HARVARD
John A. Paulson
School of Engineering
and Applied Sciences

CS153: Compilers
Lecture 12:
Closures and Environments

Stephen Chong

https://www.seas.harvard.edu/courses/cs153

Announcements

- Project 4 out
    - Due Thursday Oct 25 (14 days)
- Project 5 released today (probably)
    - Due sometime in the future
- Final exam date: Wednesday December 12, 9am
- CS Nights: Mondays 8pm, MD119

Today

- Nested functions
  - Substitution semantics
  - Environment semantics and closures
- Closure conversion
- Implementing environments and variables
  - DeBruijn indices
  - Nested environments vs flat environments

“Functional” Languages

- In functional languages, functions are first-class values
  - In addition to being called, functions can be passed as arguments (e.g., `map`), returned as results (e.g., `compose`), and stored in data structures
  - Just like other values! (`int`, `string`, ...)
- Scheme, Racket,, SML, OCaml, Haskell, Clojure, Javascript, …
- How do we represent a function value?
- Code pointer?

Function example

```
let add = fun x -> (fun y -> y+x)
let inc = add 1 (* = fun y -> y + 1 *)
let dec = add -1 (* = fun y -> y + -1 *)

let compose = fun f -> fun g -> fun x -> f(g x)
let id = compose inc dec
 (* = fun x -> inc(dec x) *)
 (* = fun x -> (fun y -> y+1)((fun y -> y-1) x) *)
 (* = fun x -> (fun y -> y+1)(x-1)) *)
 (* = fun x -> (x-1)+1 *)
```

Nested Functions

```
let add = fun x -> (fun y -> y+x)
let inc = add 1 (* = fun y -> y + 1 *)
let dec = add -1 (* = fun y -> y + -1 *)
```

- Consider `add 1`
- After calling `add`, we can't throw away its argument `x` (or any local variables that add might use) because `x` is used by the **nested function** `fun y -> y+x`

Making Sense of Nested Functions

- Let's consider what are the right semantics for nested functions
  - We will look at a simple semantics first, and then get to an equivalent semantics that we can implement efficiently

Substitution-Based Semantics

```
type exp = Int of int | Plus of exp*exp |
    Var of var | Lambda of var*exp | App of exp*exp

let rec eval (e:exp) =
  match e with
  | Int i -> Int i
  | Plus(e1,e2) ->
      (match eval e1, eval e2 with
         | Int i,Int j -> Int(i+j))
  | Var x -> error ("Unbound variable "^x)
  | Lambda(x,e) -> Lambda(x,e)
  | App(e1,e2) ->
      (match eval e1, eval e2 with
         (Lambda(x,e),v) ->
    eval (subst v x e)))
```

Replace formal argument **x** with actual argument **v**

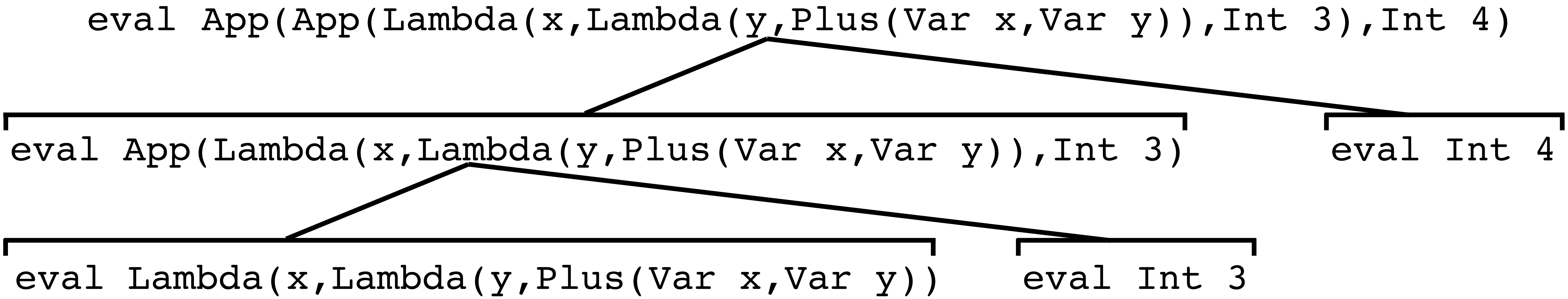
Substitution-Based Semantics

```
let rec subst (v:exp) (x:var) (e:exp) =
  match e with
   | Int i -> Int i
   | Plus(e1,e2) -> Plus(subst v x e1, subst v x e2)
   | Var y -> if y = x then v else Var y
   | Lambda(y,e') ->
       if y = x then Lambda(y,e')
                else Lambda(y,subst v x e')
   | App(e1,e2) -> App(subst v x e1, subst v x e2)
```

Slight simplification: assumes that all variable names in program are distinct.

Example

```
((fun x -> fun y -> x + y) 3) 4
```



```
eval App(App(Lambda(x,Lambda(y,Plus(Var x,Var y)),Int 3),Int 4)
```

```
eval App(Lambda(x,Lambda(y,Plus(Var x,Var y)),Int 3)
```

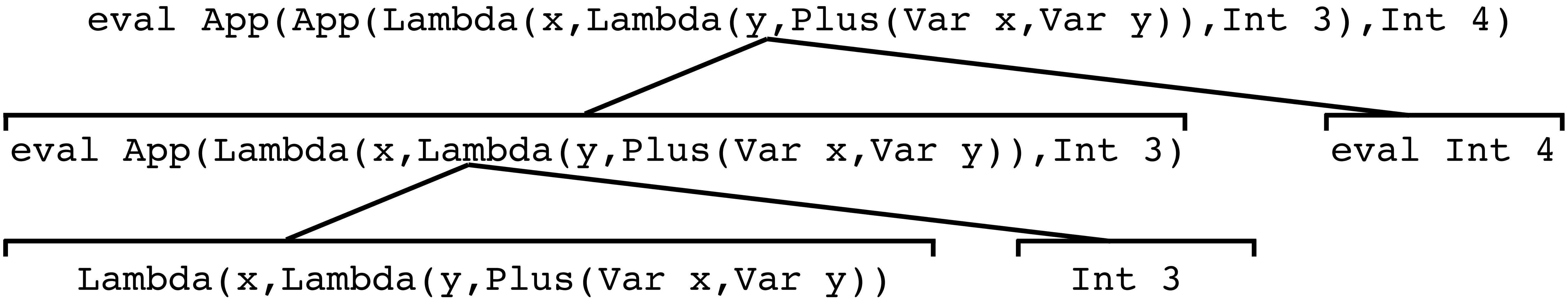
```
eval Int 4
```

```
eval Lambda(x,Lambda(y,Plus(Var x,Var y))
```

```
eval Int 3
```

Example

`((fun x -> fun y -> x + y) 3) 4`



`eval App(App(Lambda(x,Lambda(y,Plus(Var x,Var y)),Int 3),Int 4)`

`eval App(Lambda(x,Lambda(y,Plus(Var x,Var y)),Int 3)`

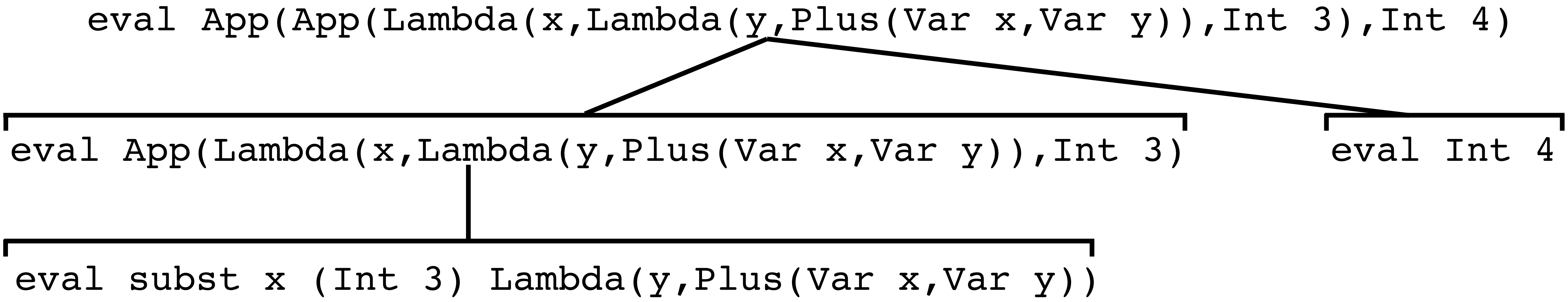
`eval Int 4`

`Lambda(x,Lambda(y,Plus(Var x,Var y))`

`Int 3`

Example

```
((fun x -> fun y -> x + y) 3) 4
```



```
eval App(App(Lambda(x,Lambda(y,Plus(Var x,Var y)),Int 3),Int 4)
```

```
eval App(Lambda(x,Lambda(y,Plus(Var x,Var y)),Int 3)
```

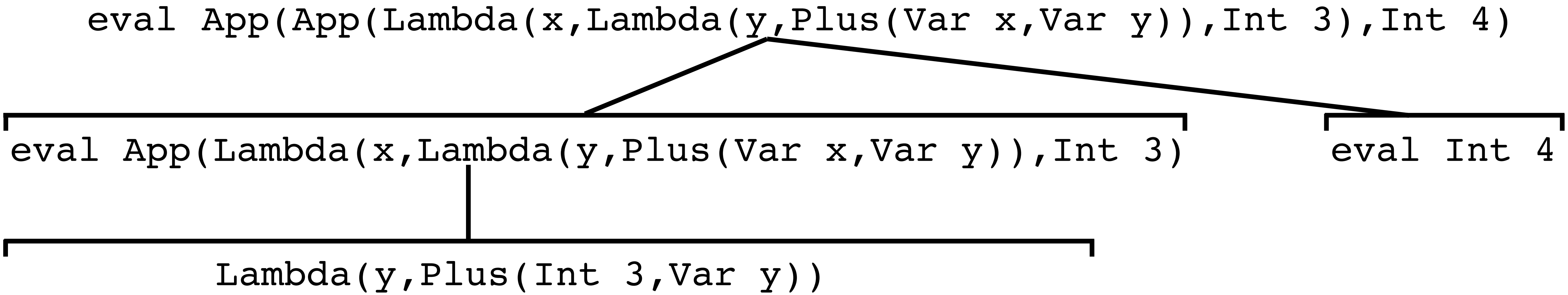
```
eval Int 4
```

```
eval subst x (Int 3) Lambda(y,Plus(Var x,Var y))
```

```
eval  Lambda(y,Plus(Int 3,Var y))
```

Example

```
((fun x -> fun y -> x + y) 3) 4
```



```
eval App(App(Lambda(x,Lambda(y,Plus(Var x,Var y)),Int 3),Int 4)
```

```
eval App(Lambda(x,Lambda(y,Plus(Var x,Var y)),Int 3)
```

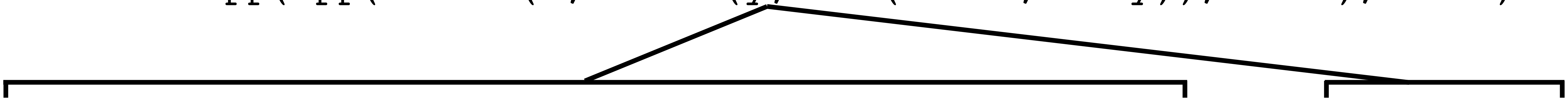
```
eval Int 4
```

```
Lambda(y,Plus(Int 3,Var y))
```

Example

```
((fun x -> fun y -> x + y) 3) 4
```

```
eval App(App(Lambda(x,Lambda(y,Plus(Var x,Var y)),Int 3),Int 4)
```



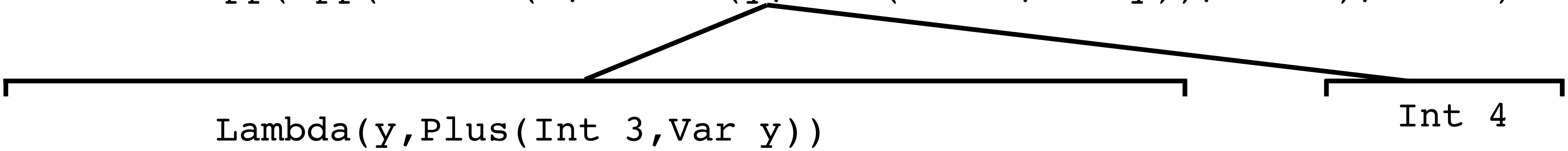
```
Lambda(y,Plus(Int 3,Var y))
```

```
eval Int 4
```

Example

```
((fun x -> fun y -> x + y) 3) 4
```

```
eval App(App(Lambda(x,Lambda(y,Plus(Var x,Var y)),Int 3),Int 4)
```



```
Lambda(y,Plus(Int 3,Var y))
```

```
Int 4
```

Example

```
((fun x -> fun y -> x + y) 3) 4
```

```
eval App(App(Lambda(x,Lambda(y,Plus(Var x,Var y)),Int 3),Int 4)
```

```
eval subst y (Int 4) Plus(Int 3,Var y)
```

```
eval Plus(Int 3,Int 4)
```

Problems

- `subst` crawls over expression and replaces variable with value
- Then `eval` crawls over expression
- So `eval (subst v x e)` is not very efficient
- Why not do substitution at the same time as we do evaluation?
- Modify `eval` to use an **environment**: a map from variables to the values

First Attempt

```
type value = Int_v of int
type env = (string * value) list

let rec eval (e:exp) (env:env) : value =
  match e with
  | Int i -> Int_v i
  | Var x -> lookup env x
  | Lambda(x,e) -> Lambda(x,e)
  | App(e1,e2) ->
      (match eval e1 env, eval e2 env with
       | Lambda(x,e'), v -> eval e' ((x,v)::env))
```

- Doesn’t handle nested functions correctly!
- E.g., `(fun x -> fun y -> y+x) 1` evaluates to `fun y -> y+x`
- Don’t have binding for `x` when we eventually apply this function!

Second Attempt

```
type value = Int_v of int
type env = (string * value) list

let rec eval (e:exp) (env:env) : value =
  match e with
   | Int i -> Int_v i
   | Var x -> lookup env x
   | Lambda(x,e) -> Lambda(x,subst env e)
   | App(e1,e2) ->
       (match eval e1 env, eval e2 env with
        | Lambda(x,e’), v -> eval e’ ((x,v)::env))
```

- Need to replace free variables of nested functions using environment where nested function defined
- But now we are using a version of `subst` again...

Closures

- Instead of doing substitution on nested functions when we reach the lambda, we can instead make a promise to finish the substitution if the nested function is ever applied
- Instead of

  ```
  | Lambda(x,e’) -> Lambda(x,subst env e’)
  ```

  we will have, in essence,

  ```
  | Lambda(x,e’) -> Promise(env, Lambda(x, e’))
  ```

  - Called a **closure**
- Need to modify rule for application to expect environment

Closure-based Semantics

```
type value = Int_v of int
           | Closure_v of {env:env, body:var*exp}
and  env = (string * value) list

let rec eval (e:exp) (env:env) : value =
  match e with
   | Int i -> Int_v i
   | Var x -> lookup env x
   | Lambda(x,e) -> Closure_v{env=env, body=(x,e)}
   | App(e1,e2) ->
       (match eval e1 env, eval e2 env with
        | Closure_v{env=cenv, body=(x,e')}, v ->
                  eval e' ((x,v)::cenv))
```

So, How Do We Compile Closures?

- Represent function values (i.e., closures) as a pair of function pointer and environment
- Make all functions take environment as an additional argument
  - Access variables using environment

Closure conversion

- Can then move all function declarations to top level (i.e., no more nested functions!)

Lambda lifting

- E.g., `fun x -> (fun y -> y+x)` becomes, in C-like code:

```
closure *f1(env *env, int x) {
  env *e1 = extend(env,“x”,x);
  closure *c =
       malloc(sizeof(closure));
  c->env = e1; c->fn = &f2;
  return c;
}
```

```
int f2(env *env, int y) {
  env *e1 = extend(env,“y”,y);
  return lookup(e1, “y”)
           + lookup(e1, “x”);
}
```

Where Do Variables Live

- Variables used in outer function may be needed for nested function
  - e.g., variable **`x`** in example on previous slide
- So variables used by nested functions can't live on stack...
- Allocate record for all variables on heap
- Hey, this is kind of like an object!!
  - Object = struct for field values, plus pointer(s) to methods
  - Closure = environment plus pointer to code

Closure Conversion

- Converting function values into closures
  - Make all functions take explicit environment argument
  - Represent function values as pairs of environments and lambda terms
  - Access variables via environment
- E.g.,

```
fun x -> (fun y -> y+x)
```

becomes

```
fun env x ->
      let e’ = extend env “x” x in
      (e’, fun env y ->
            let e’ = extend env “y” y in
            (lookup e’ “y”)+(lookup e’ “x”))
```

Lambda Lifting

- After closure conversion, nested functions do not directly use variables from enclosing scope
- Can “lift” the lambda terms to top level functions!
- E.g.,

```
fun env x ->
          let e’ = extend env “x” x in
          (e’, fun env y ->
                  let e’ = extend env “y” y in
                  (lookup e’ “y”)+(lookup e’ “x”))
```

becomes

```
let f2 = fun env y ->
                  let e’ = extend env “y” y in
                  (lookup e’ “y”)+(lookup e’ “x”)
fun env x ->
          let e’ = extend env “x” x in
          (e’, f2)
```

Lambda Lifting

- E.g.,

```
fun env x ->
          let e’ = extend env “x” x in
          (e’, fun env y ->
                    let e’ = extend env “y” y in
                    (lookup e’ “y”)+(lookup e’ “x”))
```

becomes

```
let f2 = fun env y ->
                   let e’ = extend env “y” y in
                   (lookup e’ “y”)+(lookup e’ “x”)
fun env x ->
          let e’ = extend env “x” x in
          (e’, f2)
```

```
closure *f1(env *env, int x) {
  env *e1 = extend(env,“x”,x);
  closure *c =
       malloc(sizeof(closure));
  c->env = e1; c->fn = &f2;
  return c;
}
```

```
int f2(env *env, int y) {
  env *e1 = extend(env,“y”,y);
  return lookup(e1, “y”)
          + lookup(e1, “x”);
}
```

How Do We Compile Closures Efficiently?

- Don’t need to heap allocate all variables
  - Just the ones that “escape”, i.e., might be used by nested functions
- Implementation of environment and variables

DeBruijn Indices

- In our interpreter, we represented environments as lists of pairs of variables names and values
- Expensive string comparison when looking up variable! `lookup env x`

```
let rec lookup env x =
  match env with
  | ((y,v)::rest) ->
        if y = x then v else lookup rest
  | [] -> error “unbound variable”
```

- Instead of using strings to represent variables, we can use natural numbers
  - Number indicates lexical depth of variable

DeBruijn Indices

```
type exp = Int of int | Var of int
         | Lambda of exp | App of exp*exp
```

- Original program

```
fun x -> fun y -> fun z -> x + y + z
```

- Conceptually, can rename program variables

```
fun x2 -> fun x1 -> fun x0 -> x2 + x1 + x0
```

- Don't bother with variable names at all!

```
fun -> fun -> fun -> Var 2 + Var 1 + Var 0
```

  - Number of variable indicates lexical depth, 0 is innermost binder

Converting to DeBruijn Indices

```
type exp = Int of int | Var of int
         | Lambda of exp | App of exp*exp

let rec cvt (e:exp) (env:var->int): D.exp =
  match e with
  | Int i -> D.Int i
  | Var x -> D.Var (env x)
  | App(e1,e2) ->
      D.App(cvt e1 env,cvt e2 env)
  | Lambda(x,e) =>
      let new_env(y) =
            if y = x then 0 else (env y)+1
      in
         Lambda(cvt e new_env)
```

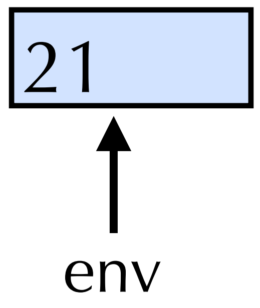
New Interpreter

```
type value = Int_v of int
           | Closure_v of {env:env, body:exp}
and  env = value list

let rec eval (e:exp) (env:env) : value =
  match e with
  | Int i -> Int_v i
  | Var x -> List.nth env x
  | Lambda e -> Closure_v{env=env, body=e}
  | App(e1,e2) ->
      (match eval e1 env, eval e2 env with
       | Closure_v{env=cenv, body=(x,e’)}, v ->
                  eval e’ v::cenv)
```

Representing Environments

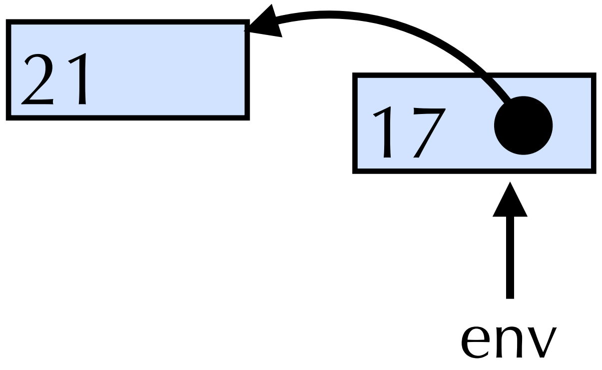
```
(((fun -> fun -> fun -> Var 2 + Var 1 + Var 0) 21) 17) 4
```



- Linked list (nested environments)

Representing Environments

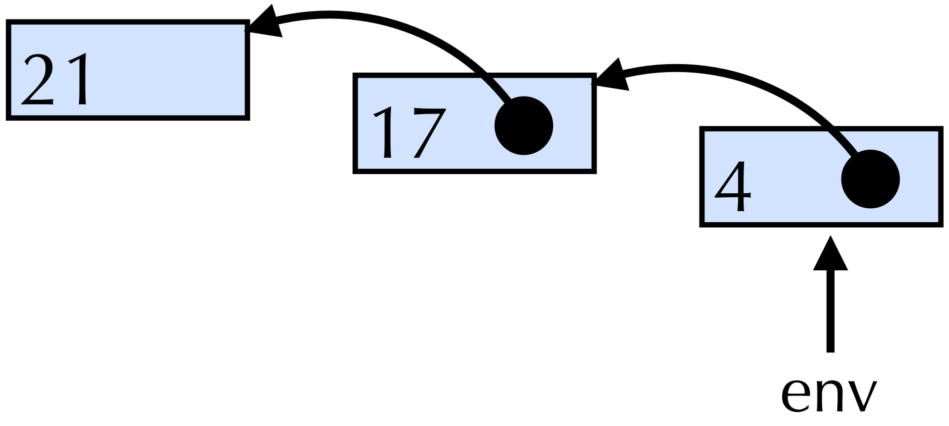
```
(((fun -> fun -> fun -> Var 2 + Var 1 + Var 0) 21) 17) 4
```



- Linked list (nested environments)

Representing Environments

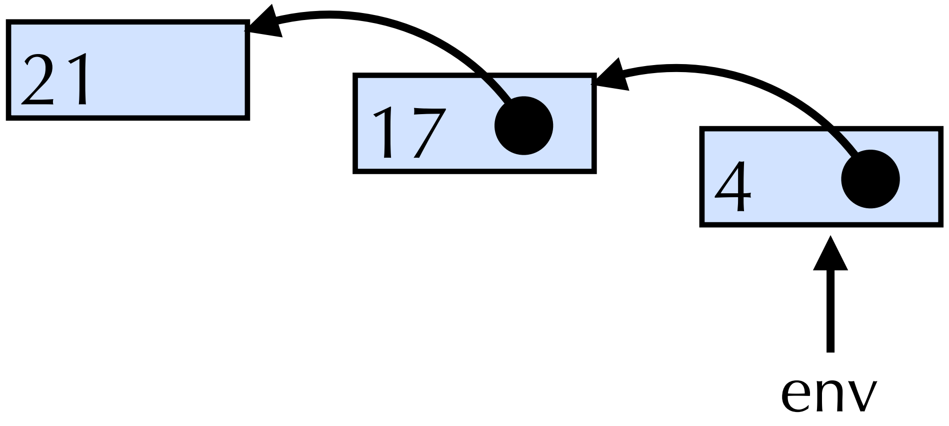
```
(((fun -> fun -> fun -> Var 2 + Var 1 + Var 0) 21) 17) 4
```



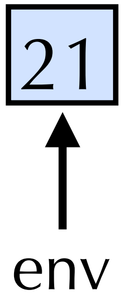
- Linked list (nested environments)

Representing Environments

```
(((fun -> fun -> fun -> Var 2 + Var 1 + Var 0) 21) 17) 4
```

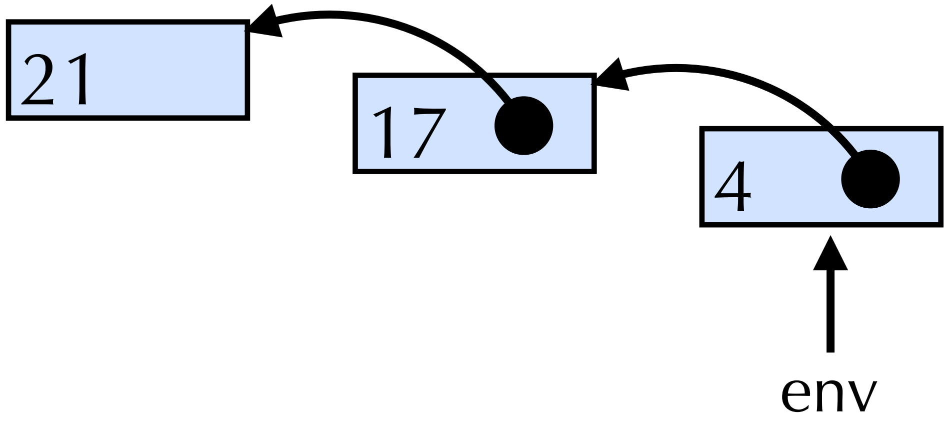


- Linked list (nested environments)
- Array (flat environment)

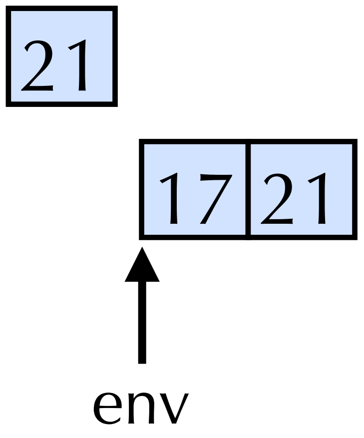


Representing Environments

```
(((fun -> fun -> fun -> Var 2 + Var 1 + Var 0) 21) 17) 4
```

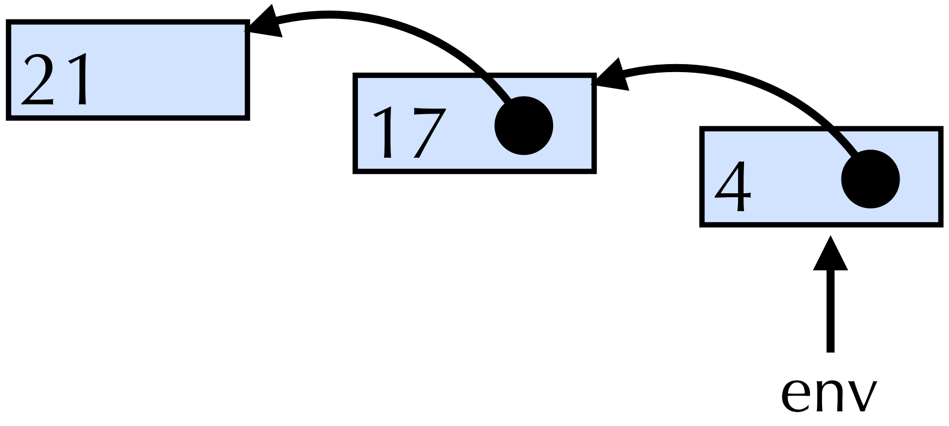


- Linked list (nested environments)
- Array (flat environment)

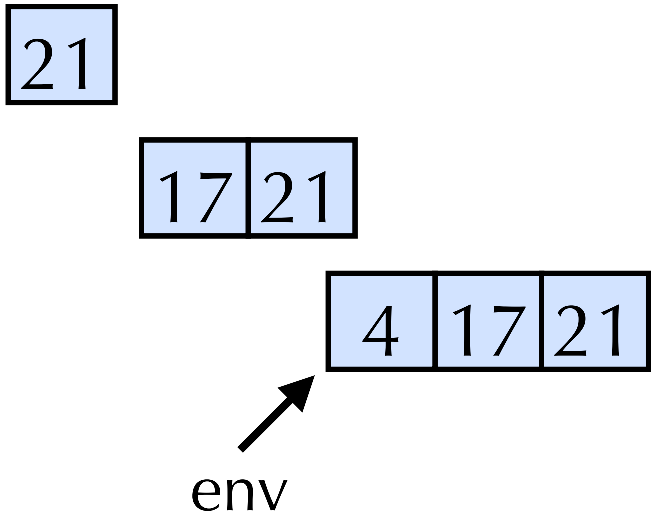


Representing Environments

```
(((fun -> fun -> fun -> Var 2 + Var 1 + Var 0) 21) 17) 4
```



- Linked list (nested environments)
- Array (flat environment)



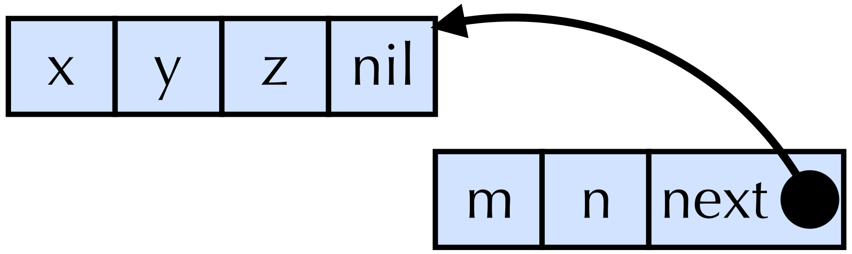
Multiple Arguments

- Can extend DeBruijn indices to allow multiple arguments

```
fun x y z -> fun m n -> x + z + n

fun -> fun-> Var(1,0) + Var(1,2) + Var(0,1)
```

- Nested environments might then be



Tips For Project 5

- You will compile Scheme-like language (i.e., untyped functional language) to Cish
- Break translation down into sequence of smaller simpler passes
  - Don't have to do entire compilation from Scish to Cish in one pass!
  - E.g., do closure conversion as one pass, then lambda lifting, then conversion to Cish, ...
  - You can define additional ASTs for intermediate passes if needed
- Be clear about what each pass is doing
  - Figure out what the invariants of each AST between passes is