succ (z + ss z)
≡ succ (succ (succ zero))

6/10/19

$List_N$:= Empty
| Cons N $List_N$

$List_B$:= Empty
| Cons B List

string := Empty
| Cons Char string

List a := Empty
| Cons a (List a)

[↑, ↑, ↓, ↓, ←, →, ←, →] : List Direction

sum : List N → N

sum Empty = Zero

sum Cons(x, xs) = plus x (sum xs)

```
tail : List a -> List a
tail Empty = Empty
tail Cons(x, xs) = xs
```

```
head : List a -> Maybe a
head Empty = Nothing
head Cons(x, xs) = Just x
```

```
Maybe a ::= Nothing
          | Just a
```

```
plus5 : Maybe N -> Maybe N
plus5 Nothing = Nothing
plus5 Just(x) = Just(plus5 x 5)
```

```
id : a -> a
id x = x
```

(this function take any type of input and give the same of output)

```
Printy ::= [Nat + N
           PList (List N)
```

6/11/19

factorial : $\mathbb{N} \to \mathbb{N}$

factorral zero = succ zero

factorral (succ zero) = succ zero

factorial (succ x) = mult (succ x) (factoral x)

fact(5)

factorral zero = succ zero

factorral (succ zero) = succ zero

factorral (succ x) = mult (succ x) (factorral 5)

fact(5) ≡ mult 5 (fact 4)

≡ mult 5 (mult 4 (fact 3))

≡ mult 5 (mult 4 (mult 3 (fact 2)))

≡ mult 5 (mult 4 (mult 3 (mult 2 (fact 1))))

≡ mult 5 (mult 4 (mult 3 (mult 2 (succ zero))))

≡ mult 5 (mult 4 (mult 3 2))

≡ mult 5 (mult 4 6) ≡ mult 5 24

≡ 120

$plus_1 : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$

$plus_1\ zero\ y = y$

$plus_1\ (s\,x)\ y = s(plus\ x\ y)$

$plus_2 : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$

$plus_2\ (s\,x)\ y = plus\ x\ (s\,y)$ } tail recursion

$plus_1\ 5\ 2 \equiv s(plus_1\ 4\ 2)$

$\equiv s(s(plus_1\ 3\ 2))$

$\equiv s(s(s(plus_1\ 2\ 2)))$

$\equiv s(s(s(s(plus_1\ 1\ 2))))$

$\equiv s(s(s(s(s(plus_1\ 0\ 2)))))$ } general recursive function

$plus_2\ 5\ 2 \equiv plus\ 4\ 3$

$\equiv plus\ 3\ 4$

$\equiv plus\ 2\ 5$

$\equiv plus\ 1\ 6$

$\equiv plus\ 0\ 7 \equiv 7$

accumulator passing style (APS)

1) find nasty function

2) write helper function

3) $$$

```
fact_helper : N -> N -> N
fact_helper zero acc = (succ zero)
fact_helper (succ zero) acc = acc
fact_helper (succ x) acc =
        or fact_helper(x, (mult(S(x),acc)))
```

4! · 5

3! · 4 · 5 = 3! · 20

```
factorial : N -> N
factorial x = fact_helper x 1
```

① write aux function with accumulate

a) handle recursion case (build up the acc.)

b) handle base case (process acc. and return an output)

② write a function you can show your parents (wrap it up in a nice package)

```
sum_aux : List N -> N -> N

sum_aux Empty acc = acc

sum_(cons x xs) acc = sum_aux (xs (plus x acc)

sum : List N -> N

sum xs = sum_aux xs Zero
          -> variable point

sum_aux -> N -> List N -> N

sum_aux . acc Empty = acc

sum_aux acc (cons xs x) = sum_aux xs (plus x
                                      zero)
```

6/12/19

```
addFive : Lrst N -> Lrst N
addFive Empty = Empty
addFive Cons(x, xs) = Cons (x+5) (allFive xs)

isSeven : Lrst N -> Lrst B
isSeven Empty = Empty
isSeven Cons(x xs) = cons (x == 7) (isSeven xs)
sqrAll : Lrst N -> Lrsd N
sqr All Empty = Empty
sqrAll (con x xs) = Cons(pow x 2) sqrAll xs)

sqrAll : Lrst N -> Lrst N
sqrAll Empty = Empty

sqrAll Cons(x, xs) = coris (mult(x, x), xs)
```

```
map : (a -> b) -> List a -> List b
map _ Empty = Empty
map f (Cons x xs) = Cons (f x) (map f xs)

sqrAll : List N -> List N
sqrAll = map (λx. pow x 2)   or map ((flip pow) 2)
```

↑ function that take argument x

if map (pow 2), it's 2^x

```
isSeven : List N -> List B
isSeven = map (==7)

(==) : N -> N -> B   like map : (N -> B) -> List N -> List B

addFive : List N -> List N
addFive = map (plus 5)

flip : (a -> b -> c) -> (b -> a -> c)
flip f y x = f x y
```

```
fold : (a -> b -> b) -> b -> List A -> b
fold f acc Empty = acc
fold f acc (Cons x xs) = fold f (f x acc) xs
```

f : a -> b -> b, acc : b, x : a, xs : List a

```
sum : List N -> N
sum xs = fold plus 0 xs

sum [1,2,3]
↳ fold plus 0 [1, 2, 3]
    f  a   List xs
  fold plus 1 [2, 3]
  fold plus 3 [3]
  fold plus 6 [ ]
  ≡ 6
```

```
map :: (a -> b) -> List a -> List b
map f xs = fold (\x xs. (Cons (f x) xs)) [] xs
           a    acc
```

6/17/19

```
if (x)                    x match {
  Then y                    case True => y
  else z                    case False => z
                          }
```

```
ifelse : B -> A -> B -> Either A B
ifelse True a b = Left a
ifelse False a b = Right b
```

```
filter_may : (A -> B) -> Maybe A -> Maybe A
filter_maybe p Nothing = Nothing
filter_maybe p (Just x) = p x match
                 A -> B   A
True  -> Just x
False -> Nothing
```

learn for exam

symbols

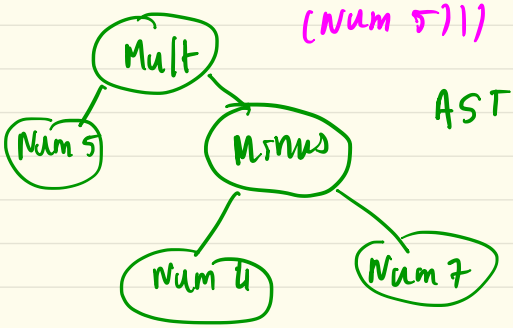
(,), ^, $Base_{10}$

x, -, x

(15 + 4) x 7) - 4

expressions

5 x (4 - 7) = Mult (Num 5) (Minus (Num 4) (Num 7))



AST

Expr ::= Num $\mathbb{Z}$
| Plus Expr Expr
| Mult Expr Expr
| Minus Expr Expr
| Pow Expr $\mathbb{N}$

($\mathbb{Z}$ has negative numbers
↳ interger

Val ::= NumVal $\mathbb{Z}$

6/18/19

eval : Expr → Val

↑ interpreter will take type Expre and return Val

Big-step operational semantics

Inference Rules

$$\frac{\text{Premise}}{\text{conclusion}}\ (\text{name})$$

eval ($\Downarrow$)

$E \Downarrow V$ (evaluate of value V)

arithetic semantic

$$\frac{}{\text{num } n \Downarrow n}\ (\text{eval-num})$$

$$\frac{E_1 \Downarrow V_1 \quad E_2 \Downarrow V_2}{\text{plus } E_1 E_2 \Downarrow V_1 + V_2}\ (\text{eval-plus})$$

$$\frac{E_1 \Downarrow V_1 \quad E_2 \Downarrow V_2}{\text{minus } E_1 E_2 \Downarrow V_1 - V_2}\ (\text{eval-minus})$$

$$\frac{E_1 \Downarrow V_1 \quad E_2 \Downarrow V_2}{\text{mult } E_1 E_2 \Downarrow V_1 * V_2}\ (\text{eval-mult})$$

$$\frac{E \Downarrow V}{\text{pow } E\ n \Downarrow V^n}\ (\text{eval-pow})$$

Plus (Num 5 , Num 2)

$$\frac{\dfrac{}{\text{Num } 5 \Downarrow 5}\,(\text{eval}_{\text{num}}) \qquad \dfrac{}{\text{Num } 2 \Downarrow 2}\,(\text{eval\_num})}{\text{Plus}(\text{Num } 5, \text{Num } 2) \Downarrow 7}\,(\text{eval-Plus})$$

eval : Expr → $\mathbb{Z}$

eval (Num n) = n

eval (Plus n, m) = Let v_1 = eval n
v_2 = eval m in $v_1 + v_2$

eval (Minus n, m) = Let v_1 = eval n
v_2 = eval m in $v_1 - v_2$

eval (Mult n, m) = Let v_1 = eval n
v_2 = eval m in $v_1 \times v_2$

eval (Pow n, m) = Let v_1 = eval n in pow(v_1, m)

---

eval_bin : ($\mathbb{Z} \to \mathbb{Z} \to \mathbb{Z}$) → Expr → Expr → $\mathbb{Z}$

eval_bin f E_1 E_2 = Let v_1 = eval(E_1)
v_2 = eval(E_2) in f v_1 v_2

eval : Expr → ℤ (use function eval_bin)

eval (Num n) = n

eval (plus n, m) = eval_bin plus n m

eval (Minus n, m) = eval_bin Minus n, m

eval (Mult n, m) = eval_bin Mult n, m

eval (Pow n, m) = let v = eval(n) in pow(v, m)

6/19/19

map_maybe : (A → B) → Maybe A → Maybe B

map_maybe Nothing = Nothing

map_maybe Just(m) f = Just(f(m))

Letfuce : the let language

let <symbol name> = <defining expressions>
in <body expressions>

① let x = 3 in x + x

② let x = 5 in
  let y = 10 in
    x + (2 * y)

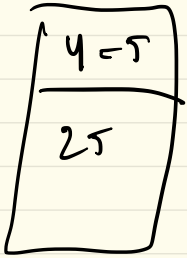
③ let x = (let y = 5 in y * y)
  in let z = 3 * x
  in y - z

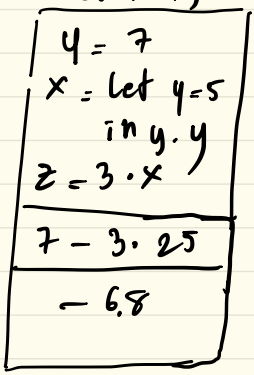
let y = 7

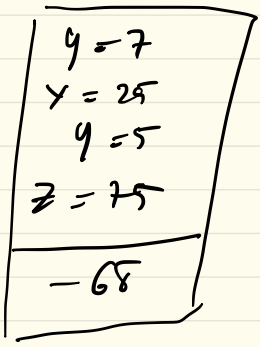
x = 5 * 5

z = 15 * 25

25 - 375 = -350







6/24/19

```
let x = 25 in
    let y = pow(x, 4) in
        let z = (x >= y) in
            if z
                then y
                else x
```

Program ::= Toplevel(Expr)

Expr ::= Ident(string)
| Const(ℤ)
| Pow(Expr, Expr)
| GTE(Expr, Expr)
| Plus(Expr, Expr)
| Minus(Expr, Expr)
| Mult(Expr, Expr)
| Bin(𝔹)

And (Expr, Expr)

Or (Expr, Expr)

Not (Expr)

Eq (Expr, Expr)

IfThenElse(Expr, Expr, Expr)

Let (string, Expr, Expr)

Const(5) : Expr

5 : $\mathbb{Z}$

what is concrete syntax?

Let ("x", const 25, ...)

Let x = 25 in ...

5 + true

plus (const 5, Bin true)

5 : $\mathbb{Z}$ (interger)

+ : $\mathbb{Z} \to \mathbb{Z} \to \mathbb{Z}$

let's call type checking

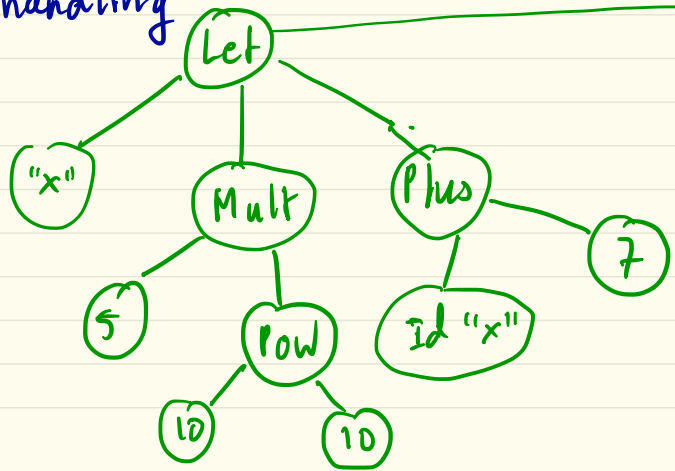
let x = x + y
in y + x

} nothing define

6/25/19

`let x = x + y in y + x`	`let x = 5.10^10 in x + 7`
ill-formed expression	well formed expression

- if computable
- every var is defined ✓
- syntactic closure ✓ dealing with parsing
- well-type ✓
- it terminates → decidability ✓
- resource handling



If we want to check if well-formed, run through the whole thing, check type and make sure everything define

WF - well Formed

$$\frac{}{WF(const(t), S)}\ (Const\text{-}WF)$$

$$\frac{x \in S}{WF(Ident(x), S)}\ (Ident\text{-}WF)$$

$$\frac{WF(e_1, S) \quad WF(e_2, S \cup \{x\})}{WF(Let(x, e_1, e_2), S)}\ (let\text{-}WF)$$

add x to S and e_2 be well formed under S

e_1: define expr

e_2: body expr

$$\frac{WF(e_1, S)}{WF(Neg(e_1), S)}\ (Neg\text{-}WF)$$

$$\frac{WF(e_1, s) \quad WF(e_2, s) \quad T \in \{\text{Plus}, \text{Minus}, \text{Mult}, \text{Pow}, \text{GEq}, \text{Eq}, \text{And}, \text{Or}\}}{WF(T(e_1, e_2), s)} \text{ (bin\_op\_WF)}$$

If e_1 is well formed under s

If e_2 is well formed under s

If T in the set of $\{\ \}$, we have $T(e_1, e_2), s)$

let $x = 5$ in $x + x$ $\}$ $\text{Let}(x, 5, \text{Plus}(\text{Id}(x), \text{Id}(x))$

6/26/19

```
foo ( x , y ) : ℕ = [
    val p = x + y
    p * p * (p + 1)
]
```

mean that

Let p = x + y in [ ]

sigma

$\sigma \vdash e \Downarrow v$ "In the context/environment σ, Expression e evaluate to value v"

σ eval env. $\sigma[x \to v]$

$\sigma ::= \emptyset \mid \sigma[x \mapsto v]$

$Val ::= NumVal(\mathbb{Z}) \mid BinVal(\mathbb{B}) \mid Error$

Let x = 5 in

  Let y = 6

  in x + y * y - x

$$\frac{}{\sigma \vdash \text{Const}(n) \Downarrow \text{NumVal}(n)} \; (\Downarrow\text{-const})$$

6/27/19

$$\frac{}{\sigma \vdash \text{Bin}(b) \Downarrow \text{BinVal}(b)} \; (\Downarrow\text{-Bin})$$

$$\frac{\sigma \vdash e_1 \Downarrow \text{NumVal}(v_1) \quad \sigma \vdash e_2 \Downarrow \text{NumVal}(v_2)}{\sigma \vdash \text{Plus}(e_1, e_2) \Downarrow \text{NumVal}(v_1 + v_2)} \; (\Downarrow\text{-Plus-Ok})$$

$$\frac{\sigma \vdash \text{Const}(3) \Downarrow \text{NumVal}(3) \quad \sigma \vdash \text{Const}(4) \Downarrow \text{NumVal}(4)}{\sigma \vdash \text{Plus}(\text{Const}(3), \text{Const}(4)) \Downarrow \text{NumVal}(7)}$$

$$\frac{\sigma \vdash e_1 \Downarrow \text{NumVal}(v_1) \quad \sigma \vdash e_2 \Downarrow \text{NumVal}(v_2) \quad T \in \{\text{Plus}, \text{Minus}, \text{Mult}\}}{\sigma \vdash T(e_1, e_2) \Downarrow \text{NumVal}(f(v_1, v_2))} \; (\Downarrow\text{-binop-arith-Ok})$$

where $f_+ \; x \; y = x + y$

$f_- \; x \; y = x - y$

$f_* \; x \; y = x * y$

$$\frac{\sigma \vdash e_1 \Downarrow v_1 \quad \sigma \vdash e_2 \Downarrow v_2 \quad \begin{array}{l} v_1 \neq \text{NumVal}(x) \\ v_2 \neq \text{NumVal}(y) \end{array} \text{ or,}}{\sigma \vdash T(e_1, e_2) \Downarrow \text{Error}}$$

(⇓-binop_arrith-nok)

$$\frac{x \in \sigma}{\sigma \vdash \text{Ident}(x) \Downarrow \sigma(x)} (\Downarrow\text{-Ident})$$

$$\frac{x \notin \sigma}{\sigma \vdash \text{Ident}(x) \Downarrow \text{error}} (\Downarrow\text{-Ident\_nok})$$

$$\frac{\sigma \vdash def \Downarrow v_1 \quad v_1 \neq \text{error} \quad \sigma[id \mapsto v_1] \vdash body \Downarrow v_2}{\sigma \vdash \text{Let}(id, def, body) \Downarrow v_2} (\Downarrow\text{-Let})$$

$$\frac{\sigma \vdash def \Downarrow v_1 \quad v_1 = \text{Error}}{\sigma \vdash \text{Let}(id, def, body) \Downarrow \text{Error}} (\Downarrow\text{-Let\_shucks.})$$

07/01/19

$$\frac{\sigma \vdash e_1 \Downarrow v_1,\ v_1 \in \mathbb{B} \qquad \sigma \vdash e_2 \Downarrow v_2,\ v_2 \in \mathbb{B} \qquad p \in \{\text{And}, \text{Or}\}}{\sigma \vdash p(e_1, e_2) \Downarrow \text{BoolVal}(g_p(v_1, v_2))} \ (\Downarrow\text{-bool-brnop-ok})$$

$$g_\wedge\ x\ y = x \wedge y$$

$$g_\vee\ x\ y = x \vee y$$

$$\frac{\sigma \vdash e_1 \Downarrow v_1 \quad v_1 \in \mathbb{B} \qquad \sigma \vdash e_2 \Downarrow v_2 \quad v_2 \in \mathbb{B}}{\sigma \vdash Eq(e_1, e_2) \Downarrow \neg(v_1 \oplus v_2)} \ (\Downarrow\text{-eq-bool})$$

$$\frac{\sigma \vdash e_1 \Downarrow v_1 \quad \sigma \vdash e_2 \Downarrow v_2 \quad v_1, v_2 \in \mathbb{Z}}{\sigma \vdash Eq(e_1, e_2) \Downarrow v_1 = v_2} \ (\Downarrow\text{-eq-int})$$

$$\frac{\sigma \vdash e_1 \Downarrow c_1(v_1) \qquad \sigma \vdash e_2 \Downarrow c_2(v_2) \qquad c_1 \neq c_2}{\sigma \vdash Eq(e_1, e_2) \Downarrow \text{Error}} \ (\Downarrow\text{-eq-nok})$$

$$\frac{\sigma \vdash p \Downarrow \text{True} \qquad \sigma \vdash e_1 \Downarrow v}{\sigma \vdash \text{IfThenElse}(p, e_1, e_2) \Downarrow v} \ (\Downarrow \text{IfThenElse-True})$$

For exam -> what is Concrete syntax
what is abstract?

---

let square = function(x)
x . x
in square(10)

concrete = what we write in program

Abstract = tree base language tool

let("square", FunDef("x", Mult(Ident "x", Ident "x"))
Funcall(Ident("square"), Const(10))

square(10)
x = 10
10 . 10
100

```
let f = function (n)
          if n > 5
            g
          else n
in f (7) (True)
```

07/02/19

closure("x", Mult [Ident "x", Ident "x"), ∅)

ex 1 `let square = function(x) x.x in square(10)`

ex 2
```
let x = 10 in
let y = 15 in
let sq1 = function(x)
            function(y)
              x + y.y
in sq1(y)(x)
```

$\sigma = [\text{square} = \text{Closure}(\text{"x"}, \text{Mult}(\text{Ident "x"}, \text{Ident "x"}), \emptyset)]$

$\emptyset\,[x \to 10] \vdash \text{Mult}(10, 10) \Downarrow 100$

Syntax

Expr ::= ...

| FunDef(string, Expr)

| FunCall(Expr, Expr)

semantic : rule, what is your syster does

↳ meaning of programming

Value ::= ...

| closure(string, Expr, Dictionar)

Parameter Expr Environment

Ex 2

closure("x", Fundef("y", x+y·y), $\begin{bmatrix} x \mapsto 10 \\ y \mapsto 15 \end{bmatrix}$)

$$\frac{}{\sigma \vdash \text{Fundef}(x, e) \Downarrow \text{Closure}(x, e, \sigma)} \; (\Downarrow\text{-FunDef})$$

$$\frac{\sigma \vdash e_2 \Downarrow v \qquad v \neq \text{Error} \qquad \sigma \vdash e_1 \Downarrow \text{closure}(p, e_c, \pi) \qquad \pi[p \mapsto v] \vdash e_c \Downarrow v_f}{\sigma \vdash \text{FunCall}(e_1, e_2) \Downarrow v_f} \; (\Downarrow\text{-FunCall\_ok})$$

07/8/19

Exam II

- 20 question
- Multiple choice
- Rough_topic list
  - semantic (of lettuce)
  - Inference rule
  - Interprete
  - support for function (closure)

warm-up

what is type of this?

Const (S(S(S Z))) : Expr

nat

what type of this? Cons(5, Cons 2 Empty) : Lst ℕ

ℕ ℕ Lst ℕ

Let (x, y, z)

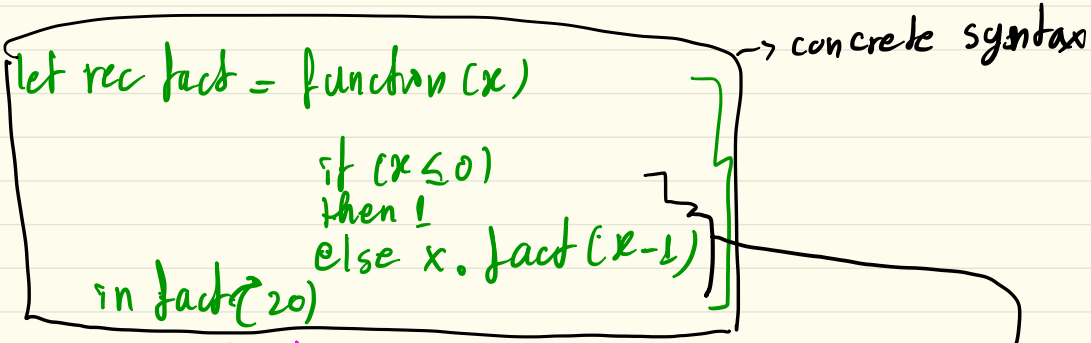
string Expr Expr

can be single character but it's strong

Plus (Const ∅, Const 5)

Expr Expr

```
let fact = function (x)
             if (x ≤ 0)
               then 1
             else  x . fact (x-1)
in fact (20)
```



let rec <function name> = function (<arg-name>)
< function body expression .

in < body of program >

Expr ::= ...

| LetRec(string, string, Expr, Expr)

Function name, argument, body of function, body of program

7/9/19

```
let rec fact = function (x)
                 if (x ≤ 0)
                  then 1
                  else x . fact(x-1)
    in fact(3)
```

fact (3-1). 3

fact (2). 3

fact (2-1). 2

fact (1). 2

fact (1-1). 1

fact (0). 1

1. 1. 2. 3

6

$\sigma' = [fact \mapsto closure("x", \text{if } x \le 0 \text{ then } 1 \text{ else } fact(x-1).x, \sigma')]$

$\sigma'' = \sigma'[x \mapsto 3]$

$\sigma''' = \sigma'[x \mapsto 2]$

$\sigma'''' = \sigma'[x \mapsto 1]$

$\sigma^5 = \sigma'[x \mapsto 0]$

$$\frac{\hat{\sigma} = \sigma[name \mapsto closure(arg, def, \hat{\sigma})] \quad \hat{\sigma} \vdash body \Downarrow V}{\sigma \vdash LetRec(name, arg, def, body) \Downarrow V} \; (\Downarrow\text{-LetRec})$$

Env ::= Empty Env
| Extend(strrng, Value, Env)
| ExtendRec(strrng, strrng, Expr, Env)
function name arg

Value ::= NumVal Z
| BrnVal B
| closure(arg, Expr, Env)
Fun Body
| Error
| Ref(N)

```
Letrec("f", "x",
    IfThenElse(
     Eq("x", one), one,
     Mult("x", FunCall("f", Minus("x", one)))
    ),
  FunCall("f", three)
```

Review for Exam II (07/10/19)

$$\frac{\sigma \vdash e_1 \Downarrow \text{BrnVal(true)} \quad \sigma \vdash e_2 \Downarrow v}{\sigma \vdash \text{ITE}(e_1, e_2, e_3) \Downarrow v} \; (\Downarrow \text{ITE\_true})$$

$$\frac{\sigma \vdash e_1 \Downarrow \text{BrnVal(False)} \quad \sigma \vdash e_3 \Downarrow v}{\sigma \vdash \text{ITE}(e_1, e_2, e_3) \Downarrow v} \; (\Downarrow \text{ITE\_False})$$

$$\frac{\sigma \vdash e_1 \Downarrow v \quad v \in \text{NumVal} \cup \text{Error}}{\sigma \vdash \text{ITE}(e_1, e_2, e_3) \Downarrow \text{Error}} \; (\Downarrow \text{ITE\_Nok})$$

$$\frac{\sigma \vdash e_1 \Downarrow v' \quad v' \neq \text{error} \quad \sigma[x \mapsto v'] \vdash e_2 \Downarrow v}{\sigma \vdash \text{Let } x = e_1 \text{ in } e_2 \Downarrow v} \; (\Downarrow\text{-Let})$$

$$\frac{\sigma \vdash e_1 \Downarrow \text{error}}{\sigma \vdash \text{Let } x = e_1 \text{ in } e_2 \Downarrow \text{error}} \; (\Downarrow\text{-Let-nok})$$

$$\frac{}{\sigma \vdash \text{FunDef}(x, e) \Downarrow \text{closure}(x, e, \sigma)}$$

$$\frac{\sigma \vdash e_1 \Downarrow \text{closure}(x, e_b, \sigma') \quad \sigma \vdash e_2 \Downarrow v_2 \quad v_2 \neq \text{error} \quad \sigma'[x \mapsto v_2] \vdash e_b \Downarrow v'}{\sigma \vdash \text{FunCall}(e_1, e_2) \Downarrow v'} \; (\text{FunCall}$$

If FunCall is not ok, when $e_1 \Downarrow$ error or $e_2 \Downarrow$ error

- operation semantic
- Inference rule
- concrete & Abstract semantic
- Environment/Dictionary
- syntax/semantic for lettuce
- WF Lettece Express
- Equality
- Process of Long Expr
- Function semantic
- well type Express

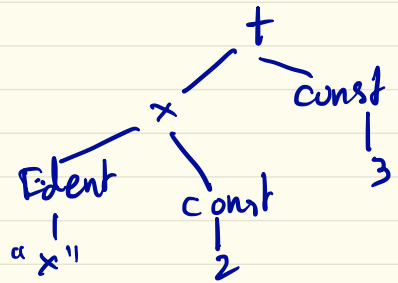
Plus s (Ident ("x"), Ident ("x")) .=> abstract syntax
structure of tree

x + y vs Concrete syntax

let x = e_1 in e_2 concrete syntax

LetExpr (Ident, Expre, Expre) AST

Concrete	Abstract
x * 2 + 3	Plus(Mult(Ident("x"), Const(2)), Const(3))



what is the type of bar

```
bar f x y = y match {
    case T => f(x)
    case F => succ(succ(x))
}
```

bar : Nat → Nat → Nat → Bool → Nat

```
let x = 5 in
    y = 6
in x + y * y - x
```

Let(x, 5, Let(y, 6, plus(x, Minus(Mult(y, y), x))))

Traits of well-Formed

-> computable

-> every var defined

-> syntactic closure

-> Type check

-> Termination

-> Resource Handing

iligal-formel

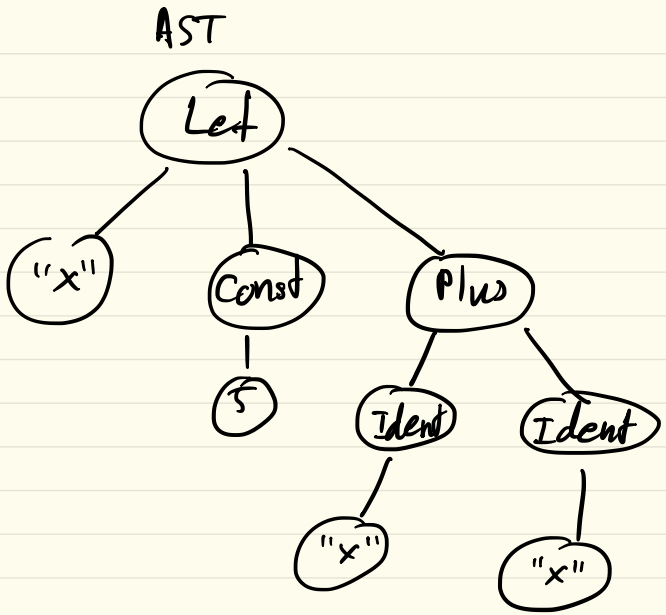
```
let x+y = x
in y + x
```

well-Formed

```
let x = 5 x10
in x+7
```

07/10/19

concrete: Let x = 5
in x + x



Abstract: Let("x", const(5), Plus(Iden("x"), Iden("x"))

concrete

Let x : int = 5
in x + x

Type Checking in Letfuee

- Identifiers are declare before use (WF)
- Arithmetic operator arguments eval to $\mathbb{Z}$.
- Equality - both args has same type
- Conditionls
  1) condition evaluates to $\mathbb{B}$
  2) then/else parts have same type
- Function calls
  1) arg to agree with input
  2) Function Id points to closure

5 + 5 : num

True ∧ False : bool

func(x) x+x : num -> num

Lettuce types

1. num_ things that eval to NumVal
2. bool_ things that eval to BoolVal
3. $T_1 \rightarrow T_2$ - things that eval to closure

Example: Const(10) : num

Bfn(T) : bool

Plus (Const(5), Const(10)) : num

07/15/19

Syntax of lettuce Types

```
Type ::= NumType
       | BoolType
       | FunType(Type, Type)
```

```
Let(x, BoolType, Const(5),
    x+x = Let x : bool = 5
          in x+x
```

→ it's well formed, but not typechecking

```
Expr ::= . . .
       | Let(string, Type, Expr, Expr)
       | FunDef(string, Type, Expr)
       | LetRec(string, Type, string, Type, Expr, Expr)
```

→ Just input type, not function type

Let(x, NumType, Const(5), x+x) = Let x : Num = 5
in x+x

Type for function Type for input

let f : num -> num = function(x : num)
x + x

in f(5)

let rec fac : num -> num = function(x : num)

Def { if x ≥ 1
then x * fac(x - 1)
else 1

in fac(3) } body)

LetRec(fac, FunType(NumType, NumType), x, NumType, Def, body)

gamma

Γ ::= ∅

| Γ [string ↦ Type]

Typechecking is semantic, what the meaning of Program

(TC - Const)

$$\frac{}{\Gamma \vdash \text{Const}(x) : \text{num}}$$

(TC - Bin)

$$\frac{}{\Gamma \vdash \text{Bin}(p) : \text{bool}}$$

07/16/19

TC-Arith

$$\frac{\Gamma \vdash e_1 : \text{num}, \Gamma' \vdash e_2 : \text{num}}{\Gamma \vdash T(e_1, e_2) : \text{num}}$$

where $T \in \{\text{Plus}, \text{Minus}, \text{Mult}, \text{Pow}\}$

TC_Arith_nok1

$$\frac{\Gamma \vdash e_1 : \tau, \quad \tau \neq \text{num}}{\Gamma \vdash T(e_1, e_2) : \text{type error}}$$

TC_Arith_nok2

$$\frac{\Gamma \vdash e_2 : \tau \quad \tau \neq \text{num}}{\Gamma \vdash T(e_1, e_2) : \text{type error}}$$

TC_Ident

$$\frac{x \in \Gamma \qquad \tau = \Gamma(x)}{\Gamma \vdash \mathrm{Ident}(x) : \tau}$$

TC_Let

$$\frac{\Gamma \vdash def : \tau_1 \quad , \quad \Gamma[x \mapsto \tau_1] \vdash body : \tau_2}{\Gamma \vdash \mathrm{Let}(x, \tau_1, def, body) : \tau_2}$$

TC_Eq $\qquad \tau_1 \neq \mathrm{FunType}$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_1 = \tau_2}{\Gamma \vdash \mathrm{Eq}(e_1, e_2) : \mathrm{bool}}$$

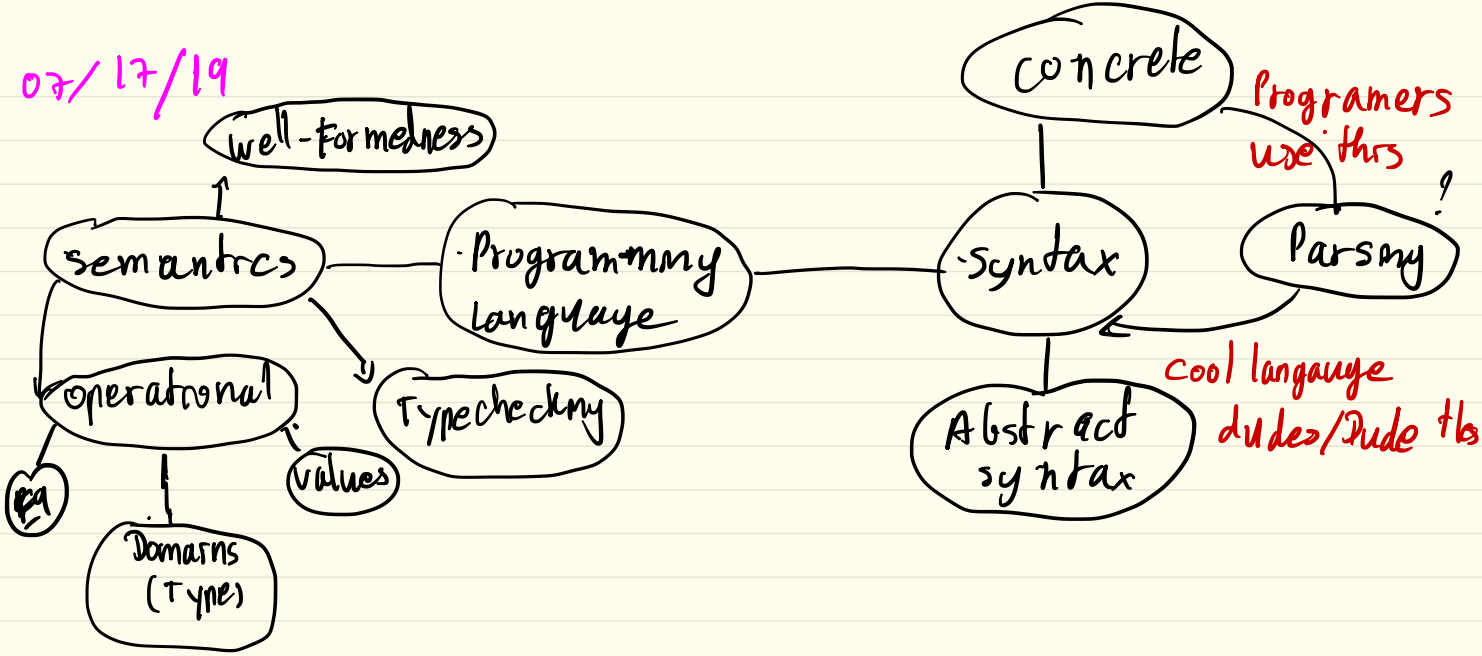
TC_FunDef

$$\frac{\Gamma[x \mapsto \tau_1] \vdash e_1 : \tau_2}{\Gamma \vdash \mathrm{FunDef}(x, \tau_1, e_1) : \tau_1 \to \tau_2}$$

TC_FunCall

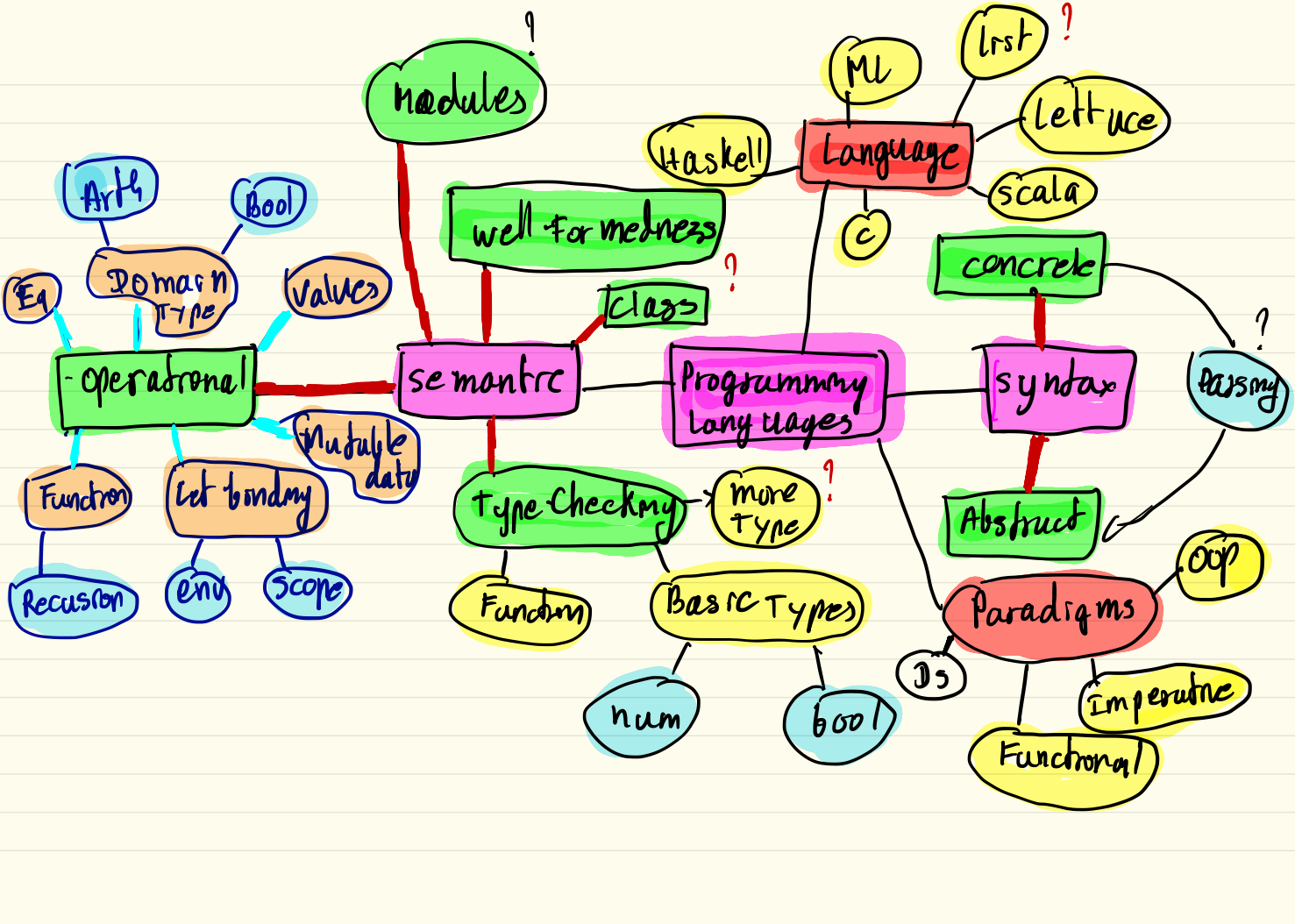
$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash \mathrm{FunCall}(e_1, e_2) : \tau_2}$$

07/17/19



→ when we take concrete syntax into abstract syntax, we call parsing.

→ a single AST can write different concrete.


modules
?
Arth
Bool
Eq
Domain Type
Values
Operational
Mutable data
Function
Let binding
Recursion
env
Scope
well Formedness
?
Class
semantic
Type Checking
more Type
?
Function
Basic Types
num
bool
ML
lrst
?
Haskell
Language
Lettuce
scala
c
concrete
?
Parsing
syntax
Abstract
Programming Languages
Paradigms
OOP
Js
Imperative
Functional

Project = Parser for leftlure

- write you a lisp
- DSL
- Type : Leftuce
- Algebraic Datatypes in leftlure

07/22/19

① Parsing : Concrete → AST

```
let x = 5
in x + x
```
} Let("x", Const(5), Plus(Ident"x", Ident"x"

② write you a lisp : (λx. x + x)(5)

(defun f(x) (plus x x))

(x(y(z)

(x y z)

③ Algebraic Datatype in lecture (most difficult)

let type color : : = Red | Blue | Green

in Red

④ Domain specific language (DSL)

```
var x = 5
x++

let x = 5

in x ++
```

1. Do I understand the feature/problem?
   a) More examples!
   b) Is it possible!
   c) why? Does something else solve the problem?
2. what is the syntax! (concrete, Abstract)

a) Is better way to write this?

③ what are the rules/semantics of our language?

a) How do I process this language?

b) do any of these rules on top of a simpler systems?

c) Is there an already defined structures or Algorithm that describes this process?

d) what are the rules! (Inference)

④ Implement

⑤ looking back

ⓐ test it, does this do what I want it to do?

ⓑ are there more features that could be added?

07/23/19

Mutuability

```
let x = new ref(10) in
let y = x + 1 in
let z = x := y in
z
```

store

σ

$x \mapsto ref(0)$

$y \mapsto 11$

```
let x = new ref (true)
  in let y = new ref (false)
  in let z = x
    in !z
```

store

$id_0 = True$

$id_1 = False$

σ

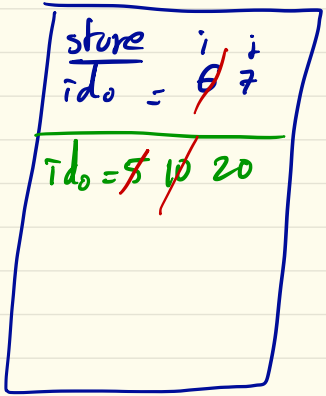
$x \mapsto ref(0)$

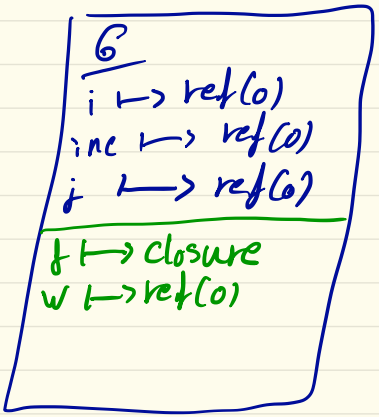
$y \mapsto ref(1)$

$z \mapsto ref(0)$

Output: $z \mapsto ref(0)$

```
let inc = function(x)
        x := *x + 1
  in let i = new ref(5)
  in let j = inc(i)
  in let k = inc(i)
    in *i
```






```
let f = function(x)
      let y = *x
    in x := y + y
  in let w = new ref(2+3) in
  [ f(w) ]
```

output: 10

```
let f = function(x)
      let y = *x
    in x := y + y
  in let w = new ref(2+3)
  in [ f(w); f(w); *w ]
```

output: 20

7/24/19

```
let inc = function (x)
      x := *x + 1
in let i = new ref(2)
   in [ inc (i)
        inc (i)
        inc (i)
        i*      ]
```

increment i three times

2+1 = 3
3+1 = 4
4+1 = 5

store
$id_0 = 2/3/4/5$ σ
inc $\mapsto$ closure(x, x := *x + 1, σ)
i $\mapsto$ ref (0)

* – de-reference

new ref(Expr) – creates a new ref in store

:= – assign reference

[... j .. j ..] – sequence

```
Expr ::= ...
       | NewRef(Expr)
       | DeRef(Expr)
       | AssignRef(Expr, Expr)
       | Seq(Lst, Expr, Expr)

Val ::= ...
      | Ref(N)

store ::= Dict N Val
```

7/24/19. Exam III review

- let rec [defined / interpreted
- Inferenee rule
  - Type checking
  - Static
  - Dynamic

, Extend rec.

let rec f x = e in e' ≡ LetRec(f, x, e, e')

$$\frac{\hat{\sigma} = \sigma[f \mapsto \text{closure}(x, e, \hat{\sigma})] \quad \hat{\sigma} \vdash e' \Downarrow v}{\sigma \vdash \text{let rec } f\ x = e \text{ in } e' \Downarrow}$$

f: name of function
x: name of arguement

```
let rec f x = e in
    let y = 5 in
        f(5)
```

Type checking

$$\frac{T' = T[f \mapsto x \mapsto T' \mapsto \psi''] \quad T' \vdash e' : \psi \quad T[\quad}{T \vdash \text{letRec}(f, x, e, e') : \psi}$$

Not on exam

$\sigma = \epsilon[fac \mapsto clo(x, e, \sigma)]$ (C) $\quad \sigma \vdash fac \Downarrow C$, $\sigma \vdash 1 \Downarrow 1$ $\quad \sigma \vdash fac\ 1 \Downarrow 1$

$\epsilon \vdash$ let rec fac x = x > 0 ? x, fac(x - 1) : 1 in fac(1) $\Downarrow$

(e) (function application)

Dynamic Typechecking = duck typechecking , run time

↳ running programming , execute, counter they not match up

static typechecking = not running program

= complile time , catch error quickly if's error

7/29/19

"ref x = e
⇓ -> sugar
let x = ref e

Expr e ::= ... | x | $e_1 + e_2$ ....

"new ref" / ref e		NewRef(Expr)
"deref	/ *e	DeRef(Expr)
assign	/ $e_1 := e_2$	AssignRef(Expr, Expr)
sequencing	/ $e_{1};\ldots;e_n$	sequence(list[Expr], Expr)

$\sigma \vdash e \Downarrow v$
pure

$h, \sigma \vdash e \Downarrow (v, h')$
effects
↳ heap/store

value v ::= ... | Ref(N)
pointer

Heap/store h ::= Dict[N, Val]
malloc : Heap => N

$\sigma[x \mapsto v]$ is Extend(string, value, Dict)

$$\frac{h, \sigma \vdash e \Downarrow (v, h') \quad n = \mathrm{malloc}(h')}{h, \sigma \vdash \mathrm{newref}\ e \Downarrow (\mathrm{Refself}(n), h'[n \mapsto v])} \text{ (new ref)}$$

or $(\mathrm{Ref}(n), h')$

$$\frac{h, \sigma \vdash e \Downarrow (v, h') \quad v = \mathrm{ref}(n)}{h, \sigma \vdash *e \Downarrow \langle h'(n), h' \rangle} \text{ (deref)}$$

Example of new ref

$$\emptyset, \emptyset \vdash \mathrm{ref}(1+1) \Downarrow \langle 2, h'[0 \mapsto 2] \rangle$$

$$\frac{\begin{array}{l} h; \sigma \vdash e_r \Downarrow (v, h') \\ h', \sigma \vdash e_l \Downarrow (\mathrm{Ref}(n), h'') \end{array}}{h, \sigma \vdash e_l := e_r \Downarrow (v, h''[n \mapsto v])} \text{ (assign)}$$

$$\frac{\forall i \in 1..n \quad h_{i-1}, \sigma \vdash e_i \Downarrow (v_i, h_i) \quad h_n, \sigma \vdash e \Downarrow (v, h')}{h, \sigma \vdash e_1; \ldots; e_n; e \Downarrow (v, h')} \text{ (seq)}$$

7/30/19

λ-abstractions

$f(x) = x + 5$ $\quad$ $f = \lambda x. x + 5$

$g(y) = 2 \cdot y + 7$ $\quad$ $g = \lambda y. 2y + 7$

→ λ is fundamental of computer science

$(\lambda x. x + 5)\ 5$

$5 + 5$

10

variable (x, y) ::= strings

Expressions (e) ::= $\lambda x. e$

| $e_1\ e_2$

| x

$\lambda x.x$ (x: bound) $= Ident$

$(\lambda x.\lambda y.x)(z)$

$(\lambda y.z)$

$\lambda x.y$ (y: Free) → y is not bound to λ

$(\lambda x.\lambda y.z)$ (z: Free)

α-renaming

$\lambda x.e =_\alpha \lambda y.[x := y]e$ if y is not free in e

β-reduction

$(\lambda x.e_2)e_1 =_\beta [x := e_1]e_2$

$(f \circ g) = \lambda x.f(g(x))$

$\circ = \lambda f.\lambda g.\lambda x.f(g(x))$

$I = \lambda x.x$

$\circ I\ f = (\lambda f.\lambda g.\lambda x.f(g(x)))(\lambda x.x)\ f$

$= \beta(\lambda g.\lambda x.((\lambda x.x)(g(x))))\ f$

$= \beta\ [\lambda x\ ((\lambda x.\ x)\ (f\ x))]$

$= \beta\ \lambda x.\ f\ x \quad = f$

η-conversion

$e = \eta\ \lambda x.\ e\ x$ if x is not free in e

} If I am not using x, remove x out

7/31/19

```
let inc = λx. x := !x + 1
in
    let i := newref 2
    in [ inc(i);
         inc(i);
         inc(i);
       } !i
```

- mutation
- effect side
- new left value
- syntax sugar

$$\frac{h, G \vdash e \Downarrow (v, h') \quad n = \text{malloc}(h')}{h, G \vdash \text{newref } e \Downarrow (\text{Ref}(n), h'[n \mapsto v])}$$

newref e

- "x += e" — concrete syntax
  ⇓
  x := x + e — AST
  ↳ sugar syntax

- e ; e' → sequence
  ⇓ sugar syntax
  let _ = e in e'