

Closures & Environments

CS153: Compilers

Greg Morrisett

"Functional" Languages:

- Lisp, Scheme, Miranda, Hope, ML, OCaml, Haskell, ...
- Functions are first-class
 - not just for calling
 - can pass to other functions (map), return from functions (compose), place in data structures.
- Functions nest
 - A nested function can refer to variables bound in an outer function.

Nesting:

```
val add = fn x => (fn y => y+x)
val inc = add 1 (* = fn y => y + 1 *)
val dec = add ~1 (* = fn y => y + ~1 *)

val compose = fn (f,g) => (fn x => f(g(x)))
val id = compose(inc,dec)
(* = fn x => inc(dec(x)) *)
(* = fn x => (fn y => y+1) ((fn y => y+~1) x) *)
(* = fn x => (fn y => y+1) (x+~1)) *)
(* = fn x => (x+~1)+1 *)
```

After calling `add`, we can't just throw away its arguments (or local variables) because those values are needed in the nested function that it returns.

Substitution-Based Semantics

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

```
fun eval (e:exp) =  
  case e of  
    Int i => Int i  
  | Plus(e1,e2) =>  
    (case (eval e1,eval e2) of  
      (Int i,Int j) => Int(i+j))  
  | Var x => error ("Unbound variable "^x)  
  | Lambda(x,e) => Lambda(x,e)  
  | App(e1,e2) =>  
    (case (eval e1, eval e2) of  
      (Lambda(x,e),v) =>  
        eval (subst(v,x,e)))
```

Substitution-Based Semantics

```
fun subst (v:exp,x:var,e:exp) =
  case e of
    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(x,e) else
      Lambda(x,subst(v,x,e))
  | App(e1,e2) => App(subst(v,x,e1),subst(v,x,e2))
```

Example:

- `App (App (Lambda (x, Lambda (y, Plus (Var x, Var y))), Int 3), Int 4)`
 - `App (Lambda (x, Lambda (y, Plus (Var x, Var y))), Int 3)`
 - `Lambda (x, Lambda (y, Plus (Var x, Var y)))`
 - `Int 3`
 - `eval (subst (Int 3, x, Lambda (y, Plus (Var x, Var y))))`
 - `Lambda (y, Plus (Int 3, Var y))`
 - `Lambda (y, Plus (Int 3, Var y))`
 - `Int 4`
 - `subst (Int 4, y, Plus (Int 3, Var y))`
 - `Plus (Int 3, Int 4)`
 - `Int 3`
 - `Int 4`
 - `Int 7`

Problems:

- Eval crawls over an expression.
- Substitute crawls over an expression.
- So `eval (subst (v, x, e))` is pretty stupid.
Why not evaluate as we substitute?

First Attempt:

```
type env = (exp * value) list
```

```
fun eval (e:exp) (env:env) : value =
```

```
  case e of
```

```
    Int i => Int i
```

```
  | Var x => lookup env x
```

```
  | Lambda(x,e) => Lambda(x,e)
```

```
  | App(e1,e2) =>
```

```
    (case (eval e1 env, eval e2 env) of
```

```
      (Lambda(x,e),v) =>
```

```
        eval e ((x,v)::env))
```

Second Attempt:

```
type env = (exp * value) list
```

```
fun eval (e:exp) (env:env) : value =  
  case e of  
    Int i => Int i  
  | Var x => lookup env x  
  | Lambda(x,e) => Lambda(x,subst(env,e))  
  | App(e1,e2) =>  
    (case (eval e1 env, eval e2 env) of  
      (Lambda(x,e),v) =>  
        eval e ((x,v)::env))
```

Aha!

- Instead of doing the substitution when we reach a lambda, we could instead make a *promise* to finish the substitution if the lambda is ever applied.
- `Lambda (x, subst (env, e)) ==>`
 `Promise (subst, Lambda (x, e))`
- Then we have to modify `App(,)` to take care of the delayed substitution...

Environment-Based

```
datatype value =  
  Int_v of int  
  | Closure_v of {env:env, body:var*exp}  
withtype env = (var * value) list  
  
fun eval (e:exp) (env:env) : value =  
  case e of  
    Int i => Int_v i  
  | Var x => lookup env x  
  | Lambda(x,e) => Closure_v{env=env,body=(x,e)}  
  | App(e1,e2) =>  
    (case (eval e1 env, eval e2 env) of  
      (Closure_v{env=cenv,body=(x,e)},v) =>  
        eval e ((x,v)::cenv))
```

Speeding up the Interpreter

We have to do expensive string comparisons when looking up a variable:

```
Var x => lookup env x
```

where

```
fun lookup ((y,v)::rest) x =  
  if (y = x) then v else lookup rest
```

DeBruijn Indices

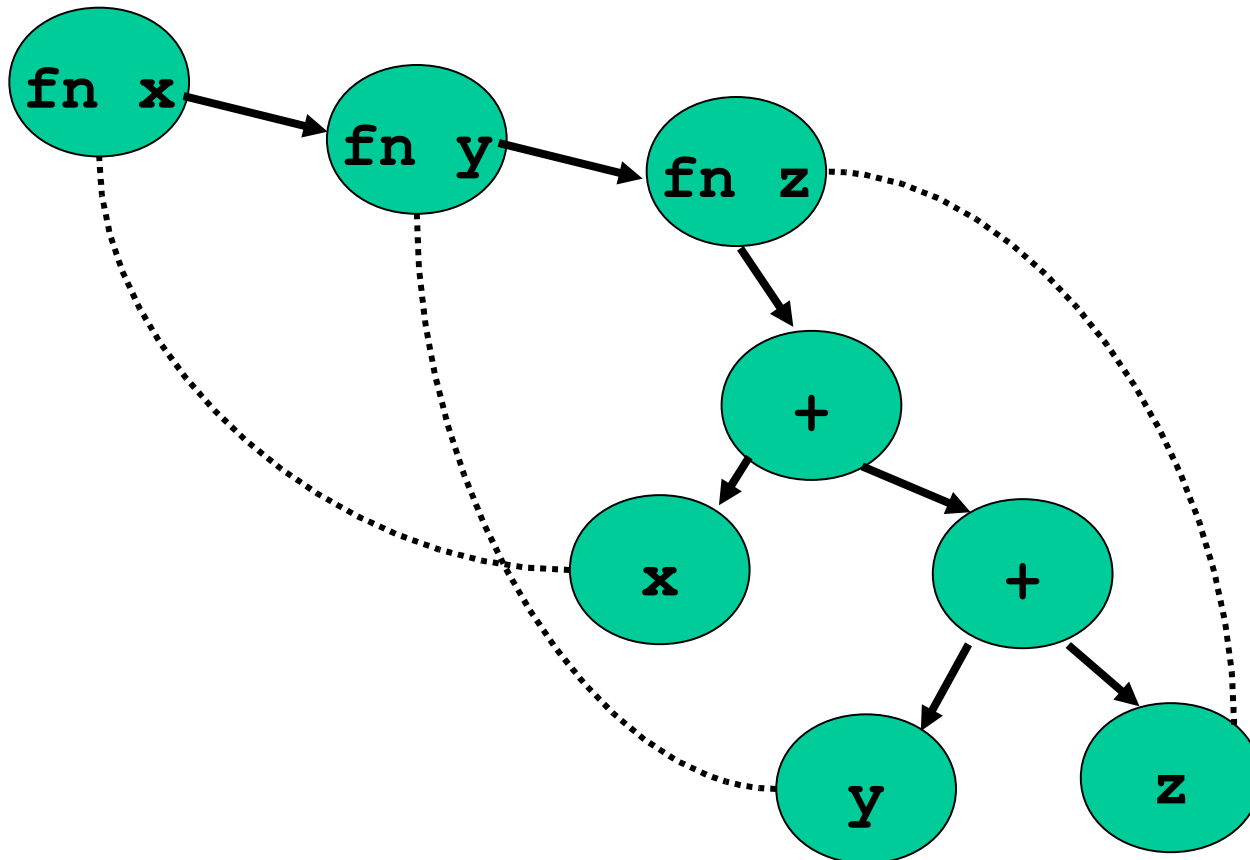
- Instead of using *strings* to represent variables, let's use natural numbers:

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

- The numbers will represent lexical *depth*:
- `fn x => fn y => fn z => x+y+z`
- `fn x2 => fn x1 => fn x0 => x2+x1+x0`
- `fn => fn => fn => 2+1+0`

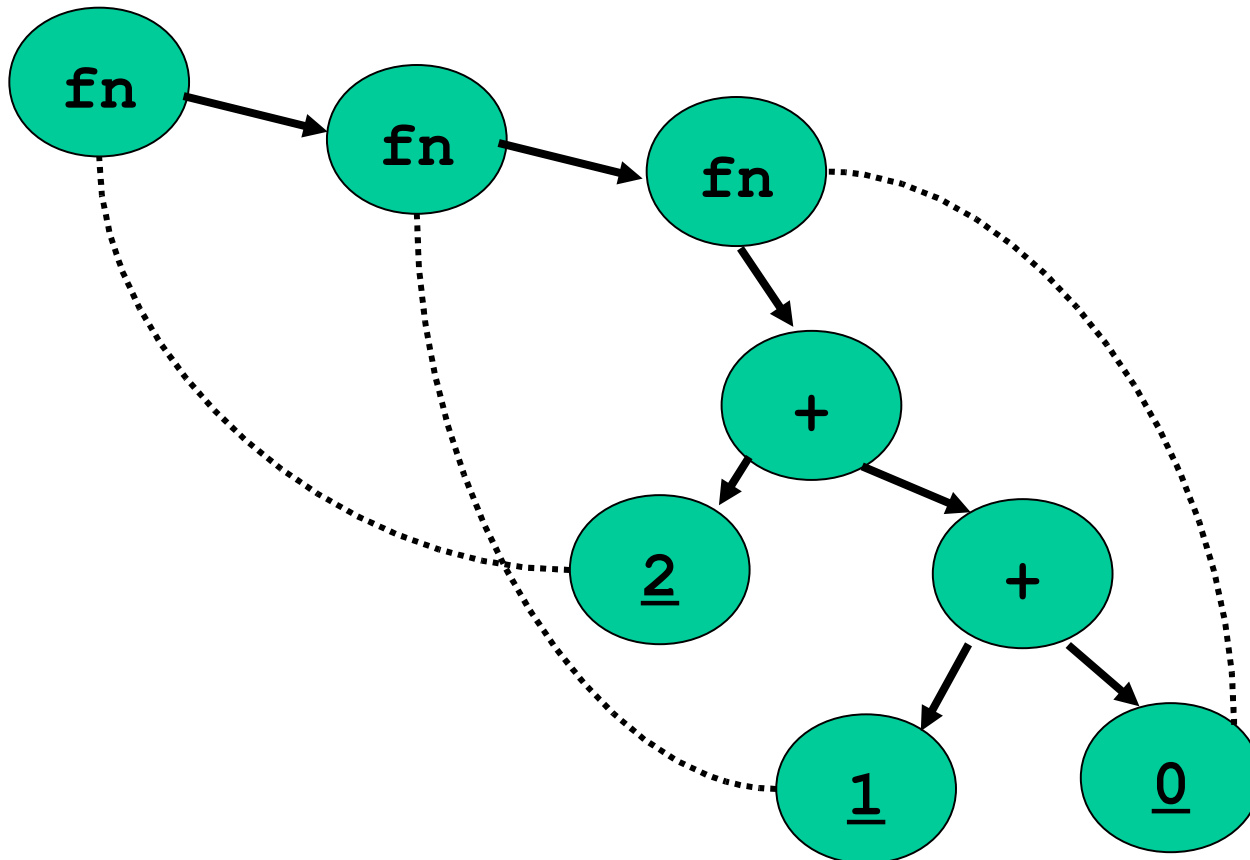
Graphs as Trees

- `fn x => fn y => fn z => x+(y+z)`



Graphs as Trees

- $fn \Rightarrow fn \Rightarrow fn \Rightarrow \underline{2} + (\underline{1} + \underline{0})$



Converting:

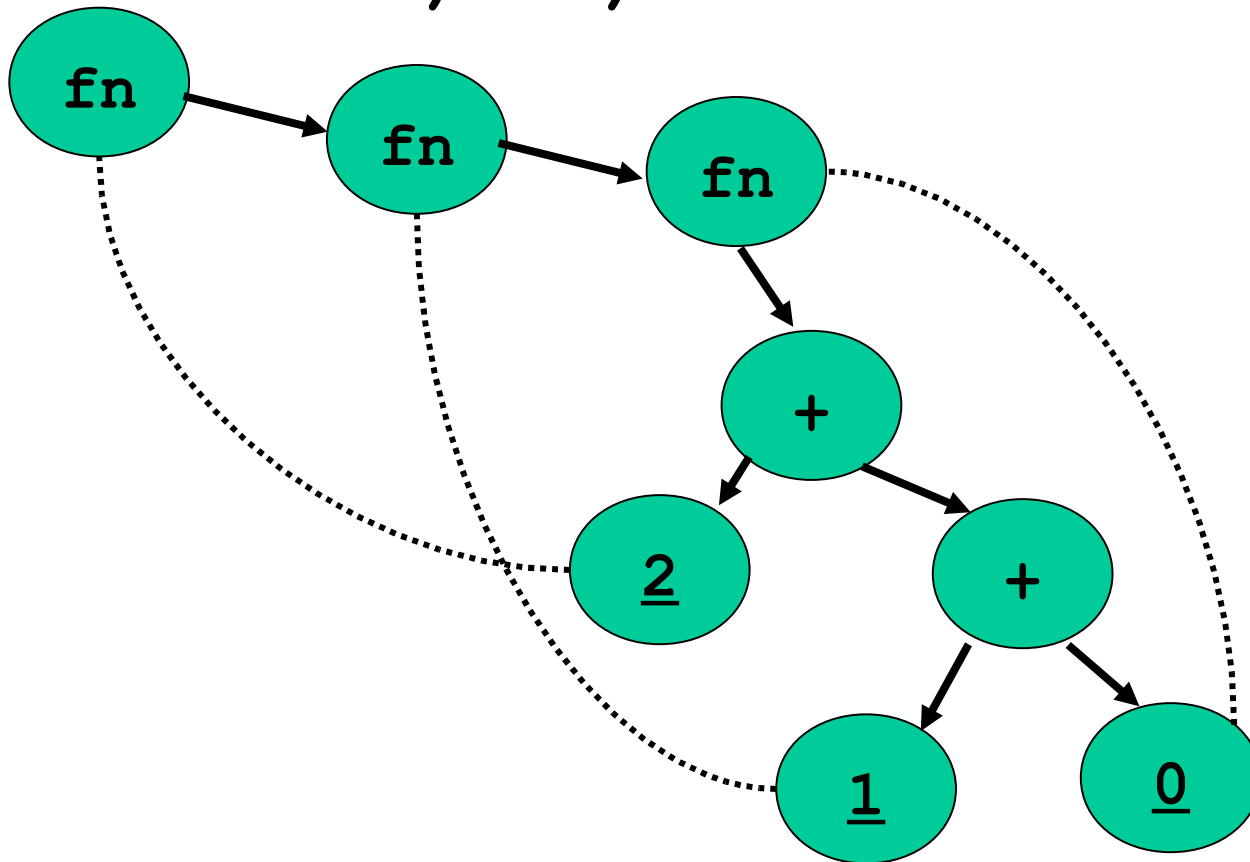
```
fun cvt (e:exp) (env:var->int) : D.exp =
  case e of
    Int i => D.Int i
  | Var x => D.Var (env x)
  | App(e1,e2) =>
    D.App(cvt e1 env,cvt e2 env)
  | Lambda(x,e) =>
    let fun new_env(y) =
      if y=x then 0 else (env y)+1
    in
      Lambda(cvt e new_env)
    end
end
```

New Interpreter:

```
datatype value =  
  Int_v of int  
  | Closure_v of {env:env, body:exp}  
withtype env = value list  
  
fun eval (e:exp) (env:env) : value =  
  case e of  
    Int i => Int_v i  
  | Var n => List.nth(env,n)  
  | Lambda(e) => Closure_v{env=env,body=e}  
  | App(e1,e2) =>  
    (case (eval e1 env, eval e2 env) of  
      (Closure_v{env=cenv,body=e},v) =>  
        eval e (v::cenv))
```

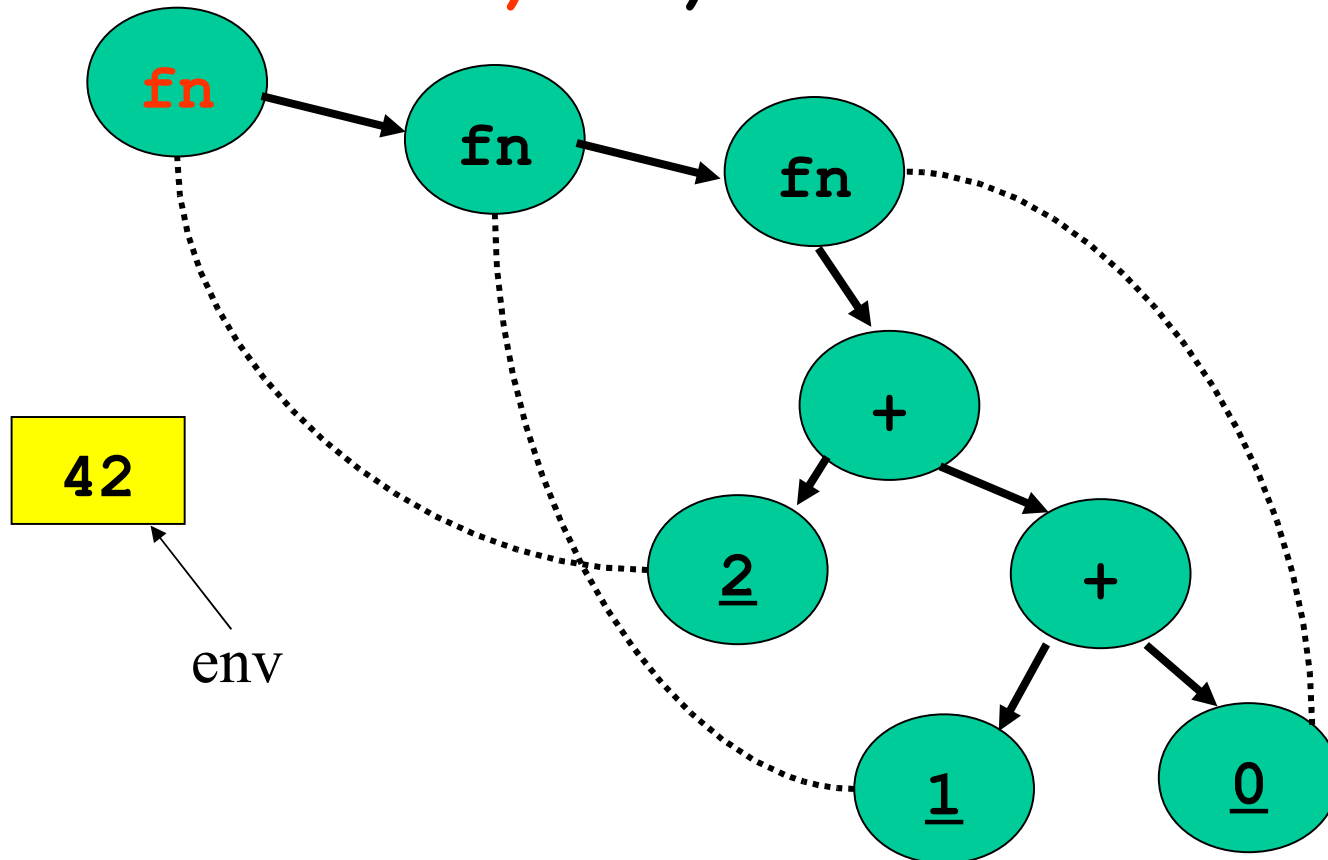
Environments

- $(((fn \Rightarrow fn \Rightarrow fn \Rightarrow \underline{2} + (\underline{1} + \underline{0}))$
42) 37) 21



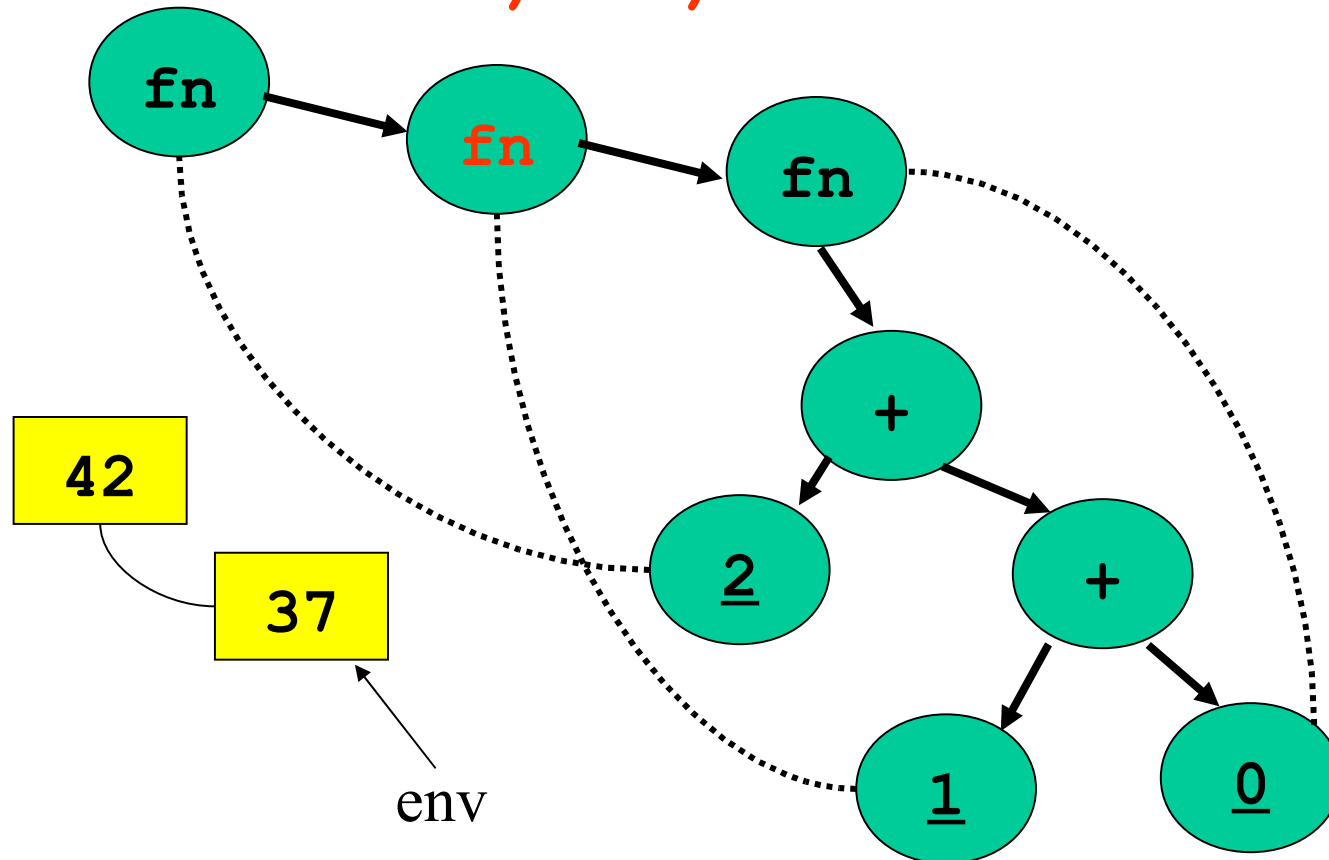
Environments

- $(((fn \Rightarrow fn \Rightarrow fn \Rightarrow \underline{2} + (\underline{1} + \underline{0}))$
 $42) 37) 21$



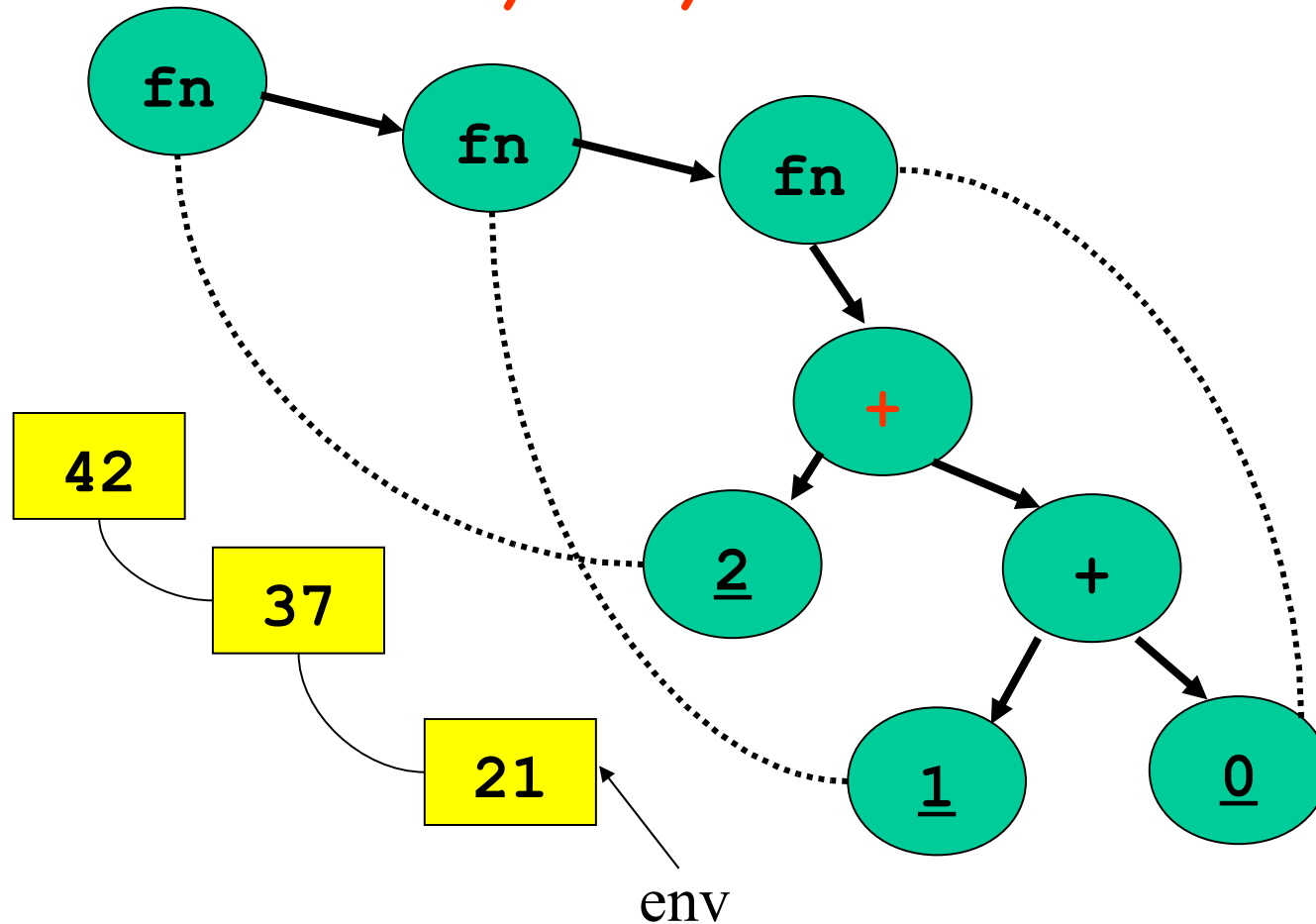
Environments

- $(((fn \Rightarrow fn \Rightarrow fn \Rightarrow \underline{2} + (\underline{1} + \underline{0}))$
 $42) 37) 21$



Environments

- $(((fn \Rightarrow fn \Rightarrow fn \Rightarrow \underline{2} + (\underline{1} + \underline{0}))$
42) 37) 21



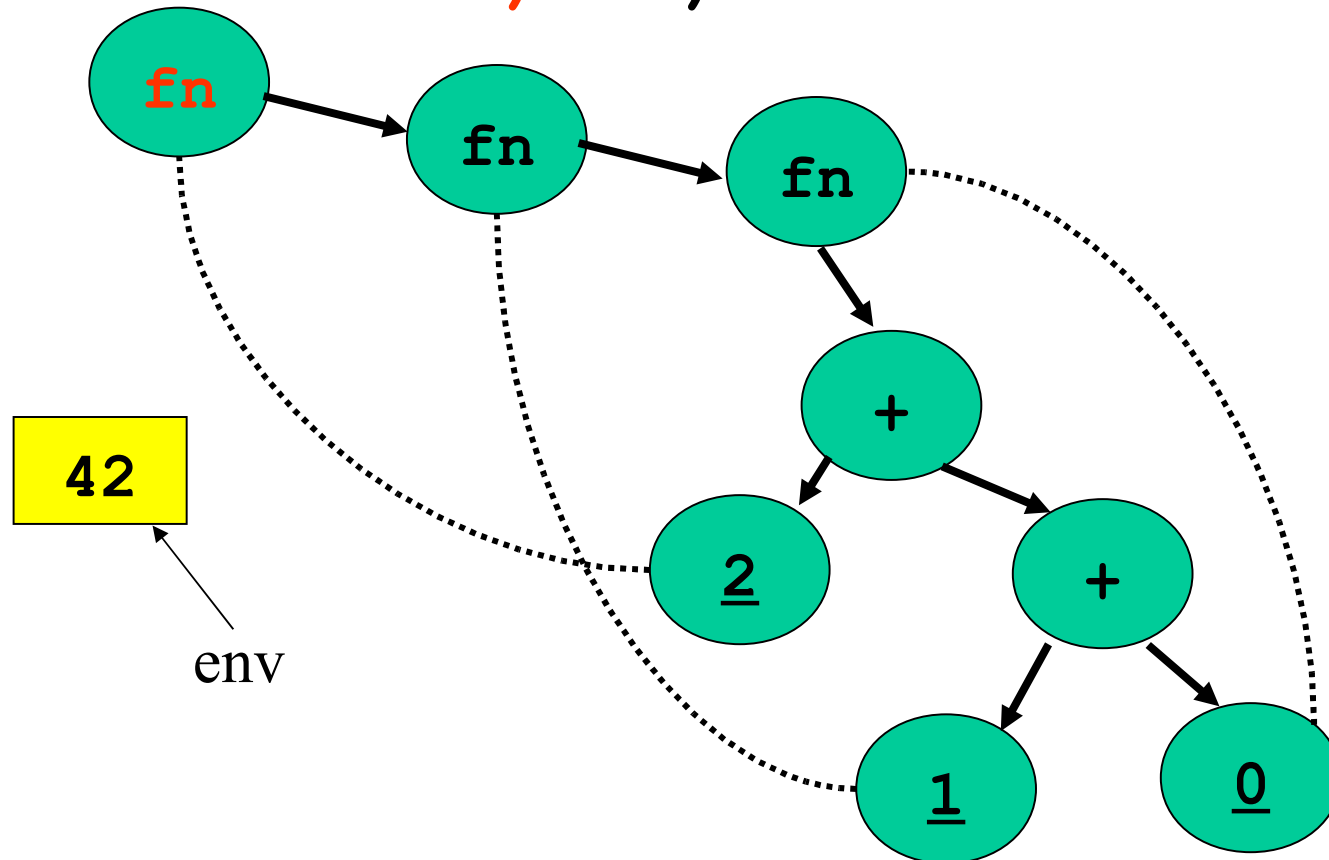
Alternative:

```
datatype value =  
  Int_v of int  
  | Closure_v of {env:env, body:exp}  
withtype env = value vector
```

```
fun eval (e:exp) (env:env) : value =  
  case e of  
    Int i => Int_v i  
  | Var n => Vector.sub(env,n)  
  | Lambda(e) => Closure_v{env=env,body=e}  
  | App(e1,e2) =>  
    (case (eval e1 env, eval e2 env) of  
      (Closure_v{env=cenv,body=e},v) =>  
        eval e (vector_cons(v,cenv)))
```

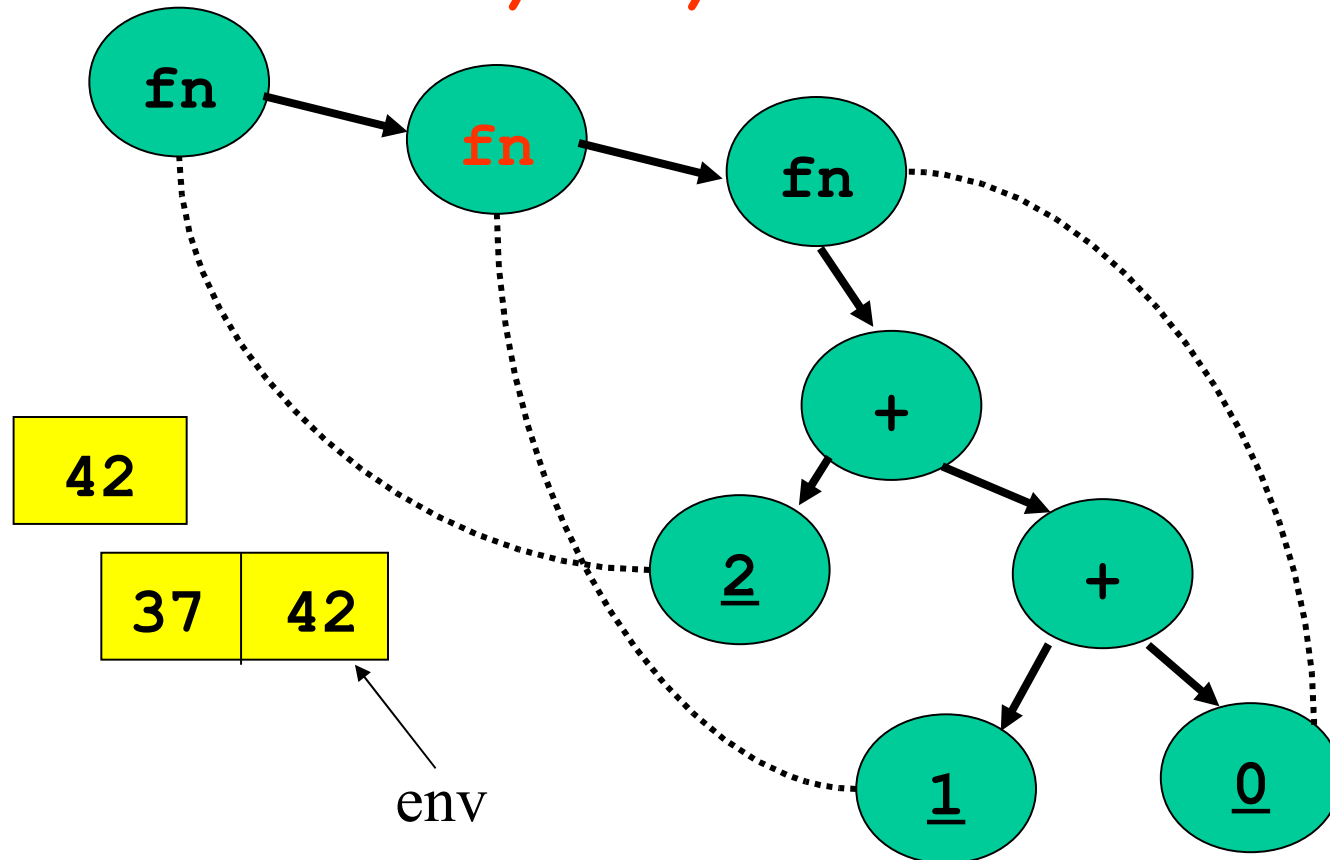
Flat Environments

- $(((fn \Rightarrow fn \Rightarrow fn \Rightarrow \underline{2} + (\underline{1} + \underline{0}))$
 $42) \ 37) \ 21$



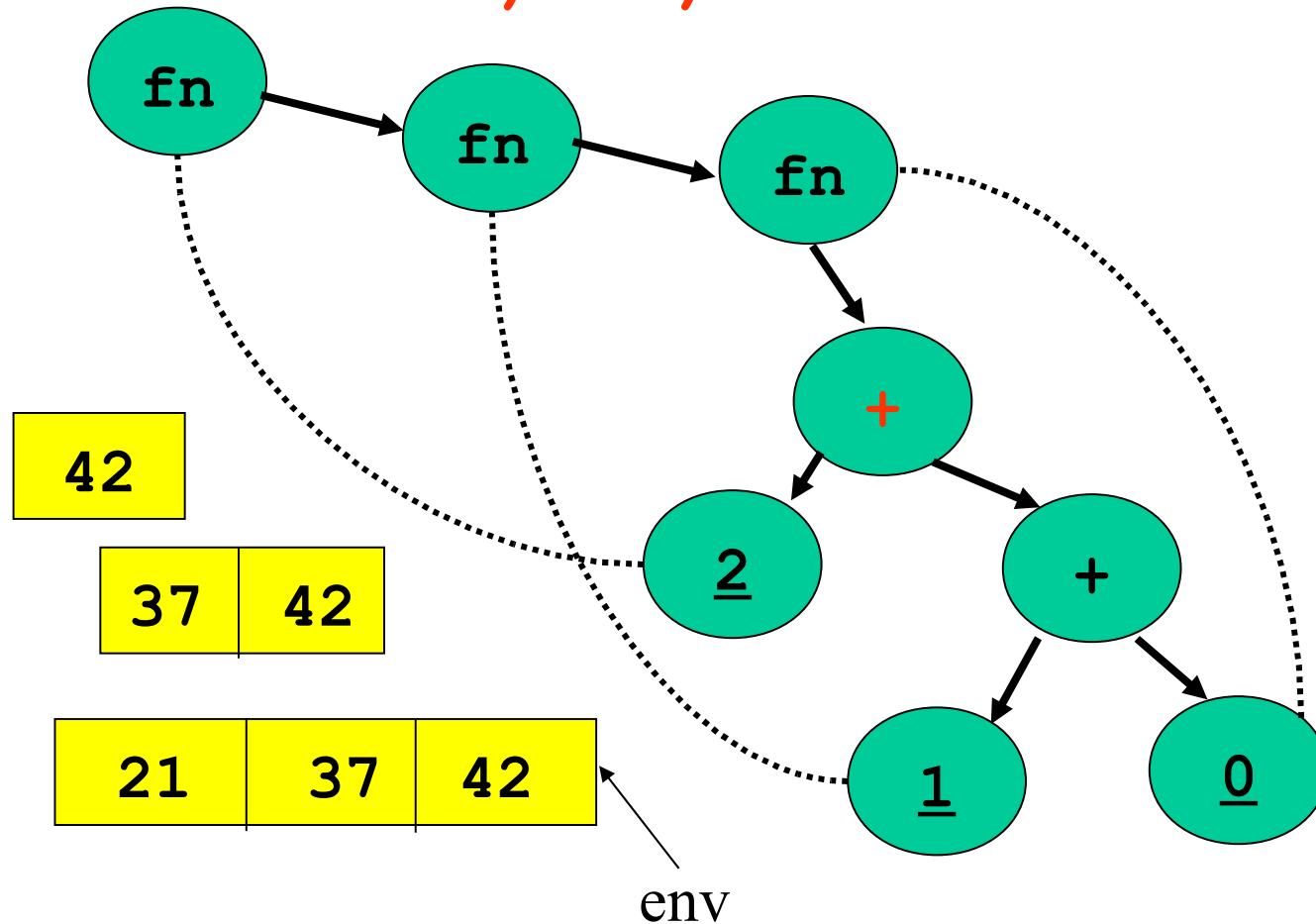
Flat Environments

- $(((fn \Rightarrow fn \Rightarrow fn \Rightarrow \underline{2} + (\underline{1} + \underline{0}))$
 $42) 37) 21$



Flat Environments

- $(((fn \Rightarrow fn \Rightarrow fn \Rightarrow \underline{2} + (\underline{1} + \underline{0}))$
 $42) 37) 21$



Schemeish Environments

```
(lambda (x y z)
  (lambda (n m) (lambda (p) (+ n z))))
```

