

CSE 130: Programming Languages

Environments & Closures

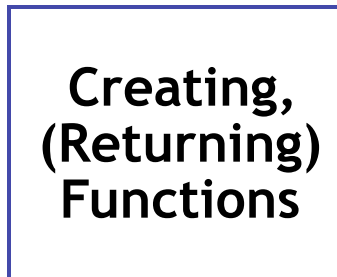
Ranjit Jhala
UC San Diego



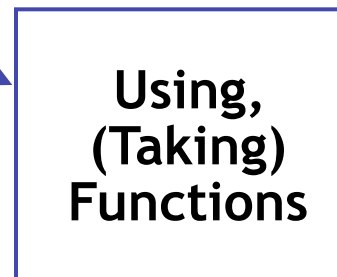
Recap: Functions as “first-class” values

- Arguments, return values, bindings ...
- What are the benefits ?

*Parameterized,
similar functions
(e.g. Testers)*



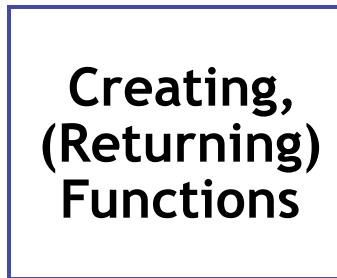
*Iterator, Accumul,
Reuse computation
pattern w/o
exposing local info*



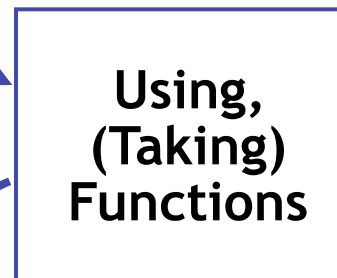
Functions are “first-class” values

- Arguments, return values, bindings ...
- What are the benefits ?

*Parameterized,
similar functions
(e.g. Testers)*



*Iterator, Accumul,
Reuse computation
pattern w/o
exposing local info*

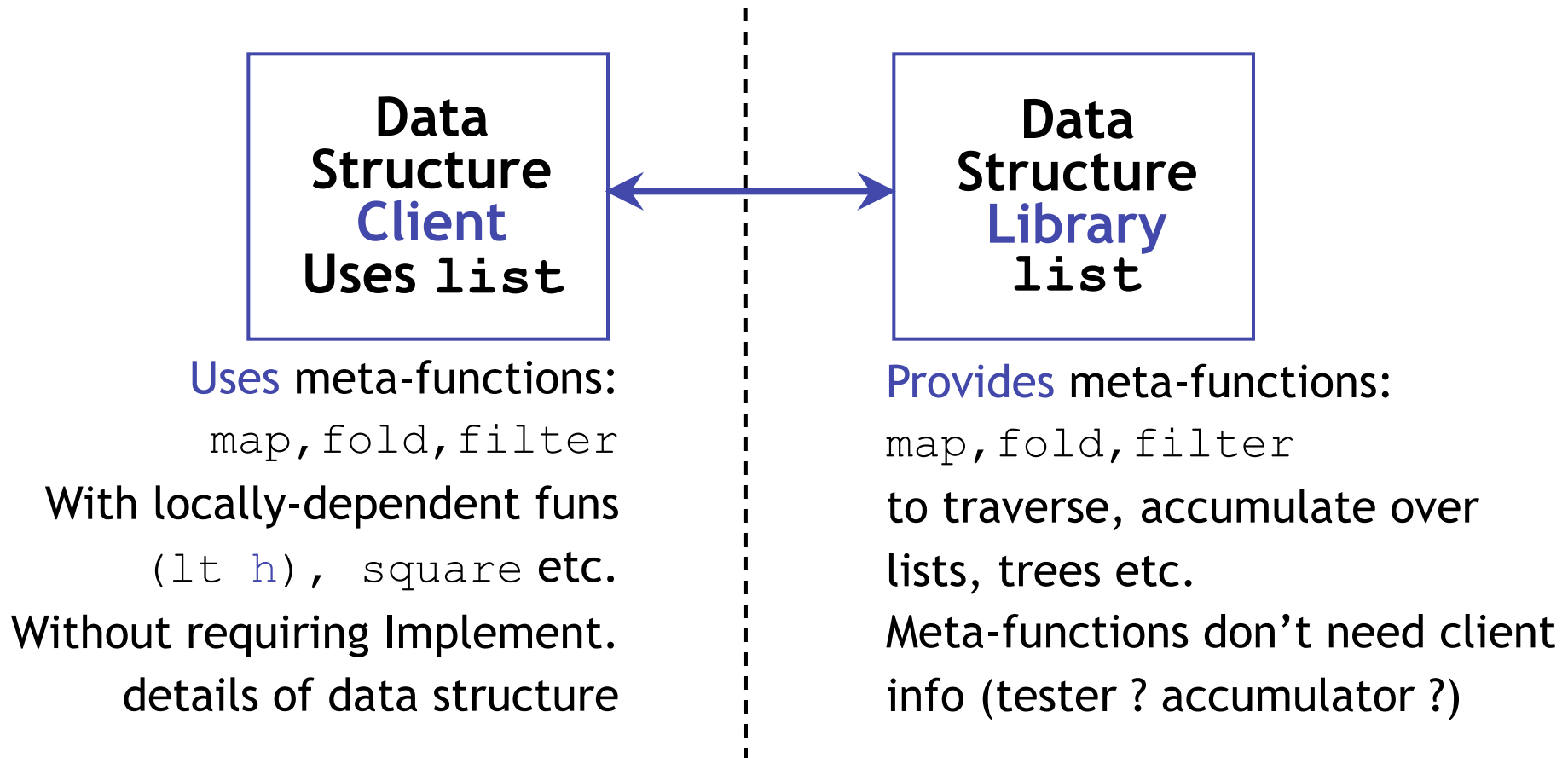


*Compose Functions:
Flexible way to build
Complex functions
from primitives.*

Funcs taking/returning funcs

Higher-order funcs enable **modular** code

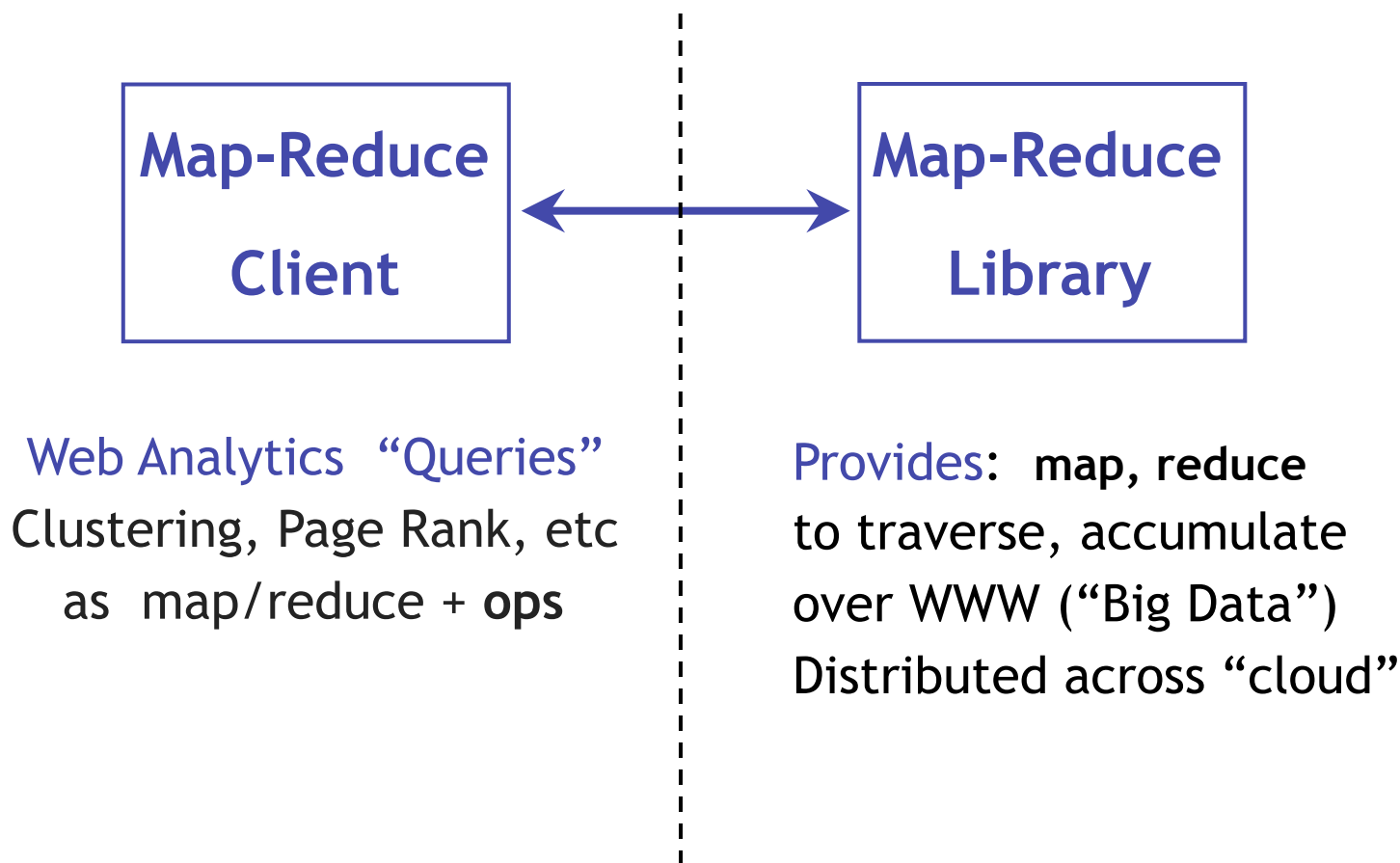
- Each part only needs **local** information



“Map-Reduce” et al.

Higher-order funcs enable **modular** code

- Each part only needs **local** information

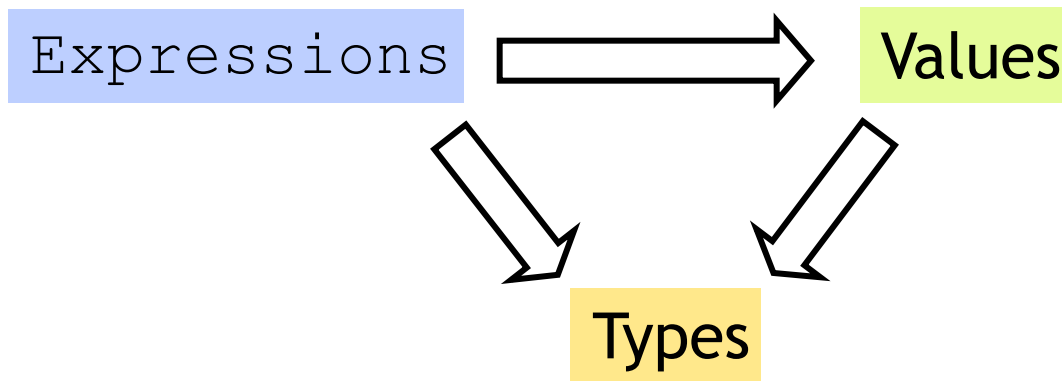


Higher Order Functions Are Awesome...

Higher Order Functions

..but how do they work

Next: Environments & Functions



Lets start with the humble variable...

Variables and Bindings

Q: How to use variables in ML ?

Q: How to “assign” to a variable ?

```
# let x = 2+2;;  
val x : int = 4
```

```
let x = e;;
```

“Bind value of expr *e* to variable *x*”

Variables and Bindings

```
# let x = 2+2;;  
val x : int = 4  
# let y = x * x * x;;  
val y : int = 64  
# let z = [x;y;x+y];;  
val z : int list = [4;64;68]
```

Later expressions can **use** x

- **Most recent** “bound” value used for evaluation

Sounds like C/Java ?

NO!

Environments (“Phone Book”)

How ML deals with variables

- Variables = “names”
- Values = “phone number”

W, Queensbury	01274 881373	P	10 Prospect Vw,
Road, Bradford	01274 603920	PJ	22 Shelf Moor Rd
l, Brighouse	01484 722933	R	5 Arnold Royd, B
ster Rd, Linthwaite	01484 844586	R	1041 Mancheste
, BD6	01274 679404	R	9 St Pauls Gro, B
Slaithwaite	01484 843163	R	10 Varley Rd, Sla
d, Wyke	01274 675753	R	156 Wilson Rd, V
Slaithwaite	01484 843681	Robert	1 Wood St, Sla
, Queensbury	01274 818683	RA	2 Cheriton Dv, Q
larsden	01484 844450	RA	5 Dirker Dv, Mars
ott, Plains, Marsden	01484 844996	RB	Dirker Bank Cott,
layton	01274 816057	RC	16 Holts La, Clay
ie, Linthwaite	01484 846885	RD	46 Stones Lane, l
Gro, Cross Roads	01535 643681	RW	37 Laburnum Gro
l, Todmorden	01706 818413	S	160 Bacup Rd, To
Av, Bradford	01274 672644	S	35 Markfield Av,
Dv, Queensbury	01274 818887	SP	9 Brambling Dv,
l, Pellon	01422 259543	T	22b Albert Vw, Pe
Rd, Sowerby Bdge	01422 839907	T	13 Industrial Rd,
l, Beechwood	01422 831577	TE	39 Whitley Av, Be
t, Clayton	01274 882408	V	17 Gregory Ct, Cl
l, Brighouse	01484 714532	W	43 Bolehill Pk, Bri

...	...
x	4 : int
y	64 : int
z	[4;64;68] : int list
x	8 : int

Environments and Evaluation

ML begins in a “top-level” environment

- Some names bound (e.g. +, -, print_string...)

```
let x = e;;
```

ML program = Sequence of variable bindings

Program evaluated by evaluating bindings in order

1. Evaluate expr **e** in current env to get value **v** : **t**
2. Extend env to bind **x** to **v** : **t**

(Repeat with next binding)

Environments

“Phone book”

- Variables = “names”
- Values = “phone number”

1. Evaluate:

Find and use most recent value of variable

2. Extend:

Add new binding at end of “phone book”

Example

```
# let x = 2+2;;  
val x : int = 4
```

```
# let y = x * x * x;;  
val y : int = 64
```

```
# let z = [x;y;x+y];;  
val z : int list = [4;64;68]
```

```
# let x = x + x ;;  
val x : int = 8
```

...	...
-----	-----

...	...
x	4 : int

...	...
x	4 : int
y	64 : int

...	...
x	4 : int
y	64 : int
z	[4;64;68] : int list

...	...
x	4 : int
y	64 : int
z	[4;64;68] : int list
x	8 : int

New binding!

Environments

1. **Evaluate**: Use **most recent** bound value of var
2. **Extend**: Add **new** binding at end

How is it different from C/Java's "store" ?


```
# let x = 2+2;;  
val x : int = 4
```

```
# let f = fun y -> x + y;  
val f : int -> int = fn
```

```
# let x = x + x ;  
val x : int = 8
```

```
# f 0;  
val it : int = 4
```

...	...
x	4 : int

...	...
x	4 : int
f	fn <code,  >: int->int

New binding:

- No change or mutation
- Old binding frozen in **f**

Environments

1. **Evaluate**: Use **most recent** bound value of var
2. **Extend**: Add **new** binding at end

How is it different from C/Java's “store” ?


```
# let x = 2+2;  
val : int x = 4
```


```
# let f = fun y -> x + y;  
val f : int -> int = fn
```

```
# let x = x + x ;  
val x : int = 8;
```

```
# f 0;  
val it : int = 4
```

...	...
x	4 : int

...	...
x	4 : int
f	fn <code,  >: int->int

...	...
x	4 : int
f	fn <code,  >: int->int
x	8 : int


Environments

1. **Evaluate**: Use **most recent** bound value of var
2. **Extend**: Add **new** binding at end

How is it different from C/Java's “store” ?

```
# let x = 2+2;  
val x : int = 4  
  
# let f = fun y -> x + y;;  
val f : int -> int = fn  
  
# let x = x + x ;  
val x : int = 8  
  
# f 0;  
val it : int = 4
```

Binding used to eval (**f** ...)

...	...
x	4 : int
f	fn <code,  >: int->int
x	8 : int

Binding for subsequent **x**

Cannot change the world

Cannot “assign” to variables

- Can extend the env by adding a fresh binding
- Does **not** affect previous uses of variable


Environment at fun declaration **frozen inside fun** “value”

- Frozen env used to evaluate **application** ($\mathbf{f} \ e$)

Q: Why is this a good thing ?

```
# let x = 2+2;;  
val x : int = 4  
# let f = fun y -> x + y;;  
val f : int -> int = fn  
# let x = x + x ;;  
val x : int = 8;  
# f 0;;  
val it : int = 4
```

Binding used to eval ($\mathbf{f} \ \dots$)

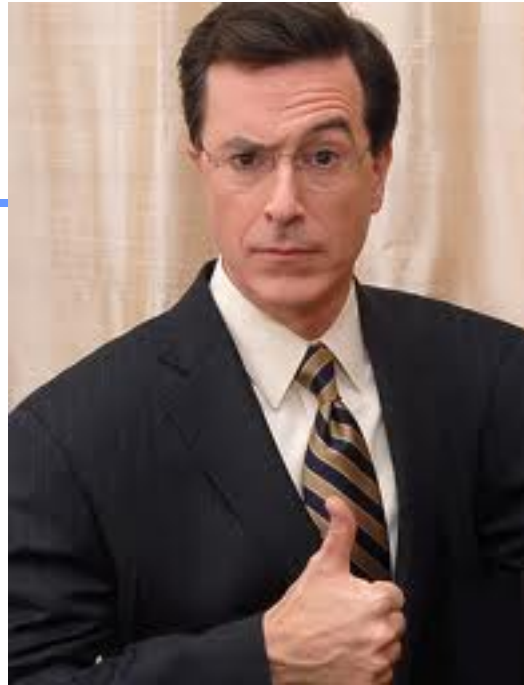
...	...
x	4 : int
f	fn <code,  >: int->int
x	8 : int

Binding for subsequent **x**

Cannot change the world

Q: Why is this a good thing ?

A: Function behavior frozen at declaration



Immutability: The Colbert Principle

“A function behaves the same way on Wednesday, as it behaved on Monday, *no matter what happened on Tuesday!*”

Cannot change the world

Q: Why is this a good thing ?

A: Function behavior frozen at declaration

- Nothing entered afterwards affects function
- Same inputs always produce same outputs
 - Localizes debugging
 - Localizes reasoning about the program
 - No “sharing” means no evil aliasing

Examples of no sharing

Remember: No addresses, no sharing.

- Each variable is bound to a “fresh instance” of a value

Tuples, Lists ...

- Efficient implementation without sharing ?
 - There is sharing and pointers but hidden from you
- Compiler’s job is to optimize code
 - Efficiently implement these “no-sharing” semantics
- Your job is to use the simplified semantics
 - Write correct, cleaner, readable, extendable systems

Q: What is the value of res ?

```
let f = fun x -> 1;;  
let f = fun x -> if x < 2 then 1 else (x * f(x-1));;  
let res = f 5;;
```

- (a) 120
- (b) 60
- (c) 20
- (d) 5
- (e) 1

Function bindings

Functions are values, can bind using **val**

```
let fname = fun x -> e ;;
```

Problem: Can't define recursive functions !

- fname is bound **after** computing rhs value
- no (or “old”) binding for occurrences of fname inside e

```
let rec fname x = e ;;
```

Occurrences of fname inside e bound to “this” definition

```
let rec fac x = if x<=1 then 1 else x*fac (x-1)
```


Q: What is the value of res ?

```
let y = let x = 10 in  
      x + x ;;  
  
let res = (x, y) ;;
```

- (a) Unbound Var Error
- (b) (10, 20)
- (c) (10, 10)
- (d) Type Error

Local bindings

So far: bindings that remain until a re-binding (“global”)

Local, “temporary” variables are useful inside functions

- Avoid repeating computations
- Make functions more readable

Let-in is an expression!

```
let x = e1 in
  e2
;;
```

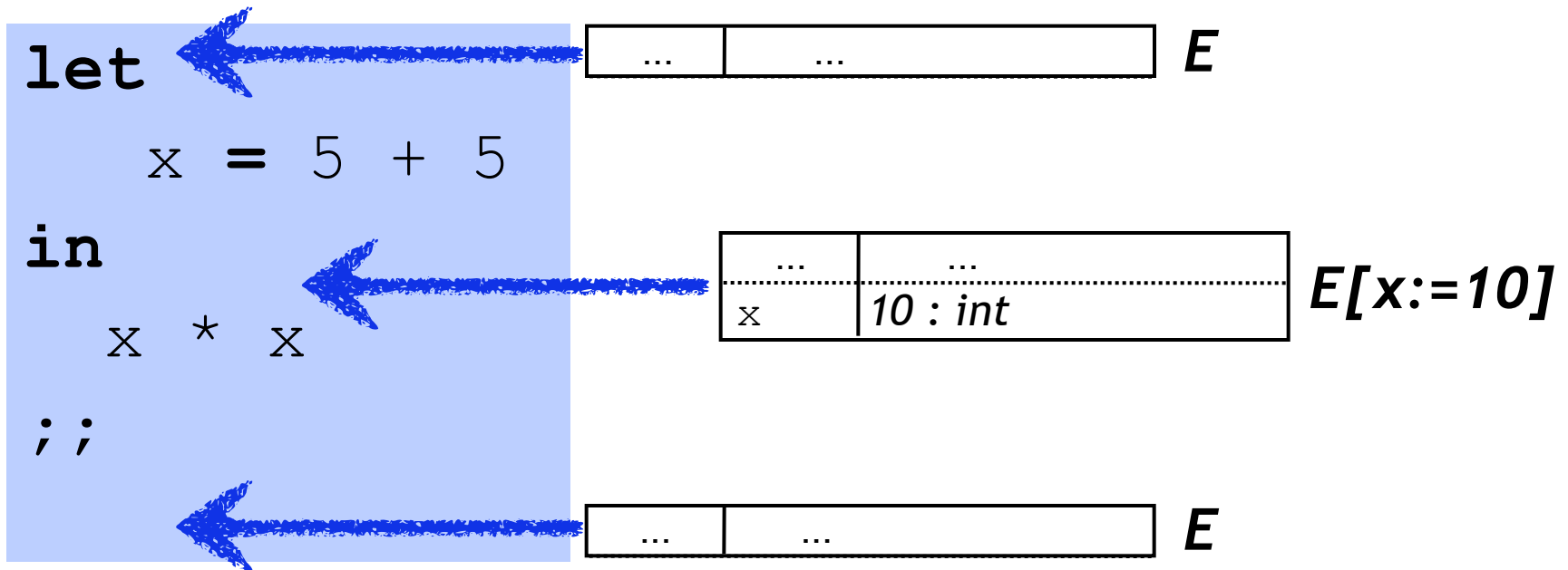
Evaluating let-in in env E :

1. Evaluate expr $e1$ in env E to get value $v : t$
2. Use extended $E [x \mapsto v : t]$ (only) to evaluate $e2$

Local bindings

Evaluating let-in in env E :

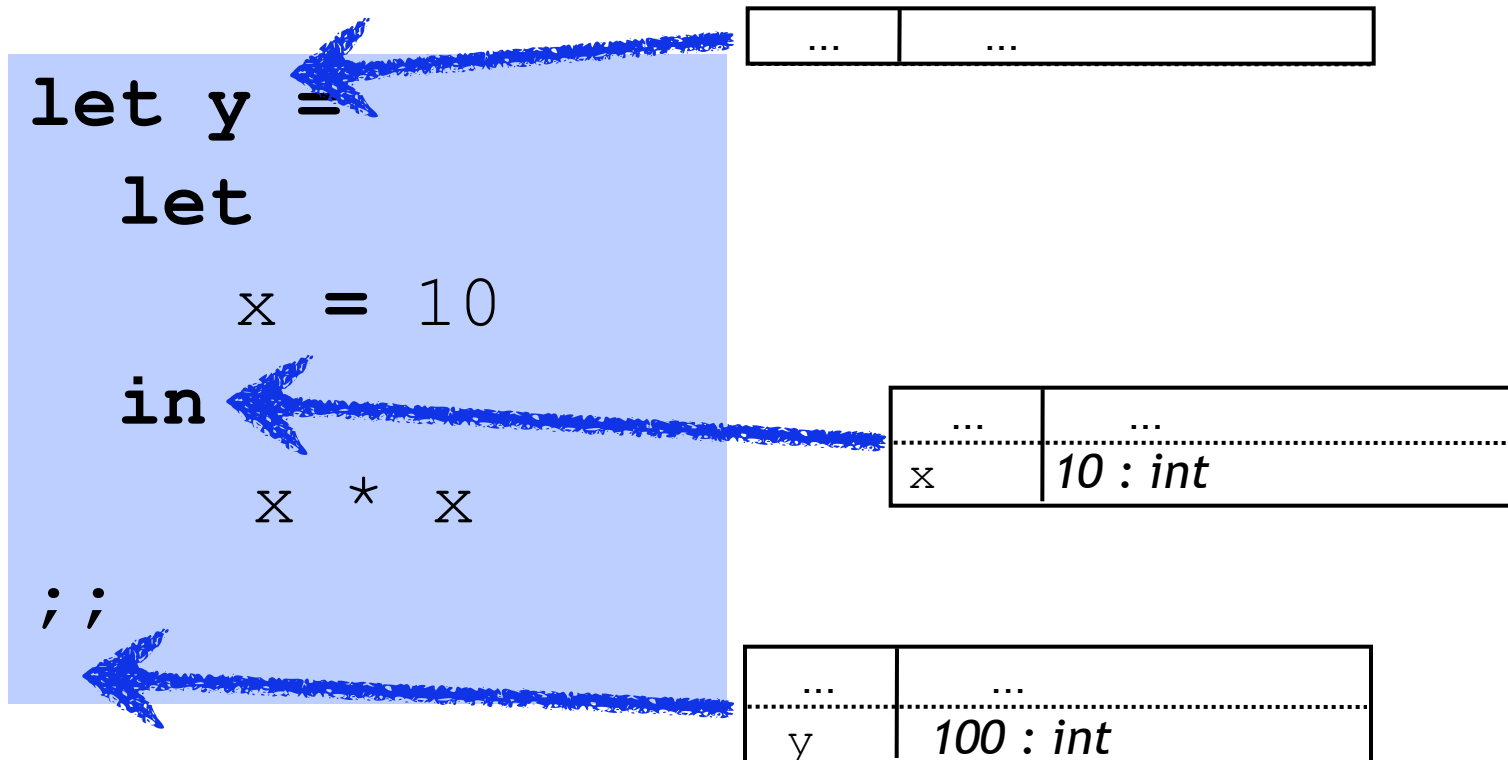
1. Evaluate expr e_1 in env E to get value $v : t$
2. Use extended $E [x \mapsto v : t]$ to evaluate e_2



Let-in is an expression!

Evaluating let-in in env E :

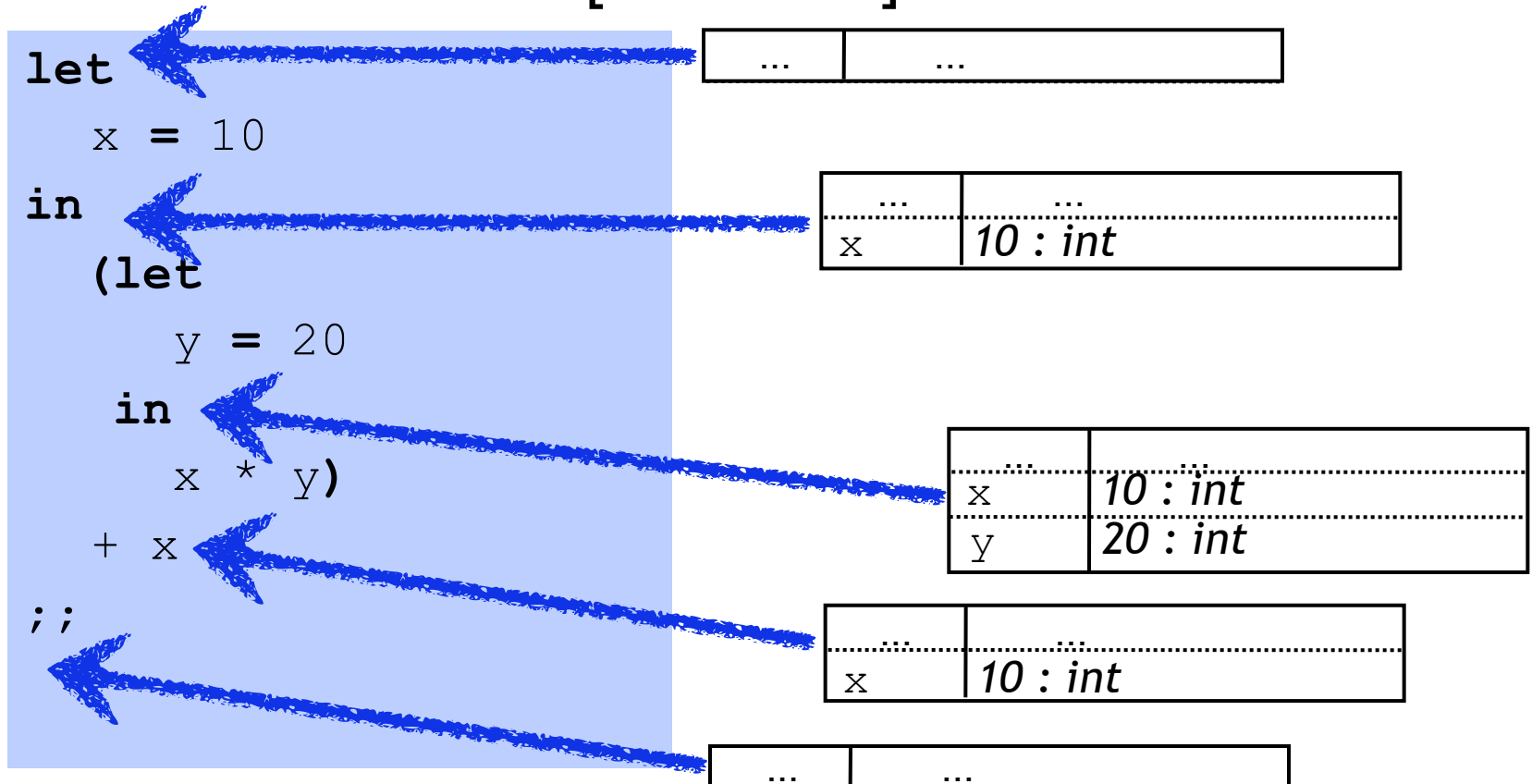
1. Evaluate expr $e1$ in env E to get value $v : t$
2. Use extended $E [x \mapsto v : t]$ to evaluate $e2$



Nested bindings

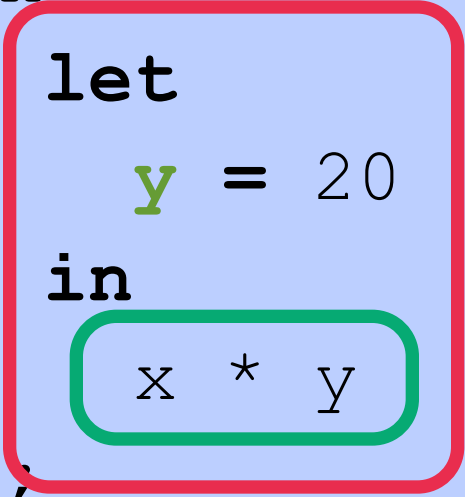
Evaluating let-in in env E :

1. Evaluate expr $e1$ in env E to get value $v : t$
2. Use extended $E [x \mapsto v : t]$ to evaluate $e2$



Nested bindings

```
let
  x = 10
in
  let
    y = 20
  in
    x * y
;;
```



BAD Formatting

```
let x = 10 in
let y = 20 in
  x * y
;;
```

GOOD Formatting

Example

```
let rec filter f xs =  
  match xs with  
  | []      -> []  
  | x::xs'  -> let ys  = if f x then [x] else [] in  
                let ys' = filter f xs      in  
                ys @ ys'
```

Recap 1: Variables are names for values

- Environment: dictionary/phonebook
- Most recent binding used
- **Entries never change**
- New entries added

Recap 2: Big Exprs With Local Bindings

- `let-in` expression
- Variable “in-scope” `in`-expression
- Outside, variable not “in-scope”

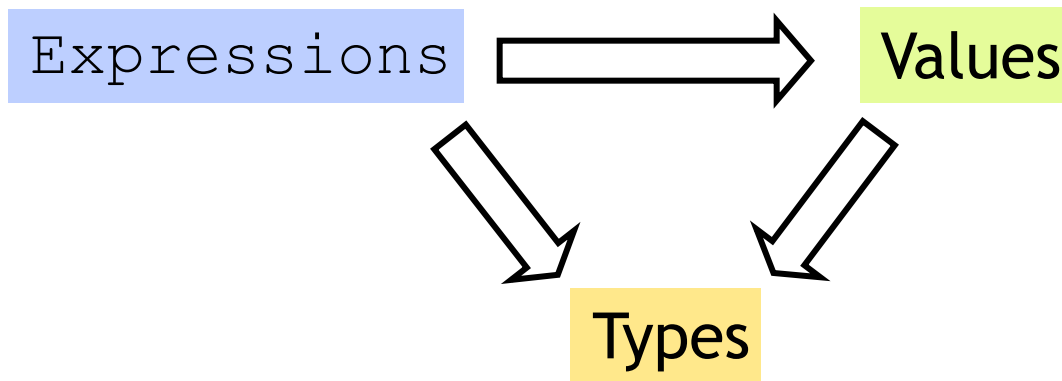
Recap 3: Env Frozen at Func Definition

- Re-binding vars cannot change function
- Identical I/O behavior at every call
- Predictable code, localized debugging

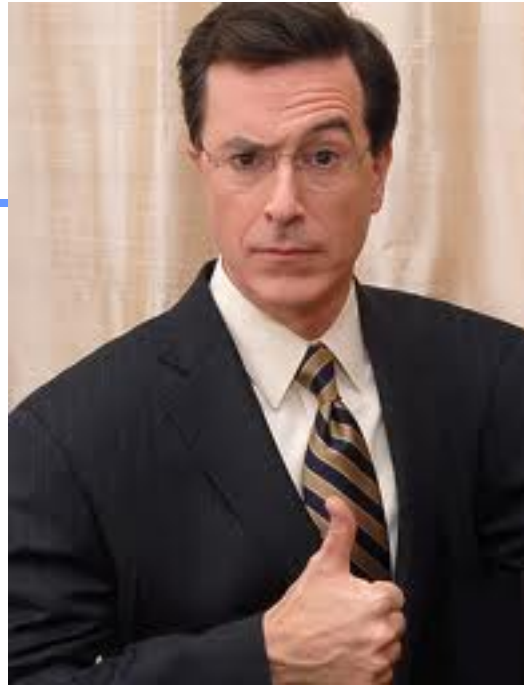
Static/Lexical Scoping

- For each occurrence of a variable,
A unique place where variable was defined!
 - Most recent binding in environment
- Static/Lexical: Determined from program text
 - Without executing the program
- Very useful for readability, debugging:
 - Don't have to figure out “where” a variable got assigned
 - Unique, statically known definition for each occurrence

Next: Functions



Q: What's the **value** of a **function** ?



Immutability: The Colbert Principle

“A function behaves the same way on Wednesday, as it behaved on Monday,
no matter what happened on Tuesday!”

Functions

Expressions

Two ways of writing function expressions:

1. Anonymous functions:

Parameter
(formal)

Body
Expr

```
let fname = fun x -> e
```

2. Named functions:

Parameter
(formal)

Body
Expr

```
let fname x = e
```

Function Application

Expressions

Application: fancy word for “call”

$(e1\ e2)$

- Function value $e1$
- Argument $e2$
- “apply” argument $e2$ to function value $e1$

Functions

Type

The type of any function is:

- $T1$: the type of the “input”
- $T2$: the type of the “output”

$T1 \rightarrow T2$

let fname = **fun**

$x \rightarrow e$

$T1 \rightarrow T2$

let fname $x = e$

$T1 \rightarrow T2$

Functions

Type

The type of any function is:

- $T1$: the type of the “input”
- $T2$: the type of the “output”

$T1 \rightarrow T2$

$T1$, $T2$ can be *any* types, including functions!

Whats an example of ?

- $int \rightarrow int$
- $int * int \rightarrow bool$
- $(int \rightarrow int) \rightarrow (int \rightarrow int)$

Type of function application

Application: fancy word for “call”

$(e1\ e2)$

- “apply” argument $e2$ to function value $e1$

$$\frac{e1 : T1 \rightarrow T2 \quad e2 : T1}{(e1\ e2) : T2}$$

- Argument must have same type as “input” $T1$
- Result has the same type as “output” $T2$

Functions

Values

Two questions about function values:

What is the *value*:

1. ... of a function ?

fun x -> e

2. ... of a function “application” (call) ?

(e1 e2)

Values of function = “Closure”

Two questions about function values:

What is the **value**:

1. ... of a function ?

```
fun x -> e
```

Closure =

Code of Fun. (**formal x + body e**)

+ Environment at Fun. Definition

Values of function = “Closure”

Two questions about function values:

What is the **value**:

1. ... of a function ?

```
fun x -> e
```

Closure =

Code of Fun. (**formal x + body e**)

+ Environment at Fun. Definition

Q: Which vars in env. of f ?

```
let x      = 2 + 2 ;;  
let f y    = x + y ;;  
let z      = x + 1 ;;
```


- (a) x
- (b) y
- (c) x y
- (d) x y z
- (e) None

Values of functions: Closures

- Function value = “Closure”
 - <code + environment at definition>
- Body not evaluated until application
 - But type-checking when function is defined

```
# let x = 2+2;;
val x : int = 4
# let f = fun y -> x + y;;
val f : int -> int = fn
# let x = x + x;;
val x : int = 8
# f 0;;
val it : int = 4
```

Binding used to eval (f ...)

x	4 : int
f	fn <code,  >: int->int
x	8 : int

Binding for subsequent x

Q: Vars in closure-env of f ?

```
let a = 20 ; ;  
  
let f x =  
  let y = x + 1 in  
  let g z = y + z in  
    a + (g x)  
; ;
```

- (a) a y
- (b) a
- (c) y
- (d) z
- (e) y z

Free vs. Bound Variables

```
let a = 20;;
```

```
let f x =  
  let y = 1 in  
  let g z = y + z in  
    a + (g x)  
;;
```

```
f 0;;
```

(e1 e2)

Environment frozen with function

Used to evaluate fun application

Which vars needed in frozen env?

Free vs. Bound Variables

```
let a = 20;;

let f x =
  let y = 1 in
  let g z = y + z in
    a + (g x)
;;

f 0;;
```

Inside a function:

A “bound” occurrence:

1. Formal variable
2. Variable bound in `let-in`

`x`, `y`, `z` are “bound” inside `f`

A “free” occurrence:

- Non-bound occurrence

`a` is “free” inside `f`

Frozen Environment
needed for values of free vars

Q: Which vars are free in f ?

```
let a = 20 ; ;  
  
let f x =  
  let a = 1 in  
    let g z = a + z in  
      a + (g x)  
; ;
```

- (a) a
- (b) x
- (c) y
- (d) z
- (e) None

Free vs. Bound Variables

```
let a = 20;;
```

```
let f x =
```

```
  let a = 1 in
```

```
  let g z = a + z in
```

```
    a + (g x)
```

```
;;
```

```
f 0;
```

Inside a function:

A “bound” occurrence:

1. Formal variable
2. Variable bound in `let-in-end`

`x`, `a`, `z` are “bound” inside `f`

A “free” occurrence:

Not bound occurrence

`nothing` is “free” inside `f`

Where do bound-vars values come from?

```
let a = 20 ;;

let f x =
  let a = 1 in
  let g z = a + z in
    a + (g x)
  ;;

f 0;
```

Bound values determined when function is evaluated (“called”)

- Arguments
- Local variable bindings

Values of function application

Two questions about function values:

What is the *value*:

1. ... of a function ?

fun x -> e

2. ... of a function “application” (call) ? *(e1 e2)*

“apply” the argument *e2* to the (function) *e1*

Values of function application

Value of a function “application” (call) $(e1\ e2)$

1. Find **closure** of $e1$
2. Execute body of **closure** with param $e2$

Free values found in **closure-environment**

Bound values by executing **closure-body**

Values of function application

Value of a function “application” (call) $(e1\ e2)$

1. Evaluate $e1$ in current-env to get (**closure**)
= **code** (formal x + body e) + env E
2. Evaluate $e2$ in current-env to get (argument) $v2$
3. Evaluate body e in env E extended with $x := v2$

Q: What is the value of **res** ?

```
let x    = 1;;  
let y    = 10;;  
let f y  = x + y;;  
let x    = 2;;  
let y    = 3;;  
let res  = f (x + y) ;;
```

- (a) 4 (b) 5 (c) 6 (d) 11 (e) 12

Q: What is the value of **res** ?

```
let x = 1;;  
let y = 10;;  
let f y = x + y;;  
let x = 2;;  
let y = 3;;  
let res = f (x + y);;
```

```
f |-> formal := y  
      body  := x + y  
      env   := [x |-> 1]  
  
x |-> 2  
y |-> 3  
  
x + y ==> 5
```

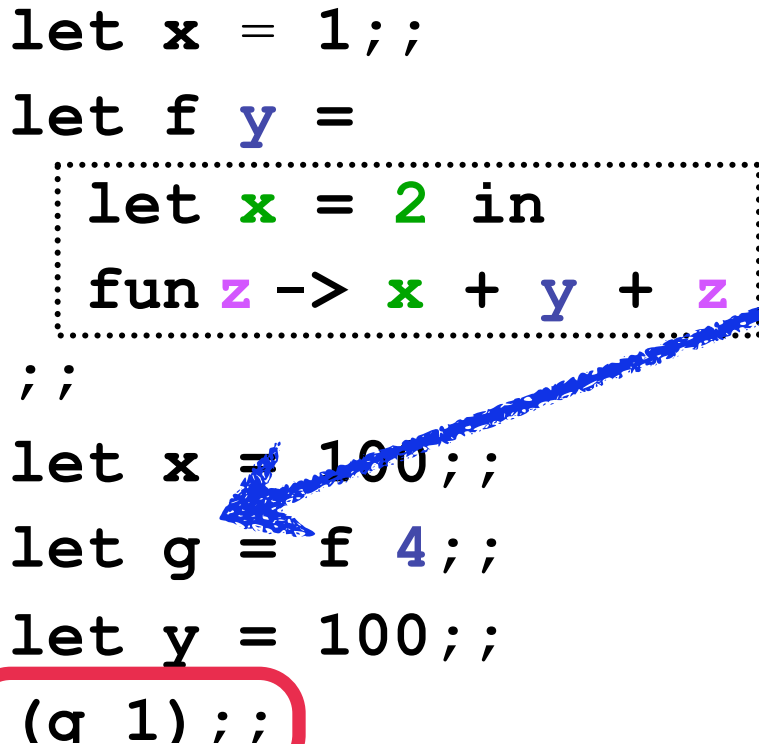
Application: $f \ (x + y)$

Eval **body** in **env** extended with **formal** |-> 5

Eval **x+y** in [**x** |-> 1, **y** |-> 5] ==> 6

Example

```
let x = 1;;  
let f y =  
  let x = 2 in  
  fun z -> x + y + z  
;;  
let x = 100;;  
let g = f 4;;  
let y = 100;;  
(g 1);;
```



Q: Closure value of g?

formal	z
body	x + y + z
env	[x ->2, y ->4]

Eval **body** in **env** extended with **formal|-> 1**

Eval **x+y+z** in **[x|->2, y|->4, z|->1]** **====> 7**

Q: What is the value of `res` ?

```
let f g =  
  let x = 0 in  
  g 2  
;;  
  
let x = 100;;  
  
let h y = x + y;;  
  
let res = f h;;
```

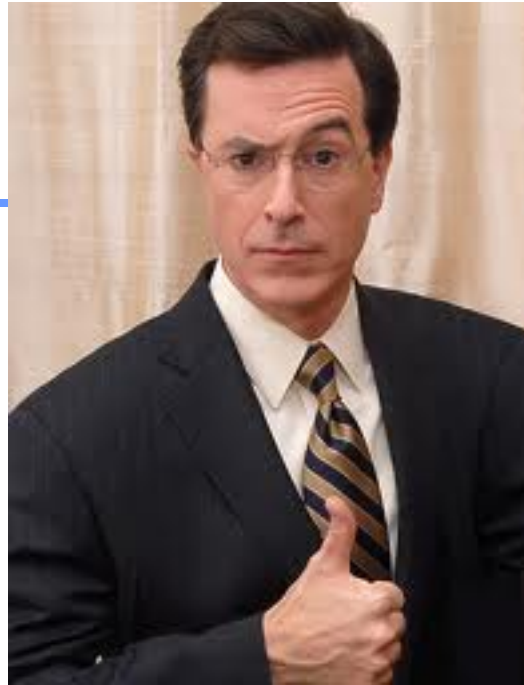
- (a) Syntax Error
- (b) 102
- (c) Type Error
- (d) 2
- (e) 100

Example 3

```
let f g =  
  let x = 0 in  
    g 2  
;;  
  
let x = 100;;  
  
let h y = x + y;;  
  
f h;;
```

Static/Lexical Scoping

- For each occurrence of a variable,
 - Unique place in program text where variable defined
 - Most recent binding in environment
- Static/Lexical: Determined from the program text
 - Without executing the program
- Very useful for readability, debugging:
 - Don't have to figure out “where” a variable got assigned
 - Unique, statically known definition for each occurrence



Immutability: The Colbert Principle

“A function behaves the same way on Wednesday, as it behaved on Monday, *no matter what happened on Tuesday!*”