(10 points) Consider the grammar G:

$$\begin{aligned} S &::= xS \mid T \\ T &::= yT \mid U \\ U &::= zU \mid \epsilon \end{aligned}$$

Where x,y and z are the terminals, T, S and U are the non-terminals, and S is the root symbol.

1. Define $L(G)$, that is, the language generated by grammar G

any # of x's, y's, of z's but don't have to be the same # so use *

$L(G) = \{ x^* y^* z^* \}$

2. Is the grammar regular, context free or context sensitive? briefly justify your answer.

Regular: format of terminal followed by a nonterminal

S ::= xS | T

but if it was in a format such as NT.T.NT like aGb | ϵ that would be context free.

(7 points) Give an example of an ambiguous grammar and show why the grammar is ambiguous

ambiguous if you can derive the same sentence via two different paths

ex: (G) ::= ØGG1 | ØG1 | ϵ

path 1: G → 0GG1 → 00G11 → 0011

path 2: G → 0G1 → 00G11 → 0011

1. Rewrite the grammar for arithmetic expressions

$$E ::= a \mid b \mid E + E$$

to express the fact that addition is right-associative.

to be right assosiative, the grammar has to have recursion on the right like:

E = T+E | T

T = a | b | (E)

to be left assosiative, the grammar has to have recursion on the left like:

E = E+T | T

T = a | b | (E)

2. Give the parse tree and abstract syntax tree for the expression $a + b + a + b$ with respect to the grammar you have defined above

RIGHT ASSOCIATIVE

a + b + a + b

a + b + x

a + y

e

LEFT ASSOCIATIVE

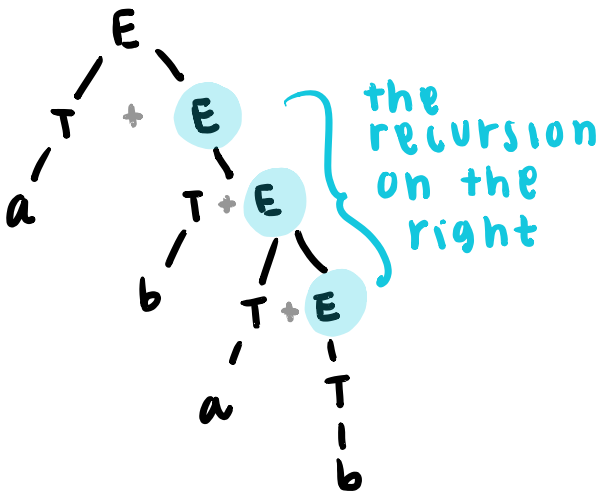
a + b + a + b

x + a + b

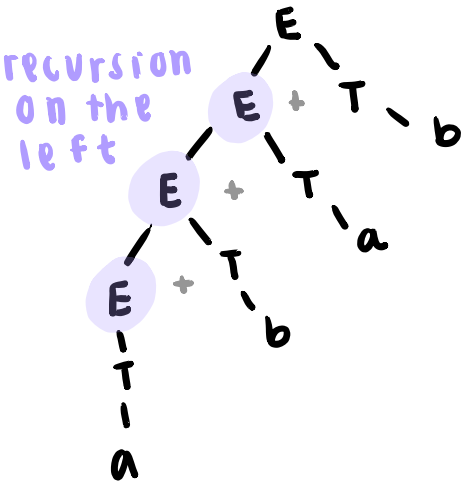
y + b

e

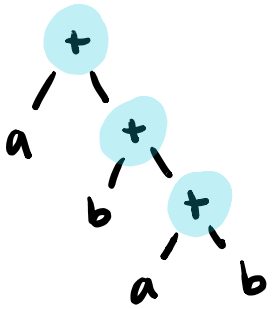
parse tree



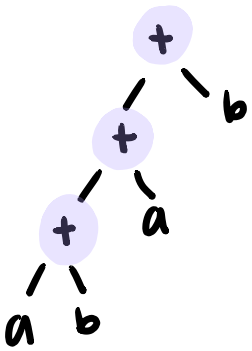
parse tree



Abstract tree



Abstract tree



⟨*assignment*⟩ ::= ‘`a = 1`’ | ‘`a = 2`’

⟨*stmt*⟩ ::= ⟨*assignment*⟩ | ⟨*if-stmt*⟩

⟨*expr*⟩ ::= ‘`x > y`’ | ‘`x < z`’

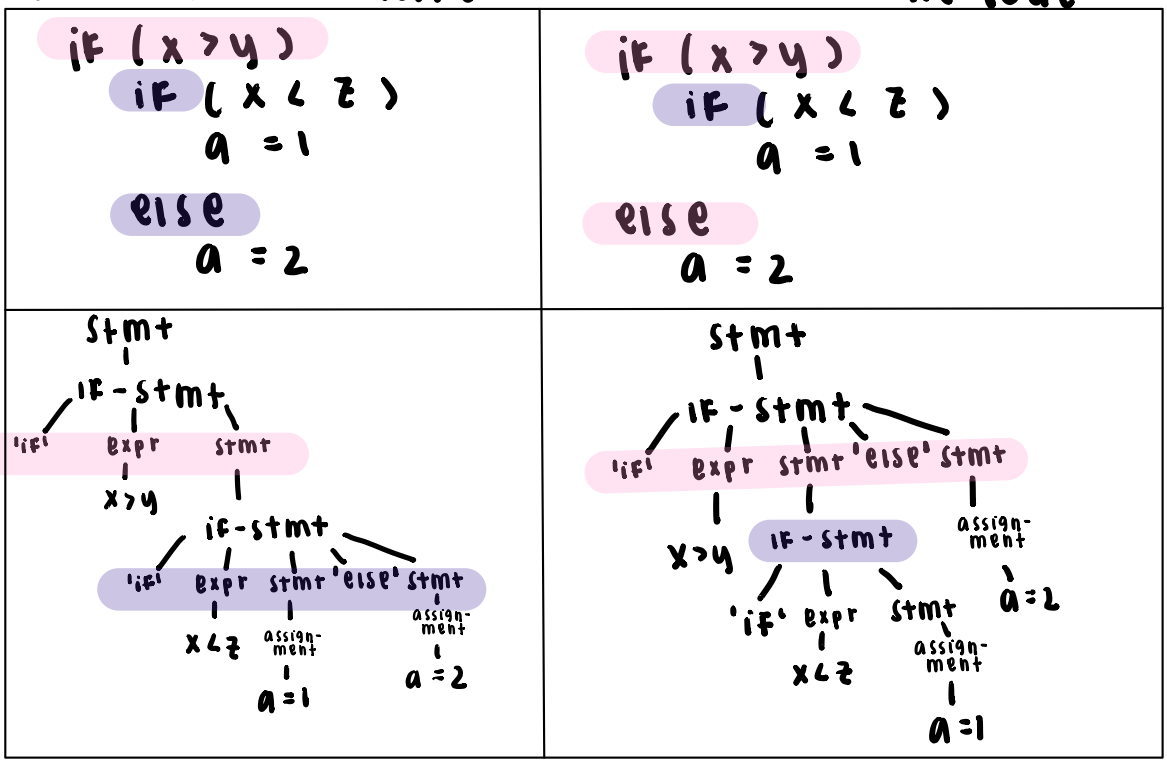
⟨*if-stmt*⟩ ::= ‘`if`’ (⟨*expr*⟩) ⟨*stmt*⟩ | ‘`if`’ ‘`(`’⟨*expr*⟩ ‘`)`’ ⟨*stmt*⟩ ‘`else`’ ⟨*stmt*⟩

Note that <> are used to denote nonterminals

Draw two parse trees for the following program fragment:

if $(x > y)$ if $(x < z)$ a $= 1$ else a $= 2$

Two different interpretations of the code:



5. Draw a parse tree for the following strings:

   (a) $1 * 2 - (3/4)$
   (b) $1 - 2 + 3$
   (c) $1 * 2 - 3/4$

6. Draw an abstract syntax tree for the following strings:

   (a) $1 * 2 - (3/4)$
   (b) $1 - 2 + 3$
   (c) $1 * 2 - 3/4$

provided grammar:

(E) ::= E + T | E - T | T

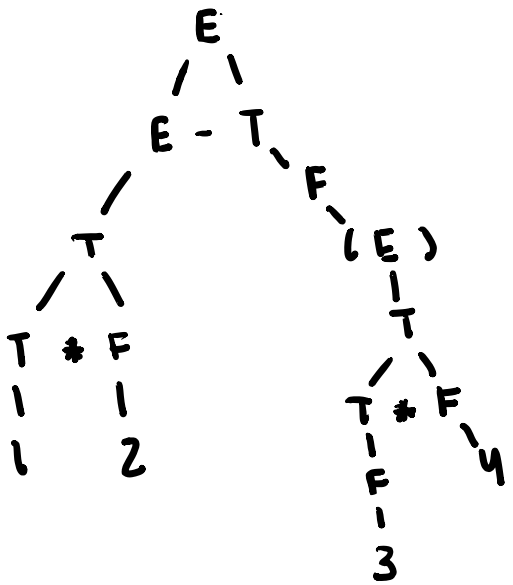
(T) ::= T * F | T / F | F

(F) ::= 1 | 2 | 3 | 4 | (E)

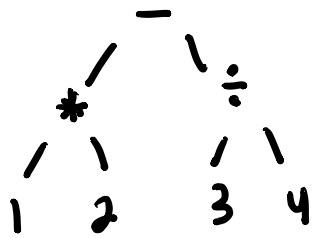
LEFT ASSOSIATIVE

1 * 2 - (3/4)

parse tree

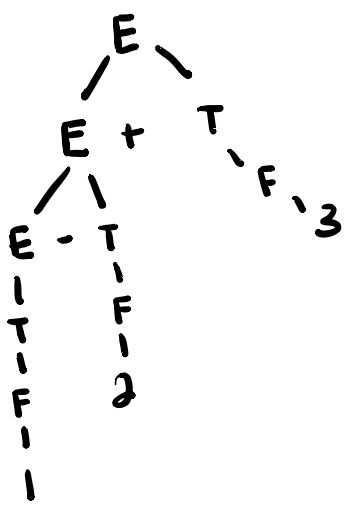


ABstract tree

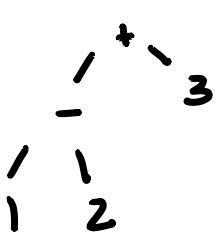


1 - 2 + 3

parse tree

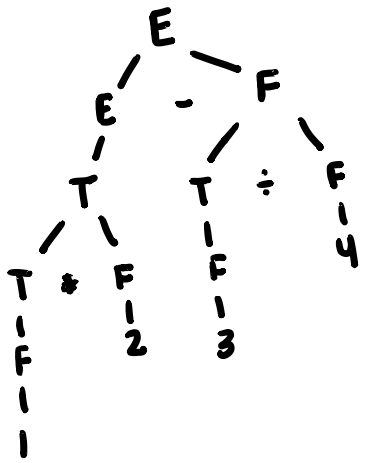


ABstract tree

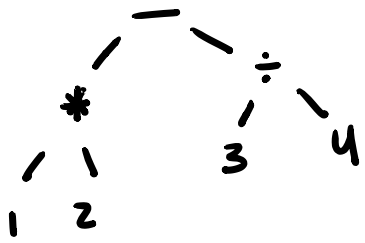


1 * 2 - 3 / 4

Parse tree

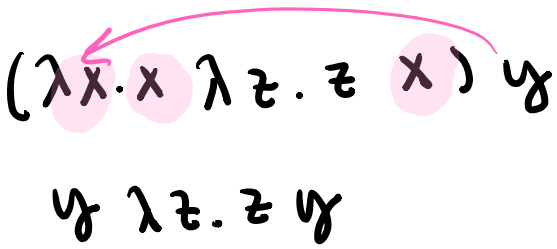


Abstract tree



(5 points) Apply β-reduction to the following term

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



y λz.z y

- $\lambda x.\ (y\ z)$ and $\lambda x.\ y\ z$ are equivalent
  1. *True*
  2. *False*

- The term $\lambda x.x\ a\ b$ is equivalent to which of the following?
  1. $(\lambda x.x)\ (a\ b)$
  2. $(((\lambda x.x)\ a)\ b)$
  3. $\lambda x.(x\ (a\ b))$
  4. $(\lambda x.((x\ a)\ b))$

$(\lambda x.((x\ a)\ b))$

- $(\lambda x.y)z$ can be beta-reduced to
  1. y
  2. $y\ z$
  3. z
  4. *Cannot be reduced*

$(\lambda x.y)\ z$

y

- Which of the following reduces to $\lambda z.z$
  1. $(\lambda y.\lambda z.x)\ z$ → $(\lambda y.\lambda b.x)\ z = \lambda b.x$
  2. $(\lambda z.\lambda x.z)\ y$ → $\lambda x.y$
  3. $(\lambda y.y)\ (\lambda x.\lambda z.z)\ w$ → $(\lambda y.y)\ (\lambda z.z)$ → $\lambda z.z$ ✓
  4. $(\lambda y.\lambda x.z)\ z\ (\lambda z.z)$

- Which of the following expressions is α-*equivalent* to (α-converts from) $(\lambda x.\lambda y.x\ y)y$
  1. $\lambda y.y\ y$
  2. $\lambda z.y\ z$ beta reduced ans.
  3. $(\lambda x.\lambda z.x\ z)\ y$ alpha reduced
  4. $(\lambda x.\lambda y.x\ y)\ z$

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

$= (\lambda x.\lambda b.x\ b)\ y$

- β-reducing the following term produces what result?

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

  1. $y\ (\lambda z.z\ y)$
  2. $z\ (\lambda y.y\ z)$
  3. $y\ (\lambda y.y\ y)$
  4. $y\ y$

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

$= (\lambda x.x\ \lambda b.b\ x)\ y$

$= y\ \lambda b.b\ y$

- β-reducing the following term produces what result?

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

  1. $\lambda x.w\ w\ z$
  2. $\lambda x.w\ z$
  3. $w\ z$
  4. *Does not reduce*

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

$\lambda x.w\ w\ z$

1. Make all parentheses explicit in the following $\lambda-$expressions

   (a) $\lambda x.x\ z\ \lambda y.x\ y$ = $(\lambda x.((x\,z)(\lambda y.x\,y)))$

   (b) $(\lambda x.x\ z)\ \lambda y.w\ \lambda w.w\ y\ z\ x$ $((\lambda x.x\,z)(\lambda y.w(\lambda w.(((w\,y)\,z)\,x)))$

   (c) $\lambda x.x\ y\ \lambda x.y\ x$ $(\lambda x.((x\,y)(\lambda x.y\,x)))$

2. Find all free (unbound) variables in the following $\lambda-$expressions

   (a) $\lambda x.x\ z\ \lambda y.x\ y$ $(\lambda x.((x\,z)(\lambda y.x\,y)))$ z is free

   (b) $(\lambda x.x\ z)\ \lambda y.w\ \lambda x.w\ y\ z\ x$ $(\lambda x.x\,z)(\lambda y.w(\lambda x.w\,y\,z\,x))$

   (c) $\lambda x.x\ y\ \lambda x.y\ x$ $(\lambda x.x\,y(\lambda b.y\,b))$

3. Give the result of performing the following substitutions:

   (a) $[(\lambda y.x\ y)/x](x\ (\lambda x.y\ x))$ Replace x w/ $(\lambda y.x\,y)$ $=(\lambda y.x\,y)(\lambda x.y\,x)$

   (b) $[(\lambda x.x\ y)/x](\lambda y.x\ (\lambda x.x))$ $=\lambda y.\ \lambda x.x\,y\ (\lambda x.x) \to \lambda y'.(\lambda x.x\,y)(\lambda x.x)$
   // don't want y bound so rename.

4. Apply $\beta-$reduction to the following $\lambda-$expressions as much as possible

   (a) $(\lambda z.z)\ (\lambda y.y\ y)\ (\lambda x.x\ a)$ $\to \lambda y.y\,y(\lambda x.x\,a) \to (\lambda x.x\,a)(\lambda x.x\,a) \to (\lambda x.x\,a)\,a \to a\,a$

   (b) $(\lambda z.z)\ (\lambda z.z\ z)\ (\lambda z.z\ y)$ $\to (\lambda z.z\,z)(\lambda z.z\,y) \to (\lambda z.z\,y)(\lambda z.z\,y) \to (\lambda z.z\,y)\,y \to y\,y$

   (c) $(\lambda x.\lambda y.x\ y\ y)\ (\lambda a.a)\ b$ $\to (\lambda y.(\lambda a.a)\,y\,y)\,b \to (\lambda a.a)\,b\,b \to b\,b$

   (d) $(\lambda x.\lambda y.x\ y\ y)\ (\lambda y.y)\ y$ $\to (\lambda y.(\lambda y.y)\,y\,y)\,y \to (\lambda y.y)\,y\,y \to y\,y$

   (e) $(\lambda x.x\ x)\ (\lambda y.y\ x)\ z$ $\to ((\lambda y.y\,x)(\lambda y.y\,x))\,z \to (\lambda y.y\,x)\,x\,z \to x\,x\,z$

   (f) $(\lambda x.\ (\lambda y.\ (x\ y))\ y)\ z$ $\to (\lambda y.(z\,y))\,y \to z\,y$

   (g) $((\lambda x.x\ x)\ (\lambda y.y))\ (\lambda y.y)$ $\to ((\lambda y.y)(\lambda y.y))\,(\lambda y.y) \to (\lambda y.y)(\lambda y.y) \to \lambda y.y$

   (h) $(((\lambda x.\lambda y.(x\ y))\ (\lambda y.y))\ w)$ $\to ((\lambda y.(\lambda y.y)\,y)\,w \to (\lambda y.y)\,w \to w$

5. Show that the following expression has multiple reduction sequences

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

   $\to ((\lambda y.y\,y\,y)(\lambda x.x\,x\,x))$

   $\to (\lambda x.x\,x\,x)(\lambda x.x\,x\,x)(\lambda x.x\,x\,x)$

   And this would go on forever...

Apply β-reduction to the following term

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

rename $(\lambda y.\ y) \rightarrow (\lambda z.\ z)$

Note: you can only rename bound variables

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

$\rightarrow \boxed{(y\ (\lambda z.\ (z\ y)))}$

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

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

$\boxed{(\lambda x.\ w\ w\ z)}$

Give all free variables in the following $\lambda-$expressions

1. $\lambda x.x\ z\ \lambda y.x\ y$

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

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

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

3. $\lambda x.x\ y\ \lambda x.y\ x$

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

Define the function BV which returns the set of bound variables occurring in a lambda-expression. You do not need to use ML code, define it as we defined the set of free variables in class.

$BV(x) = \{\emptyset\}$

$BV(\lambda x.M) = BV(M)$

$BV(M\ N) = BV(M) \cup BV(N)$

Free Variables

$FV(x) = \{x\}$

$FV(\lambda x.M) = FV(M) - x$

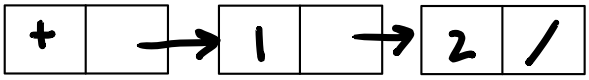
$FV(M\ N) = FV(M) \cup FV(N)$

(19 points) Consider the following simple language E:

```
E ::= num | E + E | E * E
```

where num is any integer.

1. Draw the representation in memory of the Racket expression (quote (+ 1 2))



Draw the representation in memory of the Racket expression

Ask about how this changes in memory

2. In class we have seen the following calculator for this simple language written in Racket (note the interpretations of the symbols + and *):

```
(define (calc e)   (cond
        ((number? e) e)
        ((eq? (car e) '+) (* (calc  (car (cdr e))) (calc (car (cdr (cdr e))))))
        ((eq? (car e) '*) (+ (calc (car (cdr e))) (calc (car (cdr (cdr e))))))
       (else (error "Error: Not a valid E term"))))
```

Give the result of the following invocations, and give a brief explanation.

`(calc (+ 1 2))`

Eagerly evaluates (+ 1 2) to immediately = 3 so it would be (calc 3) which just returns the number 3

= 3

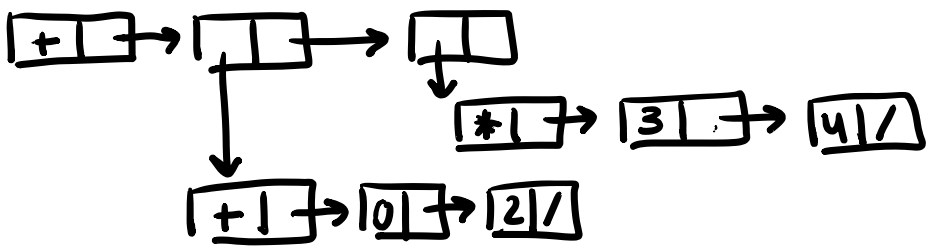
(calc (quote (+ 0 2)))

by putting quote here, this prevents ML from eagerly evaluating (+ 0 2) so calc can be called and it would actually perform the 0 * 2 = 0 and then it would be (calc (0)) which returns the # 0

= 0

(calc (quasiquote (+ (+ 0 2) (* 3 4))))

evaluate the cdr for the expr



SO, evaluate the cdr first

evaluate the calc (+ 0 2) and calc (* 3 4)

↓ ↓

0 * 2 3 + 4

= 0 ↓

= 7

then we have we go back and finish w/

calc (+ 0 7) via quasi

↳ 0 * 7 = 0

```
(calc (quasiquote (+ (unquote (+ 0 2)) (unquote (* 3 4)))))
```

quasiquote evaes car first:

unquote (+ 0 2) unquote (* 3 4)

but then unquote-ing the quasiquote makes it go back to Eagar evaluation

(+ 0 2) = 2 (* 3 4) = 12

and then we go back to the quasiquote:

calc (+ 2 12)

= * 2 12 = **24**

```
(calc '( 1 2))
```

quote makes it lazy to allow the calc fun to be implemented.

calc (1 2)

but this creates an error b/c there is no Operation.

Error : Not a valid E term

Let us consider again the simple language E of arithmetic:

```
E ::= num| E + E | E * E
```

where `num` is any integer. In ML we represent terms of E by using the datatype:

```
datatype E = NUM of int | PLUS of E * E | TIMES of E * E
```

So for example, in order to turn an integer 5 into a value of type `E` we would write `NUM 5` .

1. Represent the expression $2 + 3 + 4$ as a value of type E , where $+$ is left-associative

PLUS (PLUS ((NUM 2), (NUM 3)), (NUM 4))

2. We want to write our interpreter in SML. Complete the following skeleton:

```
fun interp (NUM n )              = interp(n)
  | interp (PLUS   (x , y))      = interp(x) + interp(y)
  | interp (TIMES  (x , y))      = interp(x) * interp(y)
```

Understand the following functions map and reduce :

```
fun map (f, [])          = []
|   map (f, (x :: xs)) = f x :: map (f,xs)

fun reduce (f, [], init)          = init
|   reduce (f, (x::xs), init )  = f (x, reduce (f, xs, init))
```

For example, we have[1]:

```
- map( fn x => if x > 0  then 1 else 0, [1,2,3,0]);
val it = [1,1,1,0] : int list

- reduce (fn (x,y) => x + y, [1,2,3,4],0);
val it = 10 : int
```

1. Compute the sum of the first four squares by invoking the above map and/or reduce. (The sum of the first three squares is $1+4+9$).

First map a list of the first 4 squared numbers

map(fn x => x * x [1,2,3,0]);

= [1,4,3,0]

then reduce it by adding these 4 numbers

reduce (fn x,y => x+y, [1,4,9,0], 0);

= 19

2. Count the number of positive numbers in a list of numbers by invoking map and/or reduce.

map a new list of 1 = positive # or
0 = negative #

map(fn x => if x>0 then 1 else 0, [1,2,-3,4,0]);

=[1,1,0,1,1]

then use the reduce function to add up all the elements of that new list to give you the # of pos. ints

reduce (fn x,y => x+y, [1,1,0,1,1], 0);

= 4

3. Flatten a list of list of natural numbers, such as $[[1,2],[3,4],[5,6],[7,8,9]]$, to a list of numbers by invoking map and/or reduce (you can assume that the append function in ML is called `append`).

Asked on piazza

ML curried MAP function with type

```
fun map(f,[]) = []
  | map(f, x::xs) = f x :: (map f xs)
```

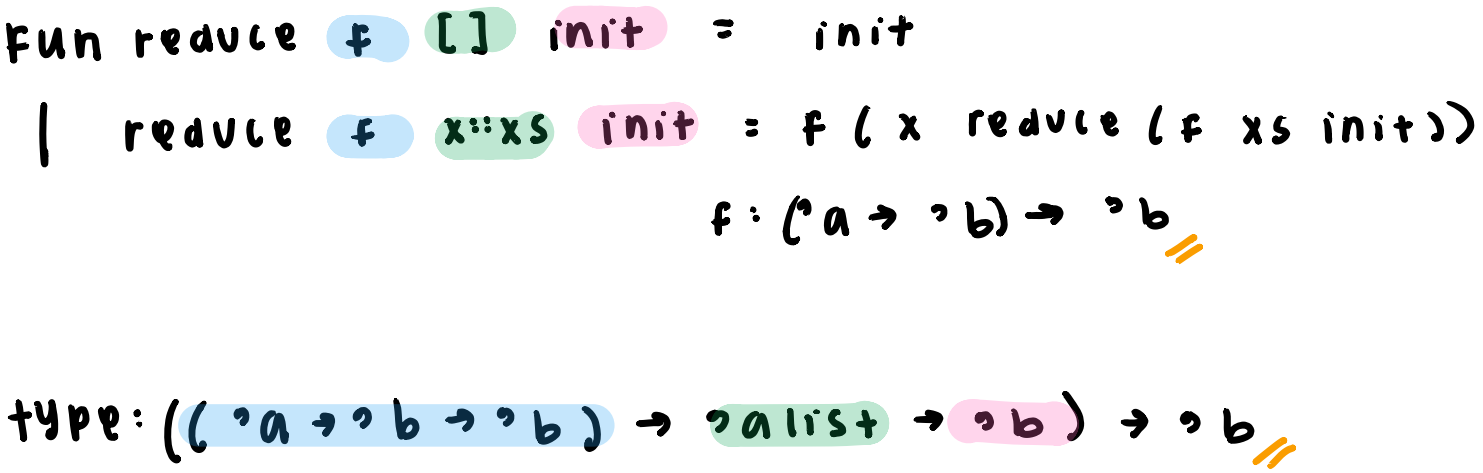
Type: ('a -> 'b) -> 'a list -> 'b list

ML uncurried MAP function with type

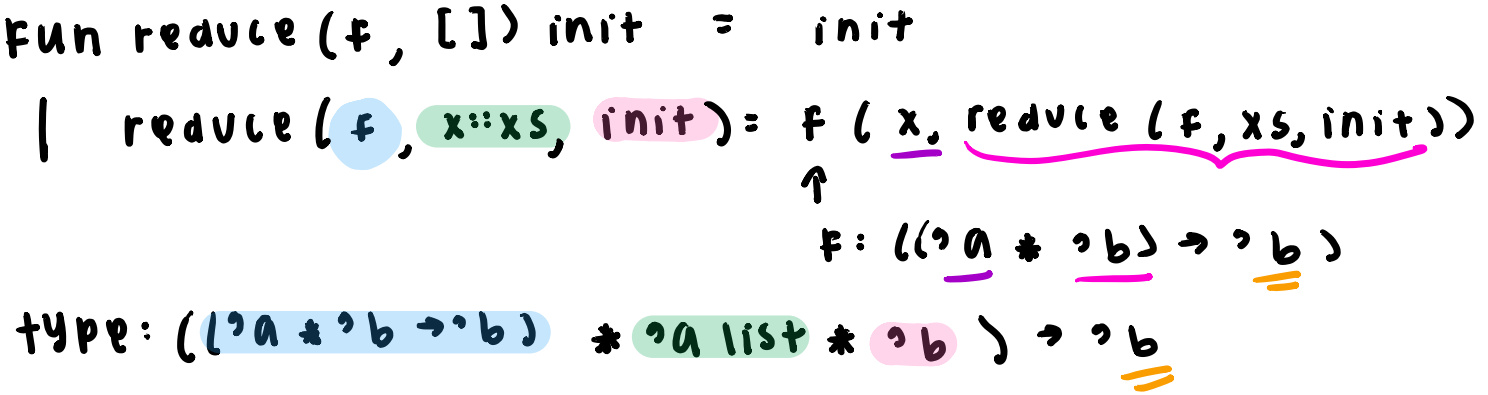
```
fun map(f []) = []
  | map(f x::xs) = f x :: map (f xs)
```

Type: (('a -> 'b) * 'a list) -> 'b list

ML curried REDUCE function with type



ML uncurried REDUCE function with type



Types for the following:

(2,3) (int * int)

(2, true) (int * bool)

[2, 3] int list

[2, true] Type error

cannot have a multi-typed list in ML!

C(f) = fn y => f y

y : 'a

fn : 'a → 'b

f : 'a → 'b

the reason C returns an arrow type is b/c it calls a function f as the result.

C: ('a → 'b) → ('a → 'b)

D(f, x) = f (f (x))

this does not return a function so in the type for D it won't be an arrow type for the return

x : 'a

f : 'a → 'a // has to also return 'a b/c nested call

D : ('a → 'a * 'a) → 'a

F(x , y , b) = if b(y) then x else y

↑ ↑ Return

b(y) : bool , y = 'a * x = 'a

b: 'a → bool

F: ('a * 'a * 'a → bool) → 'a

fun f x y = x < y

curried

x and y could be ints or floats but in ML the default is int

f: (int → int) → bool

Datatype for list containing int and bool:

datatype int-bool-list : b of bool | x of int

What is the type of sort, insert and less

```
fun sort(less, nil) = nil |
                sort(less, a: :l) =
                                let

                fun insert(a, nil) = a: :nil |
                                insert(a, b: :l) = if less(a,b) then a: :(b: :l)
                                                else b:: insert(a, l)
in
                insert(a, sort(less, l))
end;
```

less: ('a * 'a) → bool

insert : ('a * 'a list) → 'a list

sort : (('a * 'a → bool) * 'a list) → 'a list

but keep in mind, ML defavits to int so 'a is defaulted to int.

sort: (int * int → bool) * int list) → int list

CODE and TYPE for CURRY

curry = not pair input

type: ('a * 'b → 'c) → ('a → 'b → 'c)

```
fun curry f x y = f (x, y)
```

CODE and TYPE for UNCURRY

uncurry = pair input

('a → 'b → 'c) → ('a * 'b → 'c)

```
fun uncurry f (x, y) = f x y
```

MERGE in ML `'a list → 'a list → 'a list`

```
fun merge [] ys = ys
  | merge ys [] = ys
  | merge cons(x, xs), cons(y, ys) =

(cons x, fn() => cons y, fn() => merge (xs() ys())))
```

MERGE in Haskell `[a] → [a] → [a]`

```
merge [] ys = ys
merge xs [] = xs
merge (x::xs) (y::ys) = x::y::merge(xs ys)
```

Datatype for Seq

```
datatype 'a seq = Nil | cons of 'a * (unit -> 'a seq)
```

```
Nil | cons of 'a * (unit -> 'a seq)
```

```
Nil | cons of 'a * (unit -> 'a seq)
```

```
Nil | cons of 'a * (unit -> 'a seq)
```

```
Nil | cons 'a * (unit -> 'a seq)
```

```
datatype 'a seq = Nil | cons of 'a * (unit -> 'a seq)
```

```
datatype 'a seq = Nil | cons of 'a * (unit -> 'a seq)
```

ML Typechecker

Type of typeof?

term * env → typ

```
fun typeOF env (AST_ID s) = env s
  | typeOF env (AST_NUM n) = int
  | typeOF env (AST_BOOL b) = bool
  | typeOF env (AST_FUN (i, t, e)) = Arrow (t, typeof (update env i t) e)
  | typeOF env (AST_APP (e1, e2)) =
```

```
case typeof env e1 of
   Arrow (t1, t2) =>
   case typeOF env e2 of
     true => t2
   | false => raise TypeError
|_ => raise TypeError
```

```
|  typeOF env (AST_SUCC ) = ARROW(int, int)
|  typeOF env (AST_PRED) = ARROW (int, int)
|  typeOF env (AST_ISZERO ) = ARROW (int, bool)
|  typeOF env (AST_IF (e1, e2, e3)) =
```

```
let  t1 = typeof env e1
     t2 = typeof env e2
     t3 = typeof env e3
in case (t3 = bool) andalso (t1 = t2) of
     true => t3
   | false => raise TypeError

end
```

Call your type checker with the representation of the invocation.
`fn x:int => fn y: bool => x` and give the result of the

Calling typeof to get the type type

ARROW(int, ARROW(BOOL, int))

x = input int → y
y = bool → int

∴ ARROW(int, ARROW(bool, int))
x : int → bool → int

But if it was trying to find the type of the term it would:

fn x:int => fn y:bool => x

↑ AST_FUN(string, typ, term)

↖ AST_FUN(string, typ, term)

AST_FUN("x", int, AST_FUN("y", bool, AST_ID "x"))

1. Write down ML expressions of type typ corresponding to the abstract syntax trees for each of the following type expressions:

typ = what type of returns

(a) `int`

int

(b) `int -> bool`

Arrow(int, BOOL)

(c) `('a -> 'b) -> ('a -> 'b)`

ARROW(Arrow('a,'b), Arrow('a,'b))

2. Write down ML expressions of type term corresponding to the abstract syntax trees for each of the following expressions

term = AST

(a) `succ (pred 5)`

AST_APP(AST_SUCC, AST_APP(AST_Pred, AST_NUM 5))

(b) `if 7 then true else 5`

AST_IF(AST_NUM 7, AST_BOOL true, AST_NUM 5)

(c) `fn a : int => f a a`

F: a→a

a: int

AST_FUN("a", INT, AST_APP(
AST_APP(AST_ID "F", AST_ID "a"),
AST_ID "a"))

AST_FUN: string * typ * term

AST_App : term * term

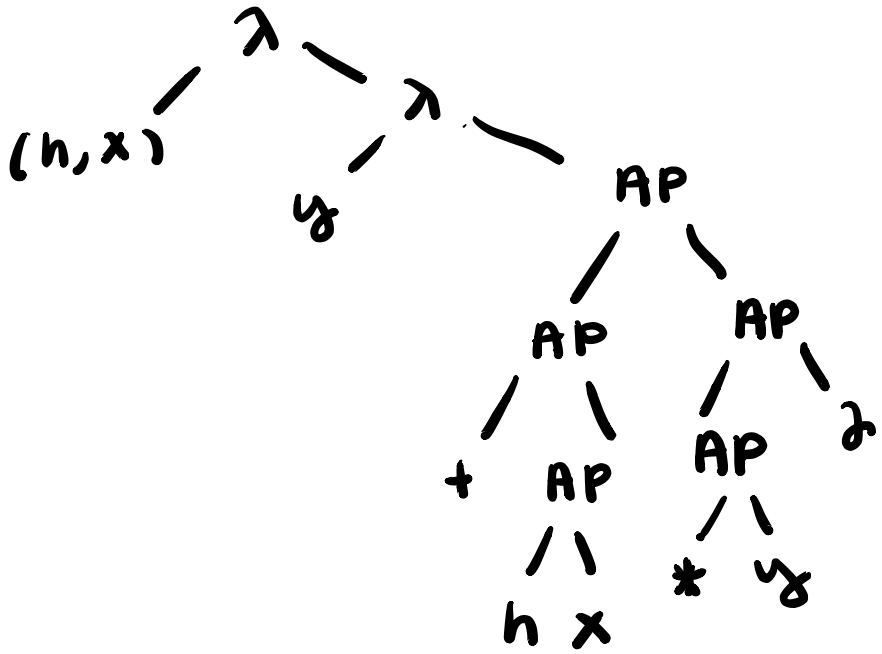
AS-IF : term * term * term

- AST_ID's for terms

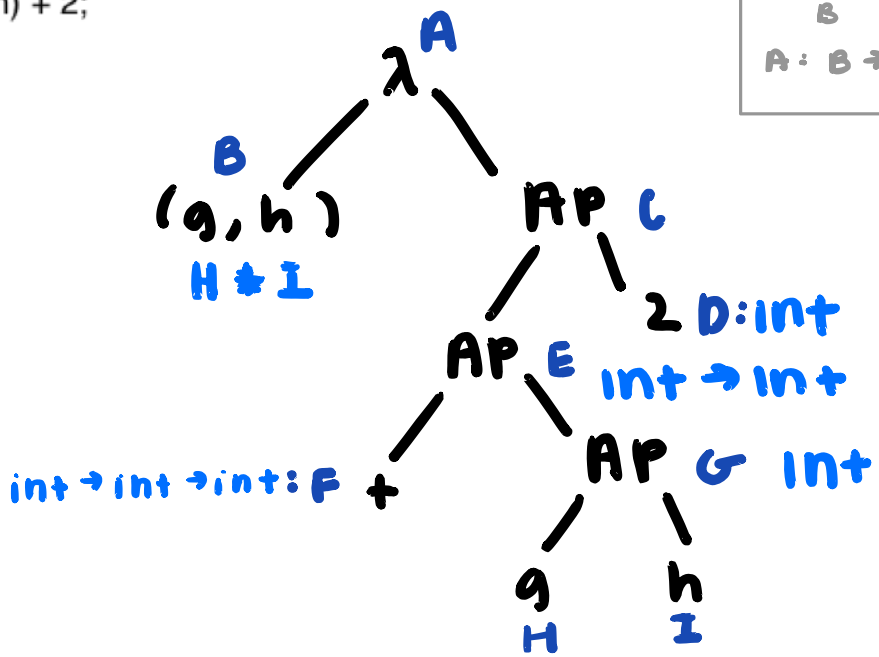
PARSE GRAPHS

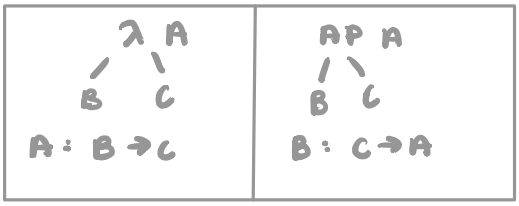
fun g (h, x) = fn y = (h x) + (y * 2)

Uncurried (pair input)



fun f(g,h) =g(h) + 2;





A : B → C

B : H * I : I → int * I

C : int

D : int

E : D → C : int → int

F : + : int → int → int : G → E : (int) → (int → int)

G : int

H : I → G : I → int :

I : I

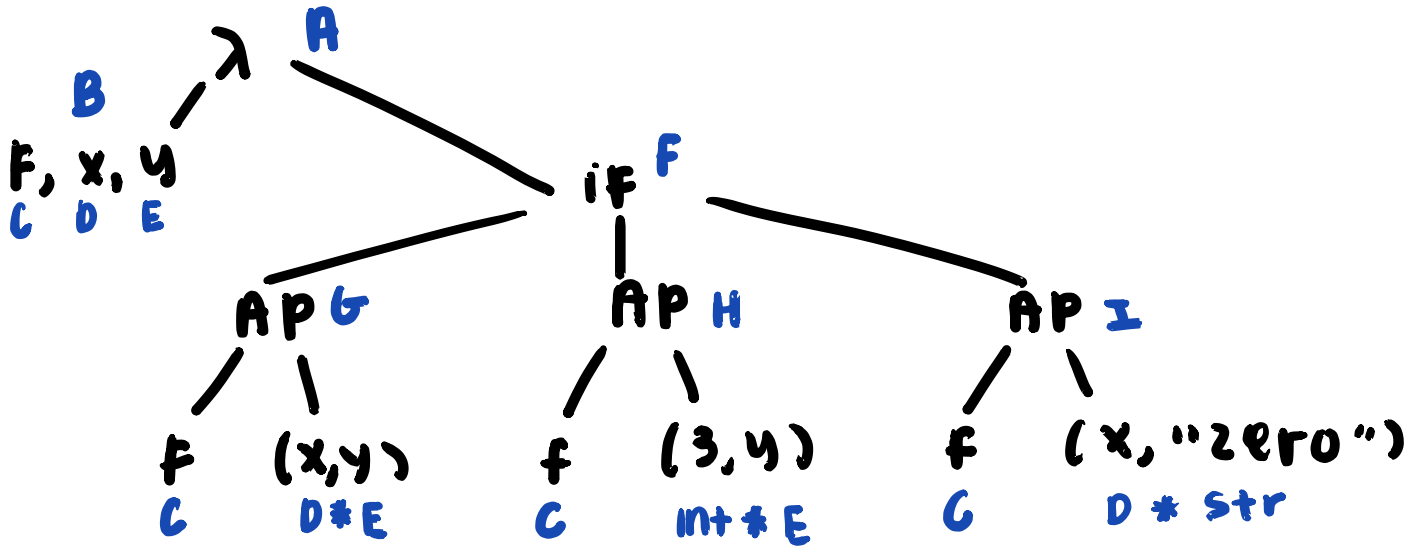
therefore,

A : B → C

(I → int * I) → int

fun ff (f, x, y) = if f(x, y) then f(3, y) else f (x, "zero")

uncurried = input pairs



A : B → F

B : (C * D * E)

C : (D * E) → G (int * E) → H (D * str) → I ∴ (int * str) → BOOL

D : int

E : str

F : F = H = I : bool

G : bool

H : H = F = I : bool

I : I = F = H : bool

(D * E) → G → BOOL
(int * E) → H
(D * str) → I

D : int

E : str

G : BOOL ∴ H, I = BOOL

∴ C : (int * str) → BOOL

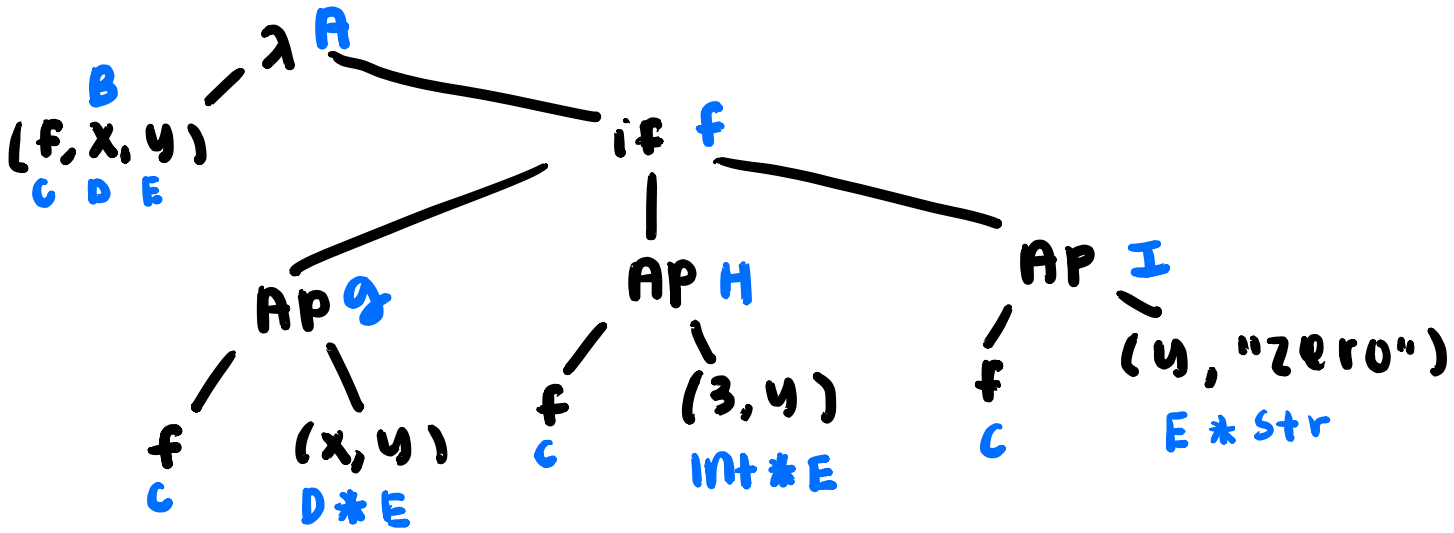
therefore,

A : B → F

((int * str → bool) * int * str) → bool

fun gg (f, x, y) = if f(x, y) then f(3, y) else f (y, "zero")

Uncurried (input pairs)



A : B → F

B : (C * D * E)

C : (D * E) → G (int * E) → H (E * str) → I

D :

E :

F : F = H = I

G : bool

H : H = F = I

I : I = F = H

(D * E) → G

(int * E) → H

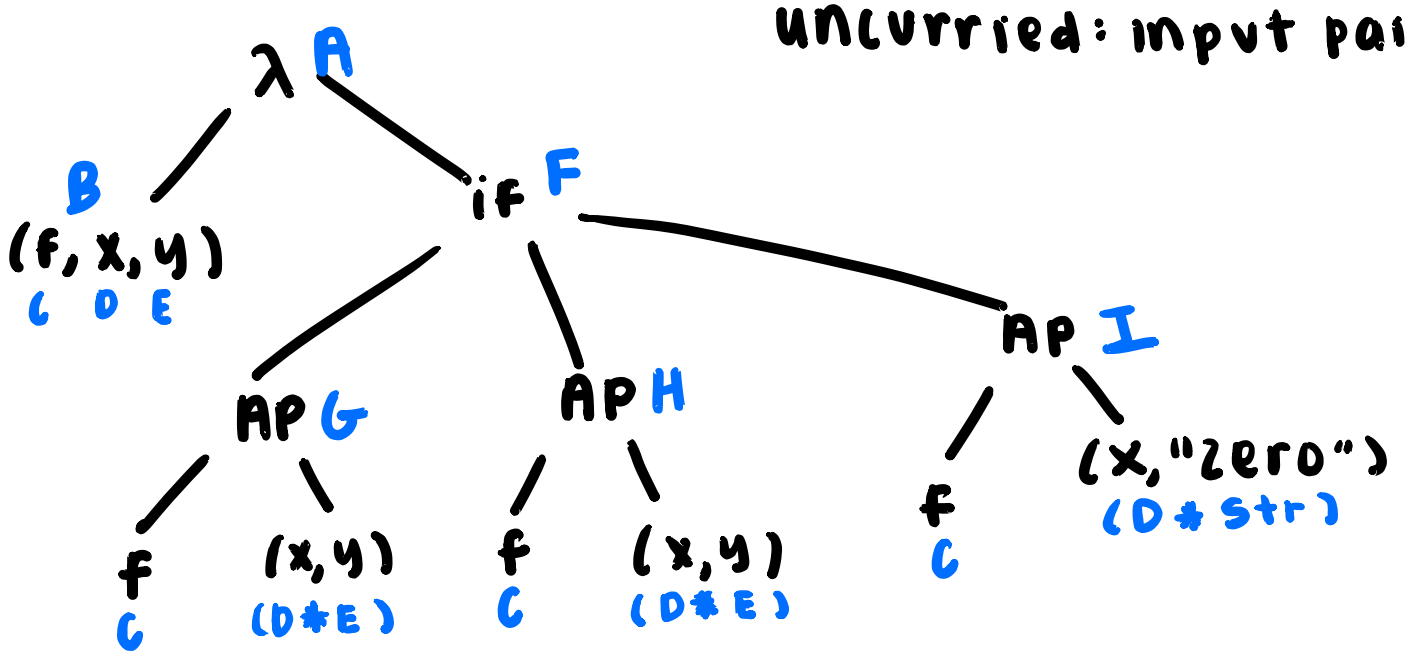
(E * str) → I

D = int

E = int E = str

TypeError

fun hh (f, x, y) = if f(x, y) then f(x, y) else f (x, "zero")



A : B → F

B : (*D*E : C*D*str

C : (D*E)→G (D*E)→H (D*str)→I : (D*str)→bool

(D*E)→G →bool

(D*E)→H

(D*str)→I

↳ E = str

H = bool

I = bool

D :

E : str

F : F = H = I : bool

G : bool

H : F = H = I : bool

I : F = H = I : bool

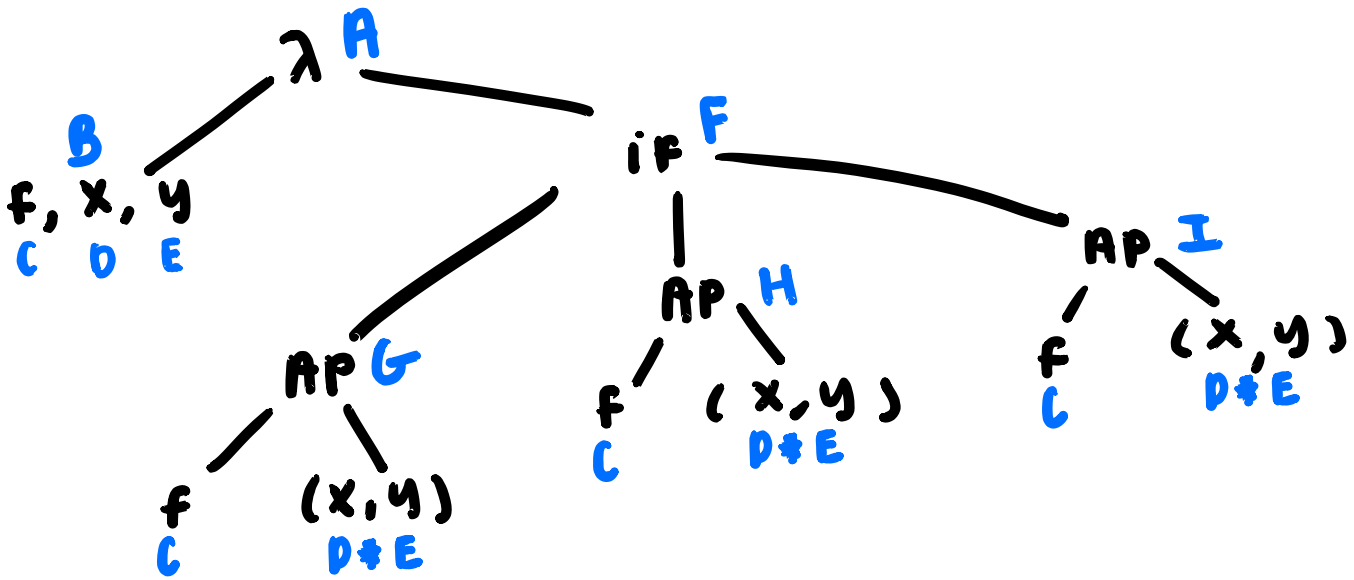
therefore,

A: B → F

((D*str)→bool * D * str) → bool

fun ee (f, x, y) = if f(x, y) then f(x, y) else f (x, y)

Uncurried = input pairs



A : B → F

B : C * D * E

C : (D * E) → G (D * E) → H (D * E) → I

D :

E :

F : F=H=I : bool

G : bool

H : F=H=I : bool

I : F=H=I : bool

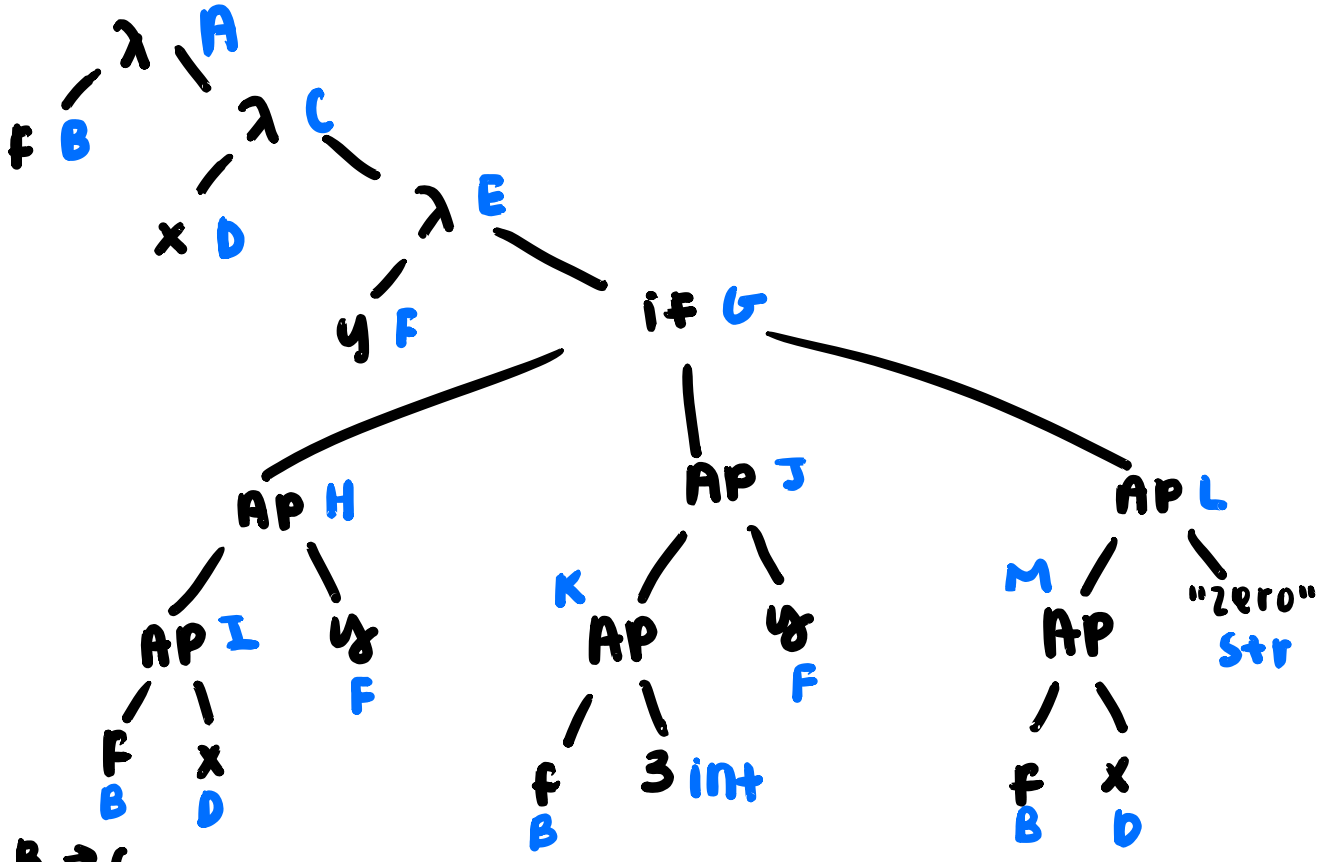
(D * E) → G (→ bool)
(D * E) → H
(D * E) → I
∴ H = bool
I = bool

therefore,

$$(((D * E) \to bool) * D * E) \to bool$$

fun ff f x y = if (f x y) then (f 3 y) else (f x "zero")

Curried



A : B → C

B : D → I int → K D → M : int → Str → bool

C : D → E : int → Str → bool

D : int

E : F → G : Str → bool

F : Str

G : G = J : L : bool

H : bool

I : F → H : F → BOOl

J : G = J : L : bool

K : F → J

L : G = J : L : bool

M : Str → L

D → I
int → K
D → M
∴ D = int

I : F → BOOl

K : F → J

M : Str → L
∴ F = Str
∴ J = bool
∴ L = bool

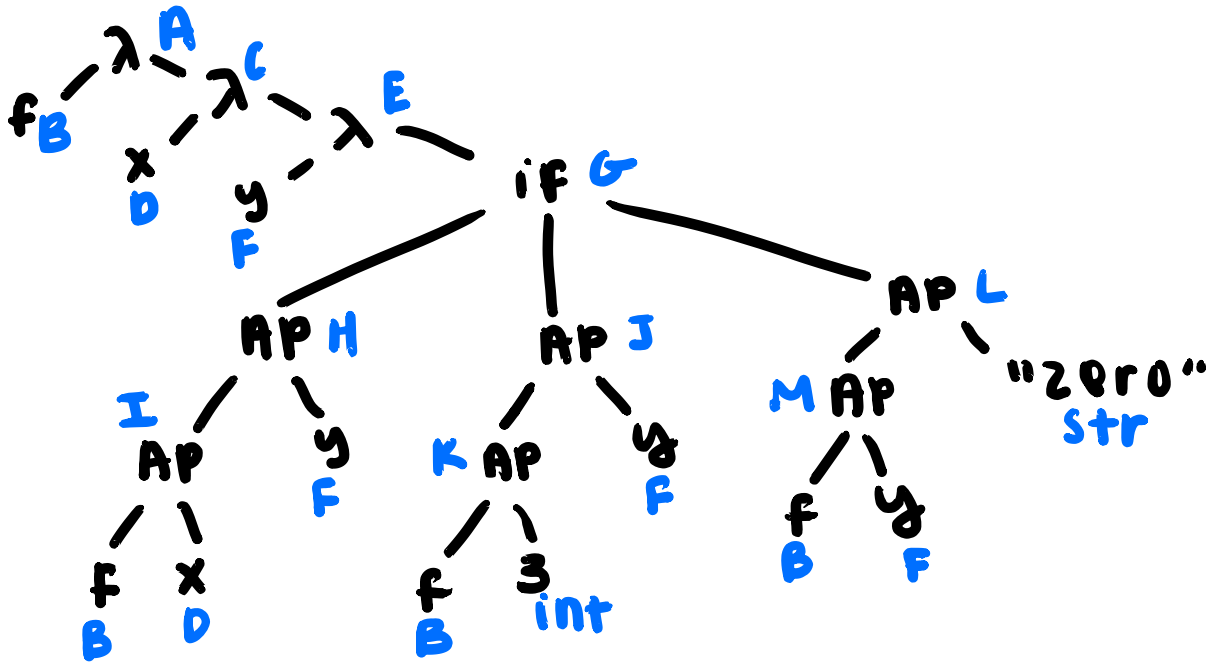
therefore,

B → C

(int → str → bool) → (int → Str → bool)

fun gg f x y = if (f x y) then (f 3 y) else (f y "zero")

Curried



A : B → C

B : D → I int → K F → M

C : D → E : int → E

D : int

E : F → G : int → G

F :

G : G = J = L

H : bool

I : F → H

J : G = J = L

K : F → J

L : G = J = L

M : Str → L

D → I
int → K
F → M

∴ D = int

∴ F = int

I : F → H

K : F → J

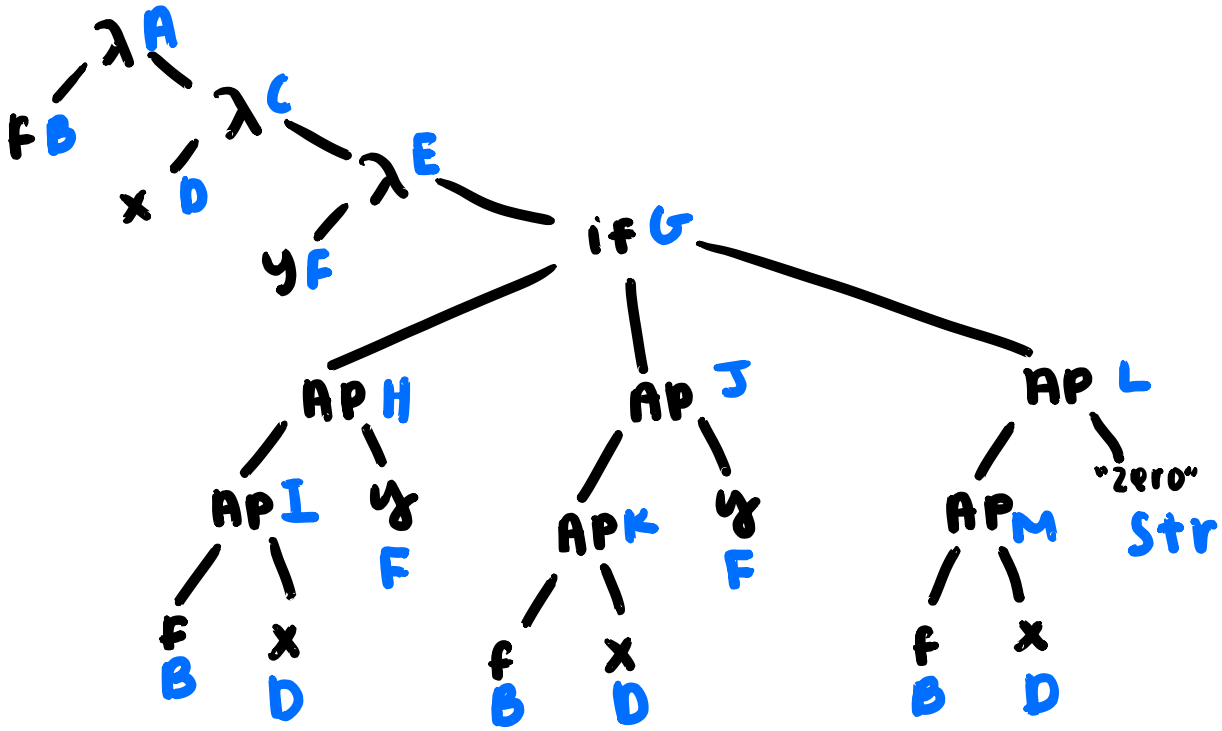
M : str → L

∴ F = Str

Type Error!

fun hh f x y = if (f x y) then (f x y) else (f x "zero")

curried



A : B → C

B : D → I D → K D → M : D → Str → bool

C : D → E

D :

E : F → G : Str → G : Str → bool

F : Str

G : G = J = L : bool

H : bool

I : F → H : bool → bool

J : G = J = L : bool

K : F → J : F → bool

L : G = J = L : bool

M : Str → L : Str → bool

D → I → F → H → bool

D → K → F → J

D → M → Str → L

∴ F = Str

∴ J, L = bool

therefore,

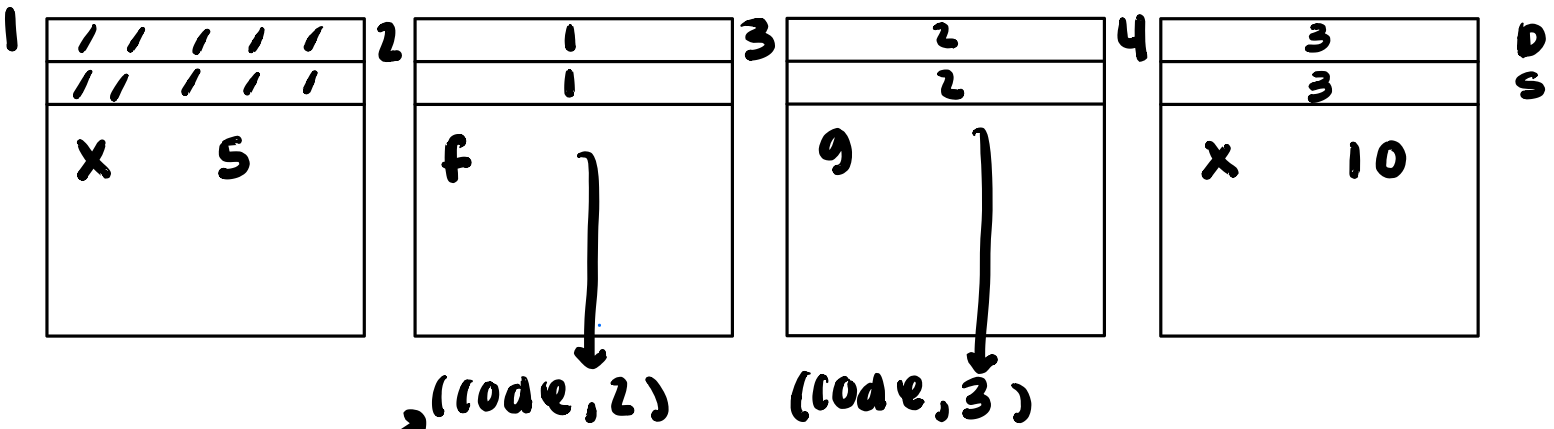
A : B → C

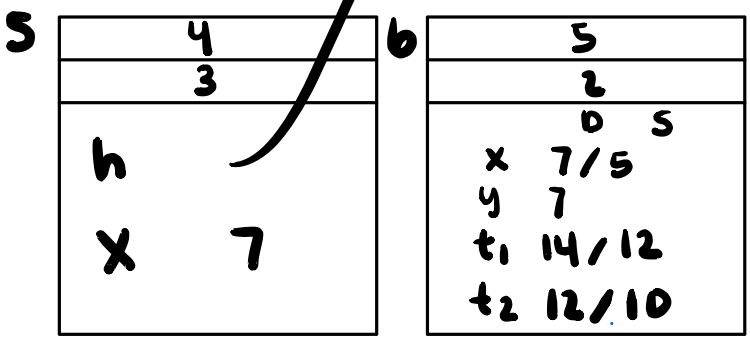
(D → Str → bool) → (D → Str → bool)

```
val x = 5;
fun f(y) = (x+y) −2;
fun g(h) = let val x = 7 in h(x) end;
let val x = 10 in g(f) end;
```

t_1 t_2

dyn. next AR
static AR defined





therefore,
statically = 10
dynamically = 12

g(f)
static = 3
b/c that it where g is defined

F(7)
static = 2
b/c thats where f is defined

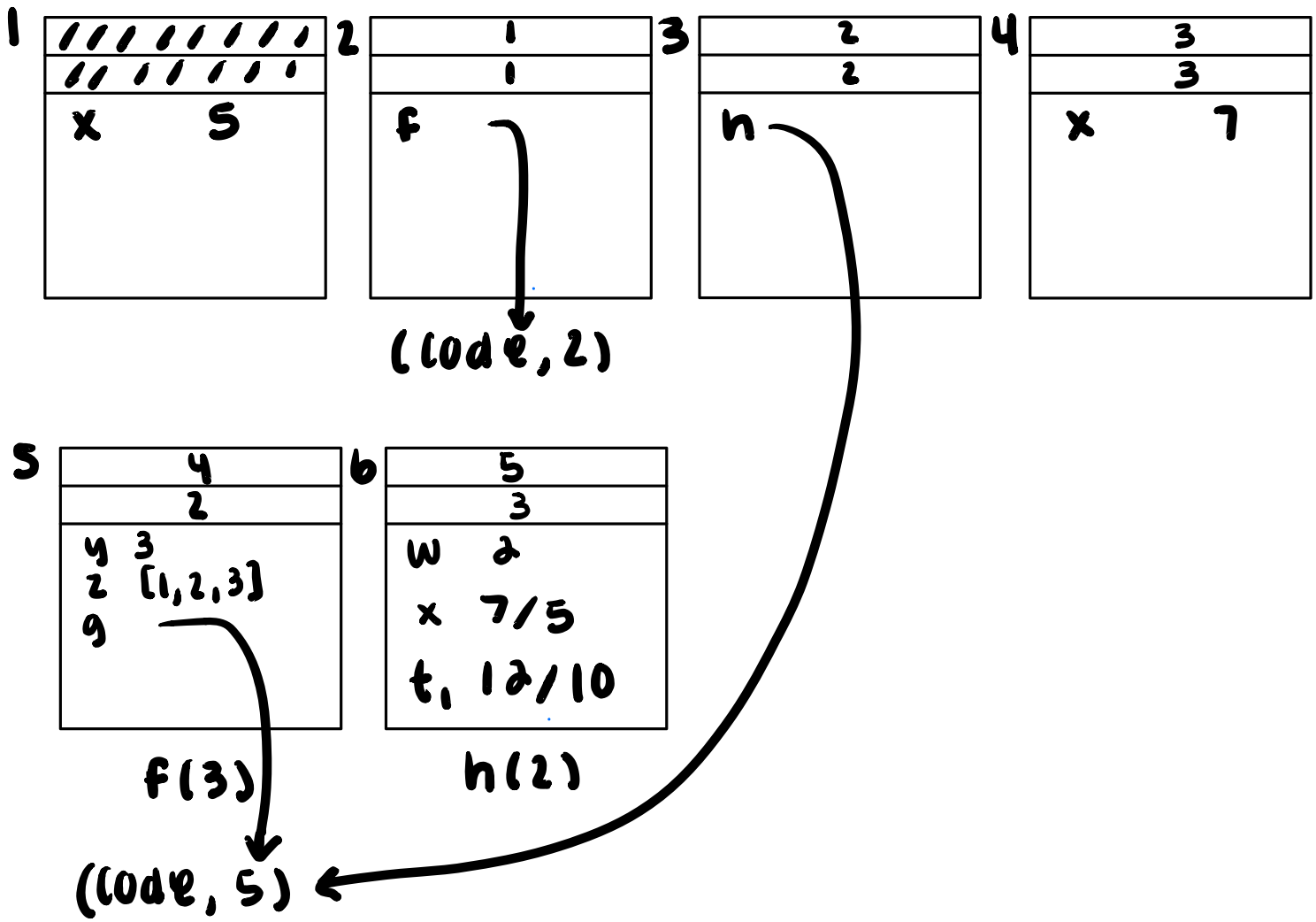
```
val x = 5;

fun f(y) =
        let val z = [1, 2, 3] (* declare list *)
            fun g(w) = w+x+y (* declare local function *)
        in
            g       (* return local function *)
        end;

val h = let val x=7 in f(3) end;

h(2);
```

t1



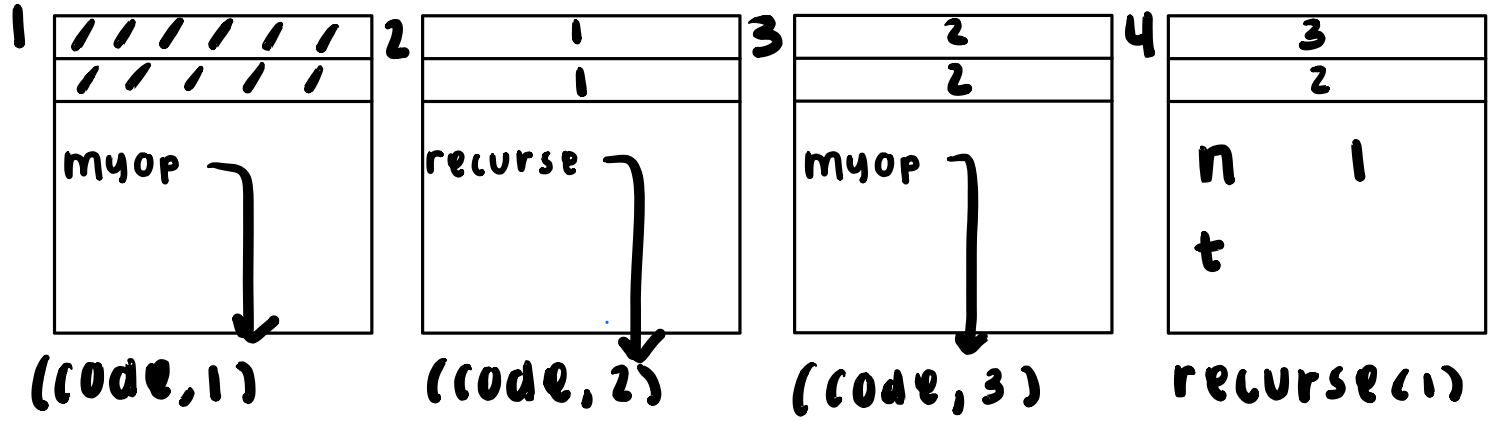
therefore,

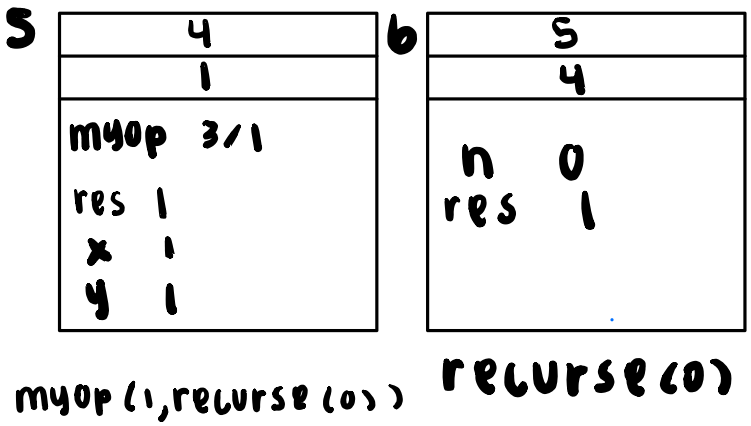
dynamically = 12

statically = 10

```
1. fun myop(x,y) = x*y;
2. fun recurse(n) =
3.     if n=0 then 1
4.     else myop(n, recurse(n-1));
5. let
6.     fun myop(x,y) = x + y
7. in
8.     recurse(1)
9. end;
```

> t





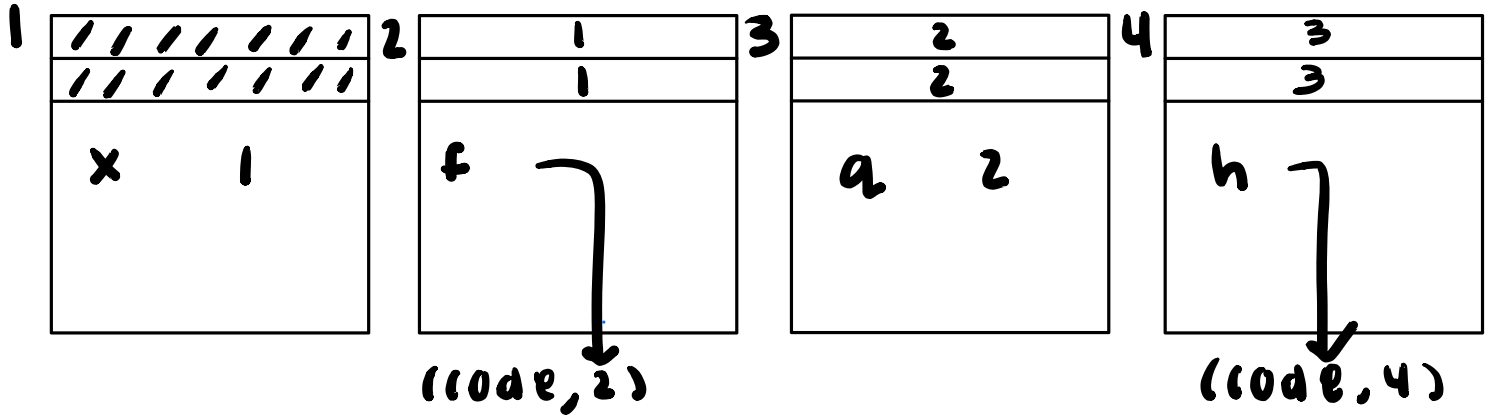
dynamic: myop AR.3 → x+y → 1+1 = 2

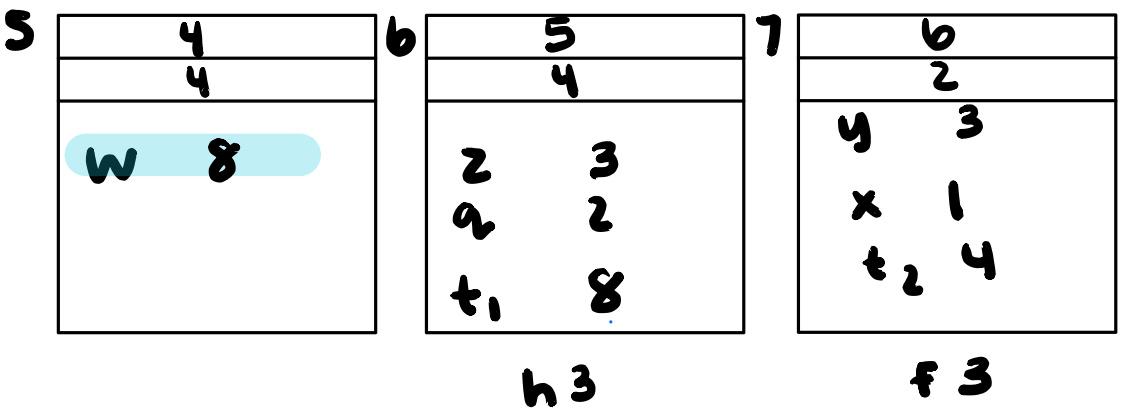
Static: myop AR.1 → x*y → 1*1 = 1

```
let val x = 1
in let fun f y = y + x;
   in  let val q = 2
       in let fun h z = (f z ) * q;
          in let val w = h 3 in w
             end
          end
       end
   end
end
```

t_2

t_1

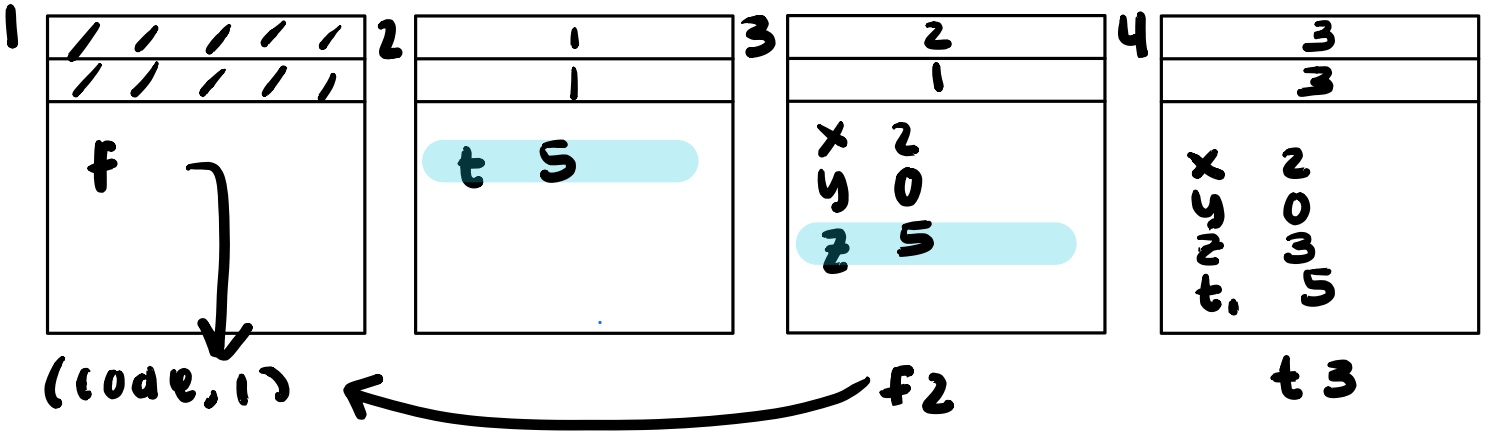




dynamic and static both return 8

```
let
fun f x = let val y = 0
          in fn z => x + y + z
          end
in  let
        val t = f 2
    in  t 3
    end
end
```

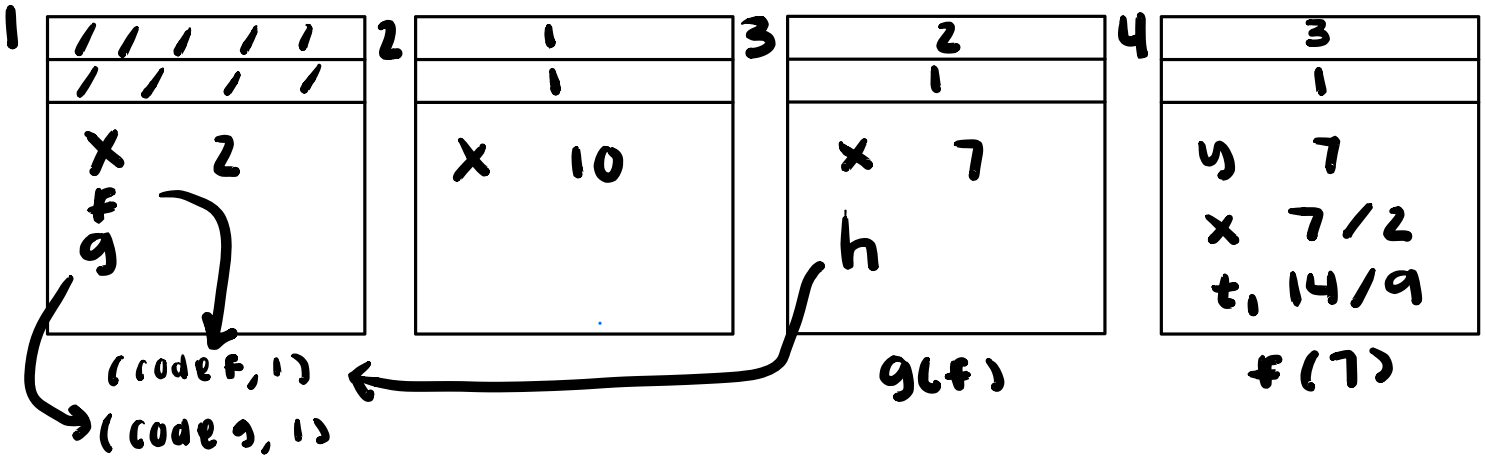
t_1



Static / dynamic = 5

this code cannot adopt a stack-based discipline b/c we are returning a function.

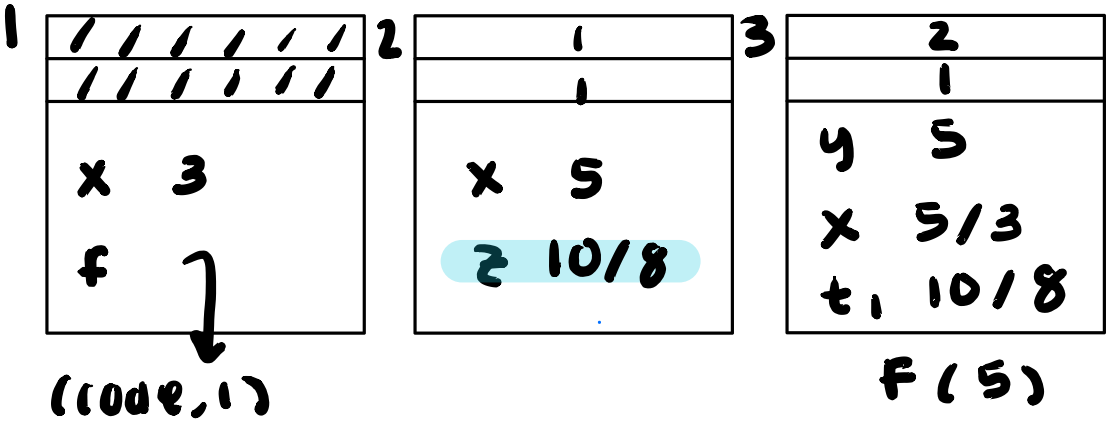
```
let val x = 2
    fun f y = x + y > t1
    fun g h = let val x = 7 in h(x) end
in
  let val x = 10
  in g(f)
  end
end
```



dynamically = 14

statically = 9

```
1. let val x = 3
2.     fun f y =  y + x ? t1
3. in  let val x = 5
4.         val z  =  x +
5.                        f (x)
6.     in z
7.     end
8. end
```



dynamically = 10

statically = 8

— NEW material here —

8.1 Exceptions

Consider the following functions, written in ML:

```
exception Excpt of int;

fun twice(f,x) = f(f(x)) handle Excpt(x) => x;

fun pred(x) = ifx=0 then raise Excpt(x) else x-1;

fun dumb(x) = raise Excpt(x);

fun smart(x) = 1 + pred(x) handle Excpt(x) => 1;
```

when an exception is raised, it looks for the handler in the current activation record.

If no handler is found, then look in the activation record of the caller.

What is the result of evaluating each of the following expressions?

a. twice(pred,1);

f = pred x = 1

twice = pred (pred (1)) handle Excpt (x) => x;

pred (1) = if x = 0 then raise Excpt (x) else x-1;
false 1-1 = 0

pred (0) = if x = 0 then raise Excpt (x) else x-1;
true

Outer pred (0) raises Excpt (0) which is handled by the function that called it which is twice and 0 is returned.

b. twice(dumb,1);

f = dumb x = 1

twice = dumb (dumb (1)) handle Excpt (x) => x;

dumb (1) = raise Excpt (x);

the inner dumb (1) function raises Excpt (1) and that is handled by twice to return 1.

c. twice(smart,0);

f = smart x = 0

twice = smart (smart (0)) handle Excpt (x) => x;

smart (0) = 1 + pred (0) handle Excpt (x) => 1;

pred (0) = if x = 0 then raise Excpt (x) else x-1;
true

pred(0) raises Excpt (x) and x = 0 so twice handles this raised Excpt so = 1

8.4 Exceptions and Recursion

Here is an ML function that uses an exception called Odd.

```
fun f(0) = 1
  | f(1) = raise Odd
  | f(3) = f(3-2)
  | f(n) = (f(n-2) handle Odd => ~n)
```

The expression ~n is ML for -n, the negative of the integer *n*.

When f(11) is executed, the following steps will be performed:

call f(11)

call f(9)

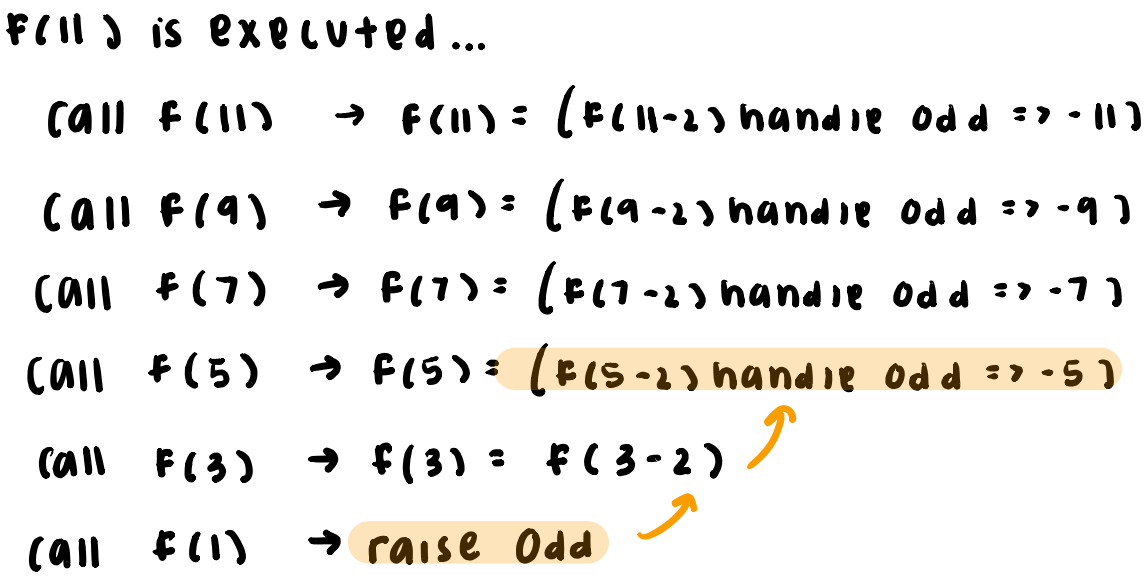
call f(7)

…

Write the remaining steps that will be executed. Include only the following kinds of steps:

- function call (with argument)
- function return (with return value)
- raise an exception
- pop activation record of function off stack without returning control to the function
- handle an exception

Assume that if f calls g and g raises an exception that f does not handle, then the activation record of f is popped off the stack without returning control to the function f.



f(1) raises the exception Odd which is then handled by f(5) which returns -5

POWER FUNCTION

Power function in ML Recursion

Type `Int → Int → Int`

Code

```
fun power x 0 = 1
  | power x y = x * power(x (y-1))
```

Power function in ML continuation passing style

Type `int → int → (int → a) → a`

Code

```
fun power_cps x 0 k = k 1
  | power_cps x y k = power_cps x (y-1) (fn v => k(x*v))
```

PRODUCT FUNCTION

Product of a list function in ML Recursion

Type

```
int list → int list
```

Code

```
fun prod [] = 1
  | prod (x::xs) = x *(prod xs)
```

Product of a list function in ML continuation passing style

Type `int list → (int → a ) → a`

Code

```
fun prod [] k   = k 1
  | prod (x::xs) = prod (xs fn v => k (x * v))
```

REDUCE FUNCTION

REDUCE in ML CURRIED

Type

f:('a → 'b → 'b) → 'b

reduce : (('a → 'b → 'b) → 'a list → 'b) → 'b

Code

```
fun reduce f [] init = init
    reduce f x::xs init = f (x reduce(f xs init))
```

REDUCE in ML UNCURRIED

Type

f : ('a * 'b * 'b)

reduce : ((('a * 'b → 'b) * 'a list * 'b) → 'b)

Code

```
fun reduce (F, [], init)
    reduce(F, x::xs, init) = f(x, reduce (F, xs, init))
```

REDUCE in HASHELL

Type

Code

postponed for now

MAP FUNCTION

MAP in ML CURRIED

Type

f: 'a -> 'b

map: (('a -> 'b) -> 'a list) -> 'b list

Code

```
fun map f [] = []
    map f x::xs = f x :: map f xs
```

MAP in ML UNCURRIED

Type

f: 'a -> 'b

reduce: (('a -> 'b) * 'a list) -> 'b list

Code

```
fun map(f,[]) = []
    map(f, x::xs) = f(x) :: map(f, xs)
```

MAP in HASKELL

Type

Code

ML Sequences / Lazy Eval.

```
datatype LazyIntList = Nil | Cons(int * (unit -> LazyIntList))
```

LazyIntList is a **sequence** for ML that allows the Eagar language (ML) to act lazy like Haskell.

This is how we can create **infinite lists** via ML. the sequence datastructure (lazy int list) starts by having a known value and continues with a function to compute the rest of the sequence.

the generic sequence datatype:

```
datatype 'aseq = Nil | Cons (a * (unit -> 'aseq))
```

This **lazy sequence datatype** provides a way to create infinite sequences with each infinite sequence represented by a function that computes the next element in the sequence.

Example:

```
datatype LazyIntList = Nil | Cons(int * (unit -> LazyIntList))

fun ones = Cons ( 1, (fn() => ones))
```

↖ tail recursion so CPS...?

```
ones: LazyIntList
```

↖ sequence type b/c infinite list of ones

Define in HASKELL an infinite list that is never ending 1's

```
ones :: [int]
ones = 1 : ones
```

Define in ML an infinite list that is never ending 1's

```
datatype lazyintlist = Nil | Cons (int * (unit -> lazyintlist))

fun ones = Cons (1, fn() => ones)
```

Define in HASKELL a function intList that creates an infinite list from n to infinity

```
intlist :: Int -> [Int]

intlist n = n : intlist n+1
```

Define in ML a function intList that creates an infinite list from n to infinity

```
datatype lazyintlist =
            Nil|Cons(int * (unit -> lazyintlist))

fun intlist n = Cons(n, fn() => intlist n+1)

intlist : int -> lazyintlist
```

Define in HASKELL a function takeN that returns the first n elements from a list

ex: takeN 4 (intList 10)
= [10, 11, 12, 13]

```
takeN :: Int -> [a] -> [a]

takeN 0 xs = []
takeN n x:xs = x: takeN (n-1) xs
```

Define in ML a function takeN that returns the first n elements from a list

```
datatype lazyintlist = Nil | (int * (unit -> lazyintlist))

fun takeN 0 xs = []
    takeN n x::xs = cons (x, fn() => takeN n-1 xs)

takeN = int -> lazyintlist -> lazyintlist
```

Define in HASKELL a function that returns a list of all positive even integers

```
evens :: [Int]
evens = map (\x -> x * 2) (intlist 0)
```

Define in ML a function that returns a list of all positive even integers

```
mapint : (int -> int) -> lazy intlist -> lazy intlist
fun mapint f [] = []
    mapint f (cons (x, fun)) =
      cons (f x, fn () => mapint f fun())

even = lazyintlist

datatype lazyintlist = Nil | Cons (int * unit -> lazy intlist)

fun evans = mapint (fn x => 2 * x) (intlist 0)
```

Define in HASKELL a function that returns a list of all positive odd integers

```
Odd :: [int]

Odd = map (\x = x * 2 - 1 ) (intlist 1 )
```

Define in ML a function that returns a list of all positive odd integers

```
Odd : lazyintlist

FUN Odd = mapint (fn x => x * 2 - 1) (intlist 1)
```

Define in HASKELL a function that merges two lists

```
merge :: [a] -> [a] -> [a]

merge  xs []  = xs
merge  []  ys = ys
merge  x:xs  y:ys = x:y:merge (xs  ys)
```

Define in ML a function that merges two lists

```
merge : 'a list -> 'a list -> 'a list

fun merge  Nil  ys = ys
  | merge  xs  Nil = xs
  | merge  x::xs  y::ys =

(cons(x, fn() => (cons(y, fn() => merge (xs  ys)))
```

Does the call

```
merge evens odds
```

terminate? Explain why or why not in a few sentences. What about

```
length (merge evens odds)
```

merge evens odds

does terminate b/c even though they are infinite, Haskell is lazy so they don't need to terminate at once.

length(merge evens odds)

does NOT terminate because you are calling a function on something that will never complete.

3. Write each of the sequences below as one or more Haskell streams (infinite lists). You may use the *merge* function defined above.

   (a) $0, 1, 8, 27, 64, 125, 216, 343, 512, 729, 1000, 1331, \ldots$

   (b) $1, 3, 9, 27, 81, 243, 729, 2187, 6561, 19683, 59049, \ldots$

   (c) $0, 0, 1, 1, 2, 4, 3, 9, 4, 16, 5, 25, 6, 36, 7, 49, \ldots$

   (d) The negative numbers

a) 0 1 2 3
map x^3

```
fun_a = map (\x -> x^3) (intlist 0)
```

b) 0 1 2 3
map 3^x

```
fun_b = map (\x -> 3^x) (intlist 0)
```

c) 0 0 1 1 2 4 3 9
↓ ↓ ↓ ↓
x²

merge (intlist w/ x²)

```
fun_c = merge (intlist 0 map (\x -> x^2) (intlist 0))
```

d) -1 -2 -3

```
fun_d = map (\x = x * -1) (intlist 1)
```

Interp in ML

```
datatype term = Num of int
              | PLUS of (int * int)
              | TIMES of (int * int)

fun interp NUM x = x
    interp PLUS x y = interp x + interp y
    interp TIMES x y = interp x * interp y
```

Interp in HASKELL with the following type
interp :: Term -> Result

```
data Term = Num Int
          | PLUS  Term Term
          |TIMES Term Term
          |DIVIDE Term Term

data Result = OK Int
            | Error String

interp (NUM n) = OK n
interp (PLUS x y) = case interp x of
                      OK n -> case interp y of
                                OK m -> OK(n+m)
                      Error e -> Errore
interp (TIMES x y) = case interp x of
                       OK n -> case interp y of
                                 OK m -> OK n*m
                       Error e -> Error e
interp (DIVIDE x y) = (case interp x of
                       OK n -> case interp y of
                                 OK m -> IF m /= 0
                                         then OK n/m
                                         else Error

                                 Error e -> Errore
                       Error s -> Errors
```

Interp in HASKELL with the following type

interp :: Term -> Result using a Check function that has type check :: Result (Int -> Result) -> Result

```
check (OK n) f   = f n
check (Error s) f = Error s

interp (Num n) = OK n
interp (Plus x y) = (interp x) `check`
                    (\n -> interp y) `check`
                    (\m -> OK n+m)
interp (Times x y) = (interp x) `check`
                     (\n -> interp y) `check`
                     (\m -> OK n*m)
interp (Divide x y) = interp x `check`
                      (\n -> interp y `check`)
                      (\m -> if m /= 0
                             then OK n/m
                             else Error)
```

Interp in HASKELL with the following type

using a Monad for the result type

```
data RESULT r =
                OK     r
              | ERROR  String
```

```
instance Monad result where
  return n     = OK n
  (>>=)(OK n) f = f n
  (>>=)(Error e) = Error e
```

and interp :: Term -> RESULT Int

```
interp Num x   =  OK x
interp Plus  x  y  = do
                n <- interp x ;
                m <- interp y ;
                return (n+m)
interp times x y  =  do
                n <- interp x;
                m <- interp y;
                return n*m;
interp Divide x y  = do
                n <- interp x;
                m <- interp y:
                if m /= 0
                  then return n/m;
                  else Error
```

1. For the following Algol-like program, write the number printed by executing the program under each of the listed parameter passing mechanisms.

```
begin
   integer i;
   procedure pass ( x, y )
    begin
      integer x, y; // types of the formal parameters begin
      x := x + 1;
      y := x + 1;
      x := y;
      i := i + 1
    end;

   i := 1;
   pass (i, i);
   print i
end
```

(a) pass-by-value 2

(b) pass-by-reference 4

(c) pass-by-value-result 3

(a) pass-by-value

```
begin
   integer i;
   procedure pass ( x, y )
    begin
      integer x, y; // types of the formal parameters begin
      x := x + 1;
      y := x + 1;
      x := y;
      i := i + 1
    end;

   i := 1;
   pass (i, i);
   print i
end
```

(1,1)

x = 1+1 = 2
y = 2+1 = 3
x = 3
i = 1+1 = 2

print i

= 2

```
begin
   integer i;
   procedure pass ( x, y )
    begin
      integer x, y; // types of the formal parameters begin
      x := x + 1;
      y := x + 1;
      x := y;
      i := i + 1
    end;

   i := 1;
   pass (i, i);
   print i
end
```

x = 1+1 = 2
y = 2+1 = 3
x = 3
i = 1+1 = 2

Referencing py = 3

Return 3

(b) pass-by-reference

```
begin
   integer i;
   procedure pass ( x, y )
    begin
      integer x, y; // types of the formal parameters begin
      x := x + 1;
      y := x + 1;
      x := y;
      i := i + 1
    end;

   i := 1;
   pass (i, i);
   print i
end
```

int ref, int ref

x = ref 2
y = ref 3
x = ref 3
i = ref 4

so i prints the ref 4

= 4

ex)

```
var x : integer;
x := 0;
procedure p(y : integer);
begin;
y := 1;
x := 0;
end;
p(x);
```

call by value result

```
var x : integer;
x := 0;
procedure p(y : integer);
begin;
y := 1;
x := 0;
end;
p(x);
```

copy-in phase

p(0)
y = 0
py = &y

execute the body

y = 1

x = 0

copy out phase

value of the formal y is copied back to the l-value saved.

py = 1

therefore x = 1

call by reference

```
var x : integer;
x := 0;
procedure p(y : integer);
begin;
y := 1;
 x := 0;
end;
p(x);
```

x = 0

call by value:

```
var x : integer;
x := 0;
procedure p(y : integer);
begin;
y := 1;
 x := 0;
end;
p(x);
```

x = 0

ex)

```
var x : integer;
x := 1;
procedure p(y : integer);
begin;
y := 3;
x := 4;
end;
p(x);
```

call by value

x = 1

call by reference

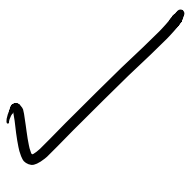
x = 4

call by value result

x = 3

instance monad result

return n = OK n
(>>=) OK n f = f n
(>>=) Error s f = Error s



a → m a

(>>=) m a → (a → m b) → m b

check OK n f = f n
check Error s f = Error s

interp
without check/monad :

```
interp NUM n = OK n
interp PLUS x y = case interp x of
                    OK n → case interp y of
                             OK m → OK (n+m)
                    Error s → Error s
```

....

Using check

```
interp NUM n = OK n
interp PLUS x y = interp x 'check'
                  \n → interp y 'check'
                  \m → OK n+m
```

With monad

```
interp NUM n = OK
```