Closures & Environments

CS4410: Spring 2013

"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:

```
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 *)
```

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

```
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 (substitute v x e)))
```

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)
```

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 (substitute v x e))` is pretty stupid. Why not evaluate as we substitute?

First Attempt:

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

let rec eval (e:exp) (env:env) : value =
  match e with
   | Int i -> Int 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))
```

Second Attempt:

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

let rec eval (e:exp) (env:env) : value =
  match e with
   | Int i -> Int 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))
```

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) as code  ==>`**
  `Promise(subst,code)`
- Then we have to modify App(_,_) to take care of the delayed substitution…

Closure-Based Semantics

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

Speeding up the Interpreter

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

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

where

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

DeBruijn Indices

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

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

- The numbers will represent lexical *depth*:

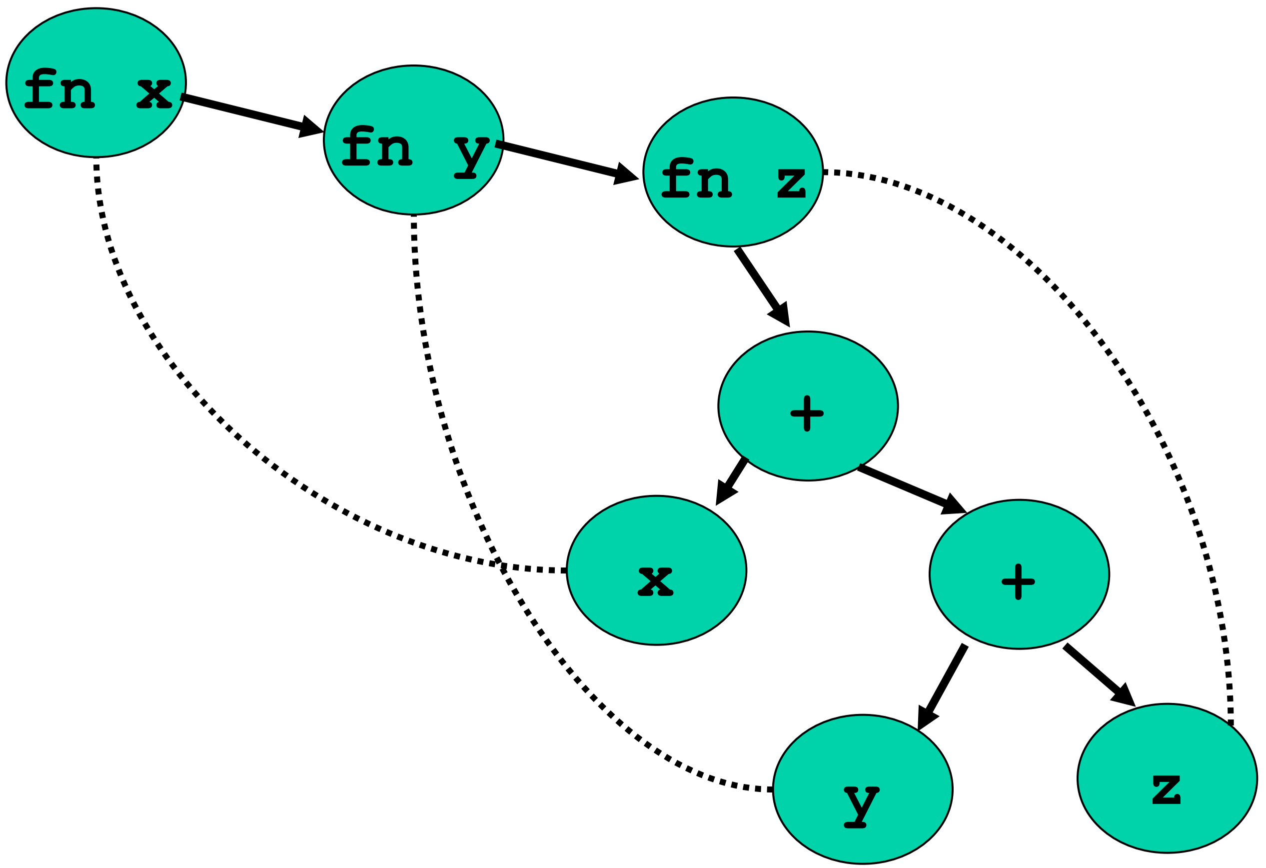
```
fun x -> fun y -> fun z -> x+(y+z)
```

fun x_2 **->** **fun** x_1 **->** **fun** x_0 **->** $x_2+(x_1+x_0)$

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

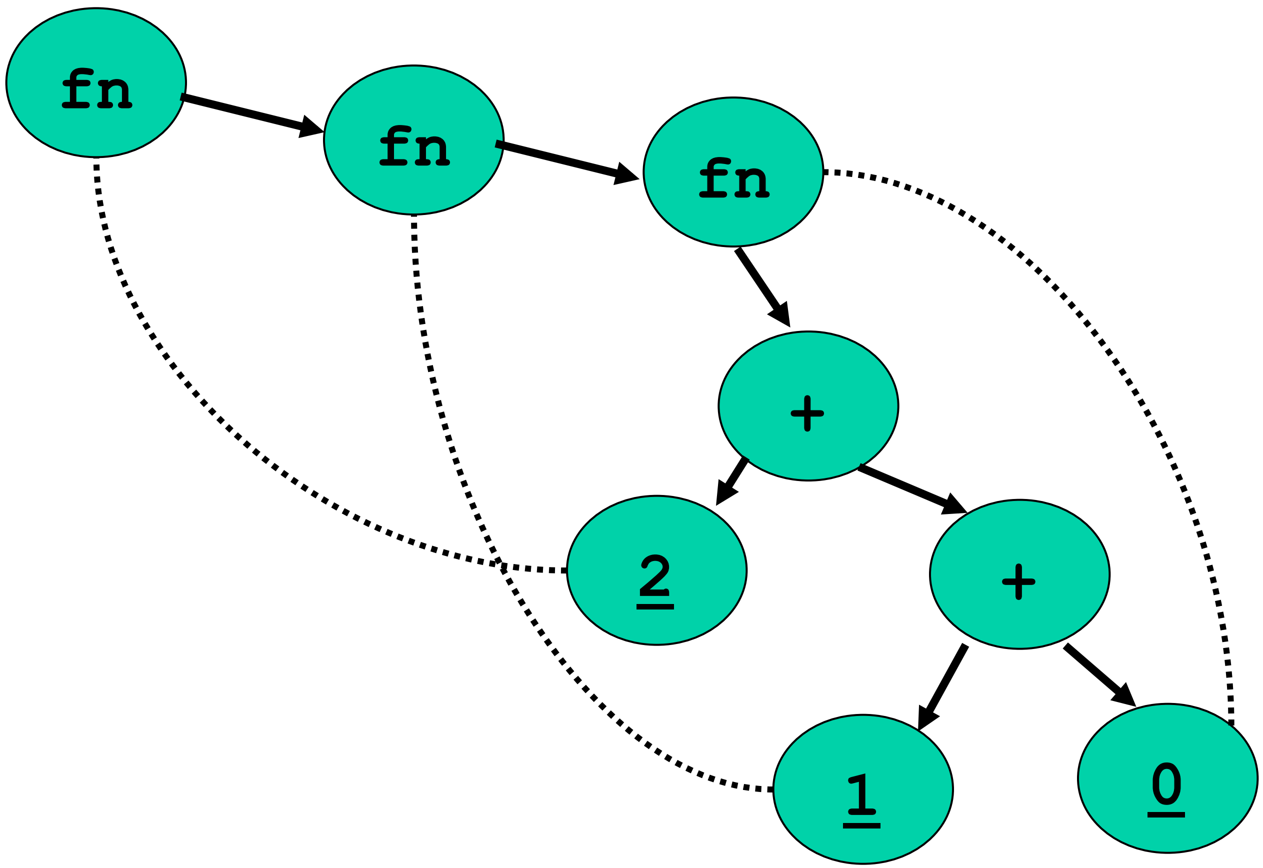
Graphs as Trees

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



Graphs as Trees

fun -> fun -> fun -> 2 + (1 + 0)



Converting:

```
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)
```

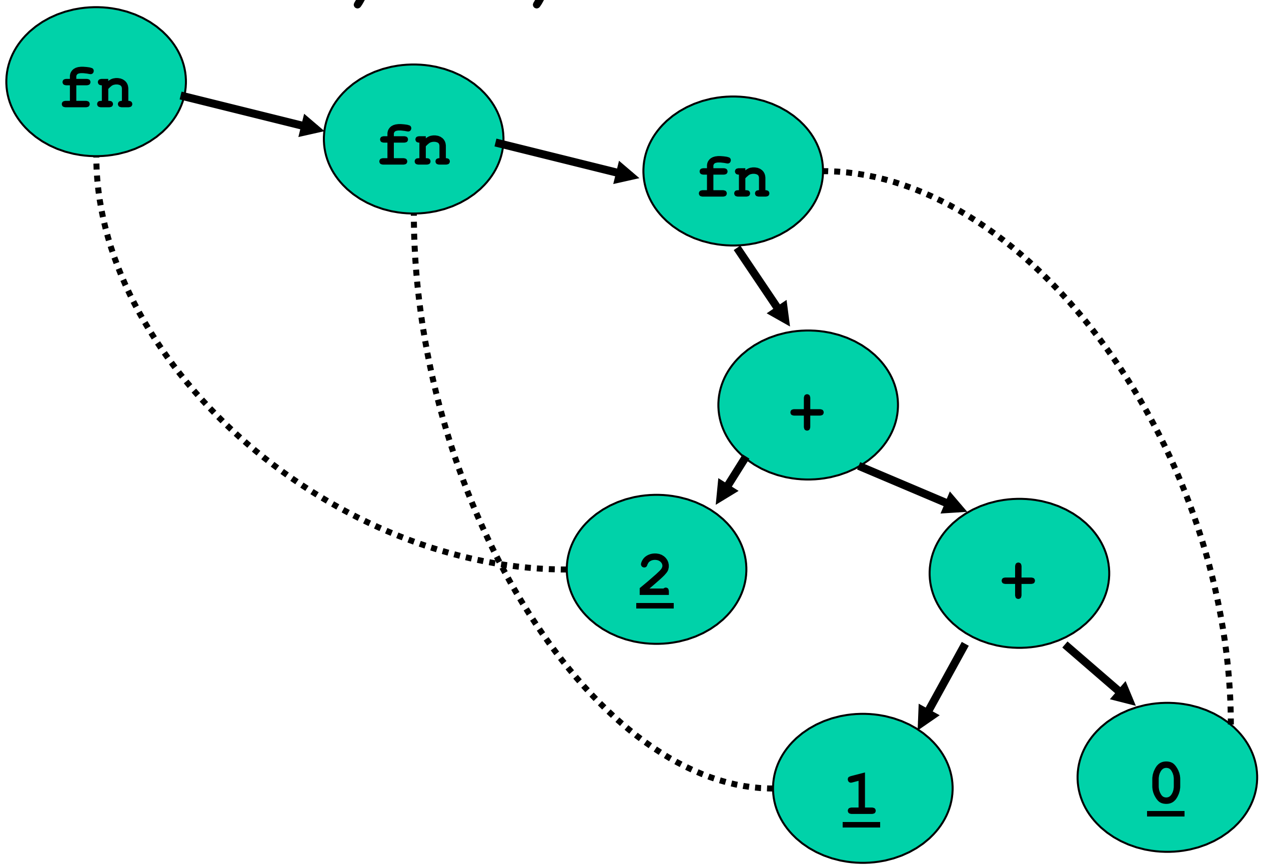
New Interpreter:

```
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 n -> List.nth(env,n)
    | Lambda e -> Closure_v{env=env,body=e}
    | App(e1,e2) ->
        (match eval e1 env, eval e2 env with
         | Closure_v{env=cenv,body=e}, v ->
               eval e (v::cenv))
```

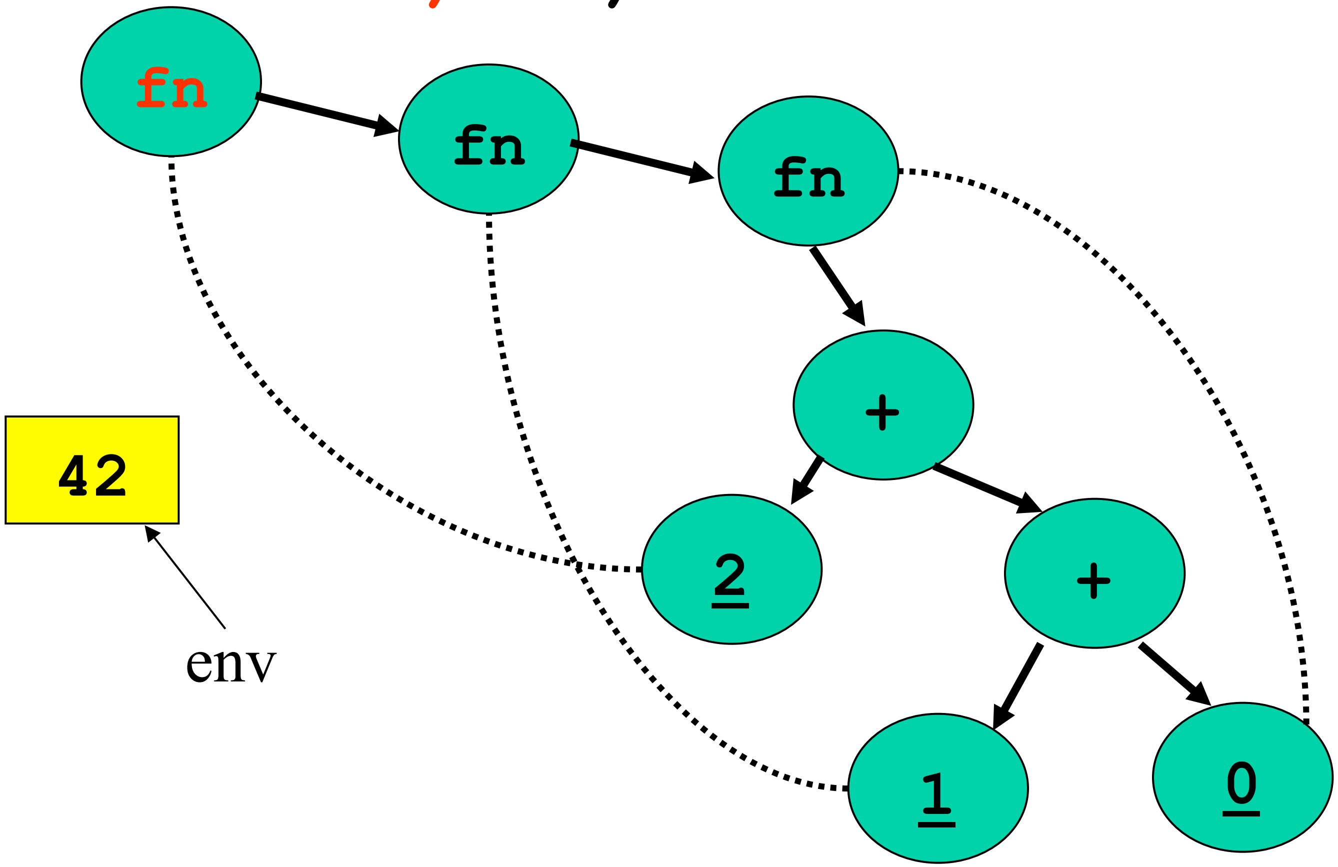
Environments

```
(((fun -> fun -> fun -> 2 + (1 + 0))
              42) 37)   21
```



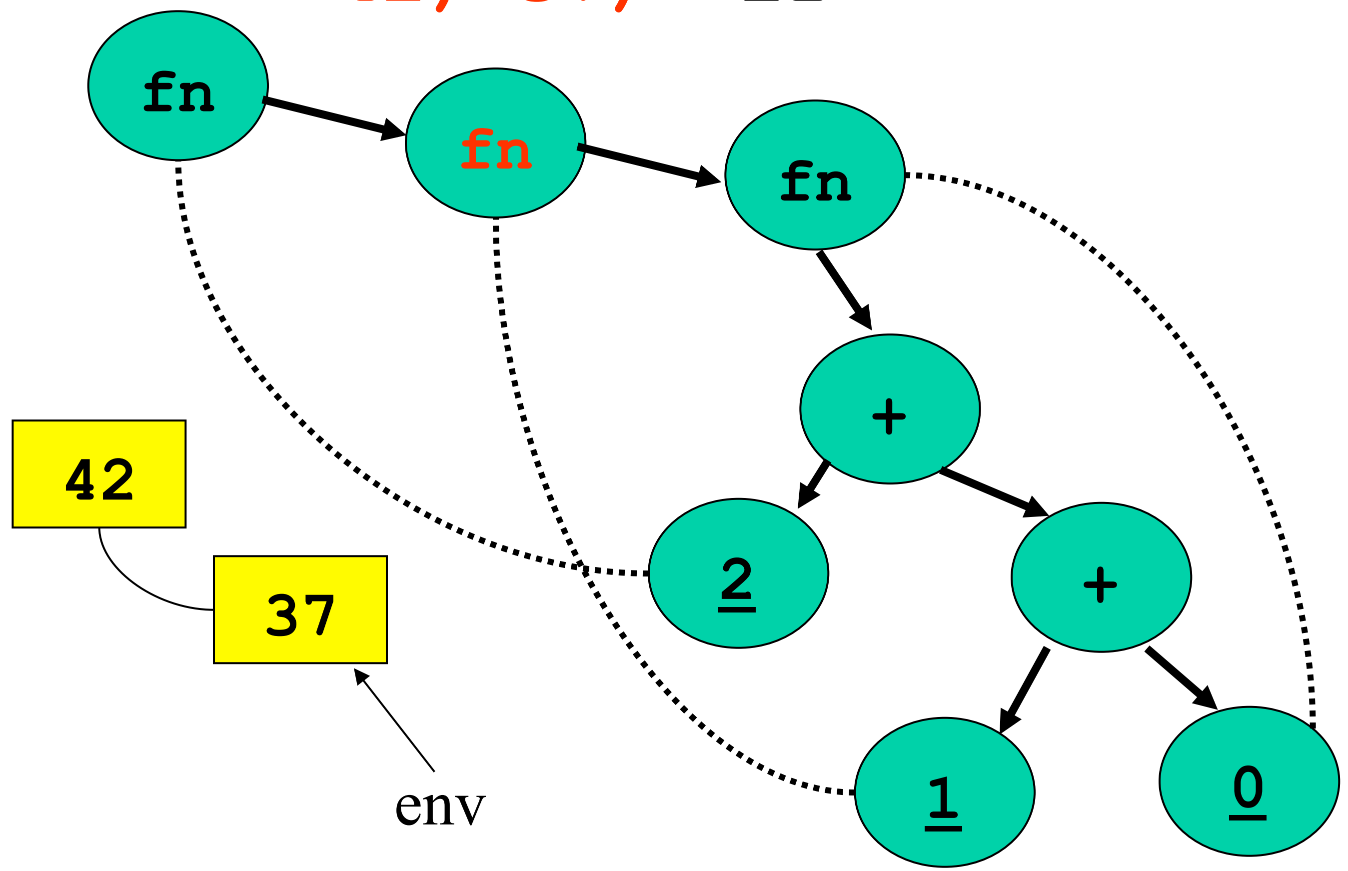
Environments

```
(((fun -> fun -> fun -> 2 + (1 + 0))
              42) 37)  21
```



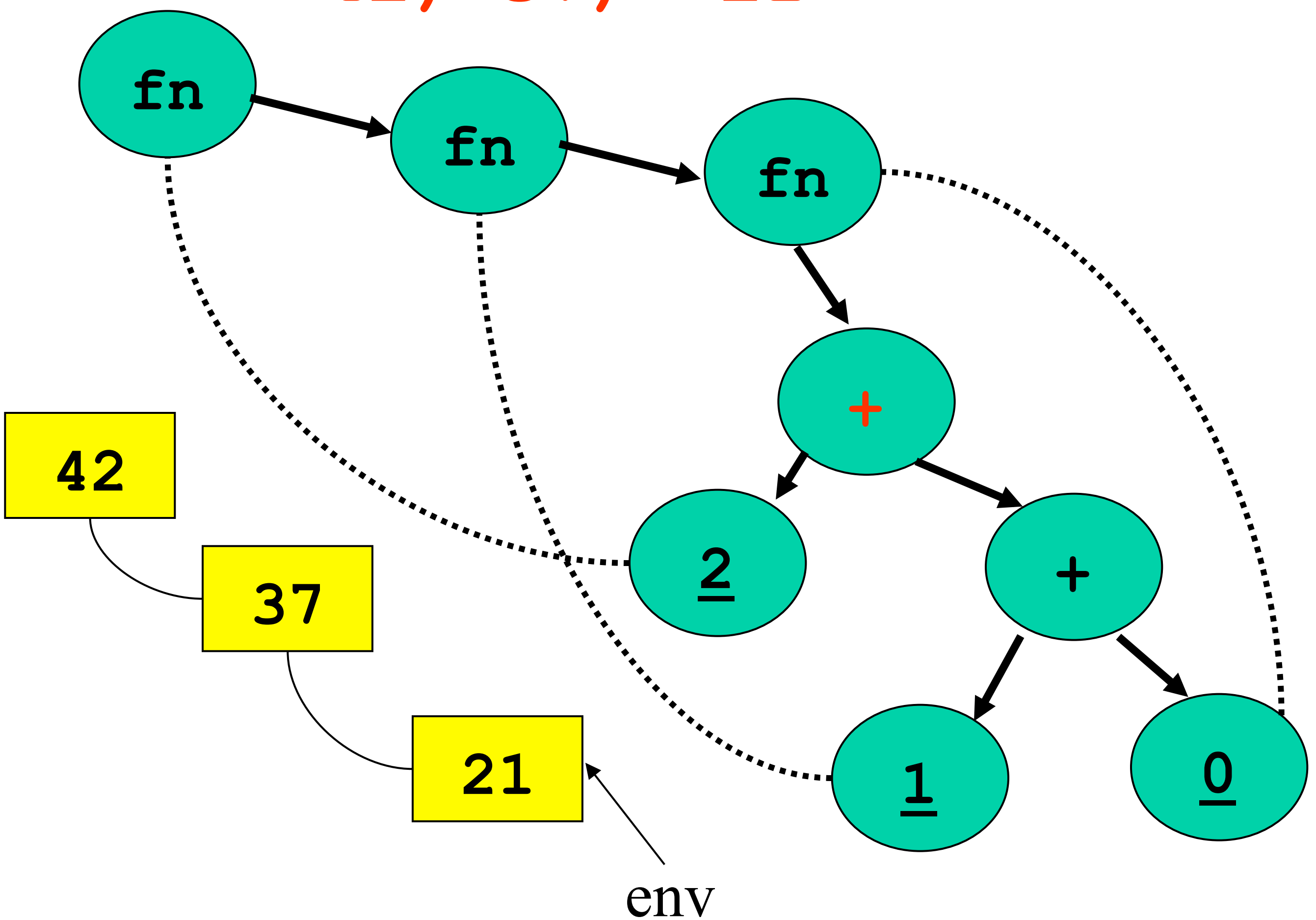
Environments

(((fun -> fun -> fun -> 2 + (1 + 0))
42) 37) 21



Environments

```
(((fun -> fun -> fun -> 2 + (1 + 0))
        42) 37)  21
```



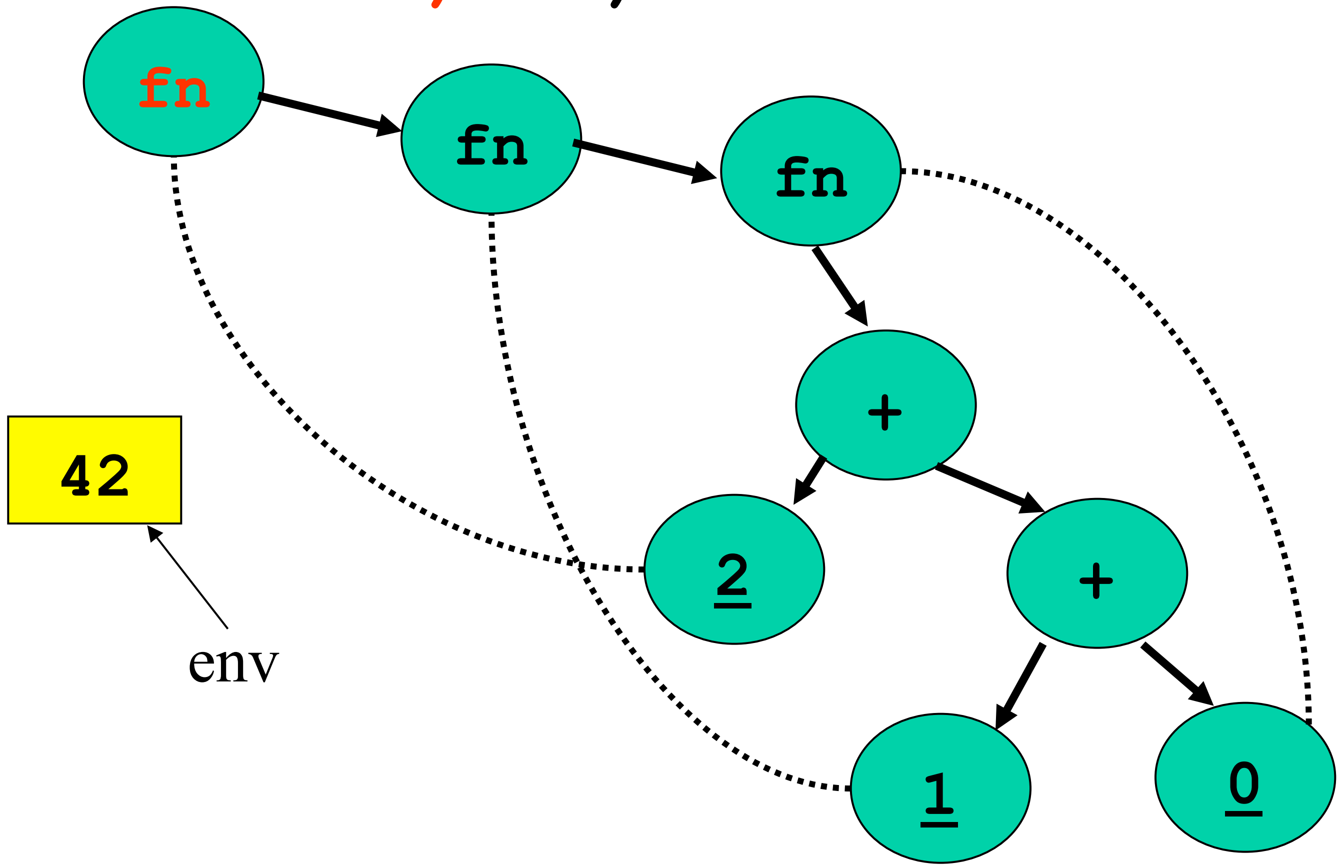
Alternative:

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

let rec eval (e:exp) (env:env) : value =
  match e with
   | Int i -> Int_v i
   | Var n -> Array.sub(env,n)
   | Lambda(e) -> Closure_v{env=env,body=e}
   | App(e1,e2) ->
       (match eval e1 env, eval e2 env with
        | Closure_v{env=cenv,body=e}, v ->
              eval e (Array.append([|v|],cenv)))
```

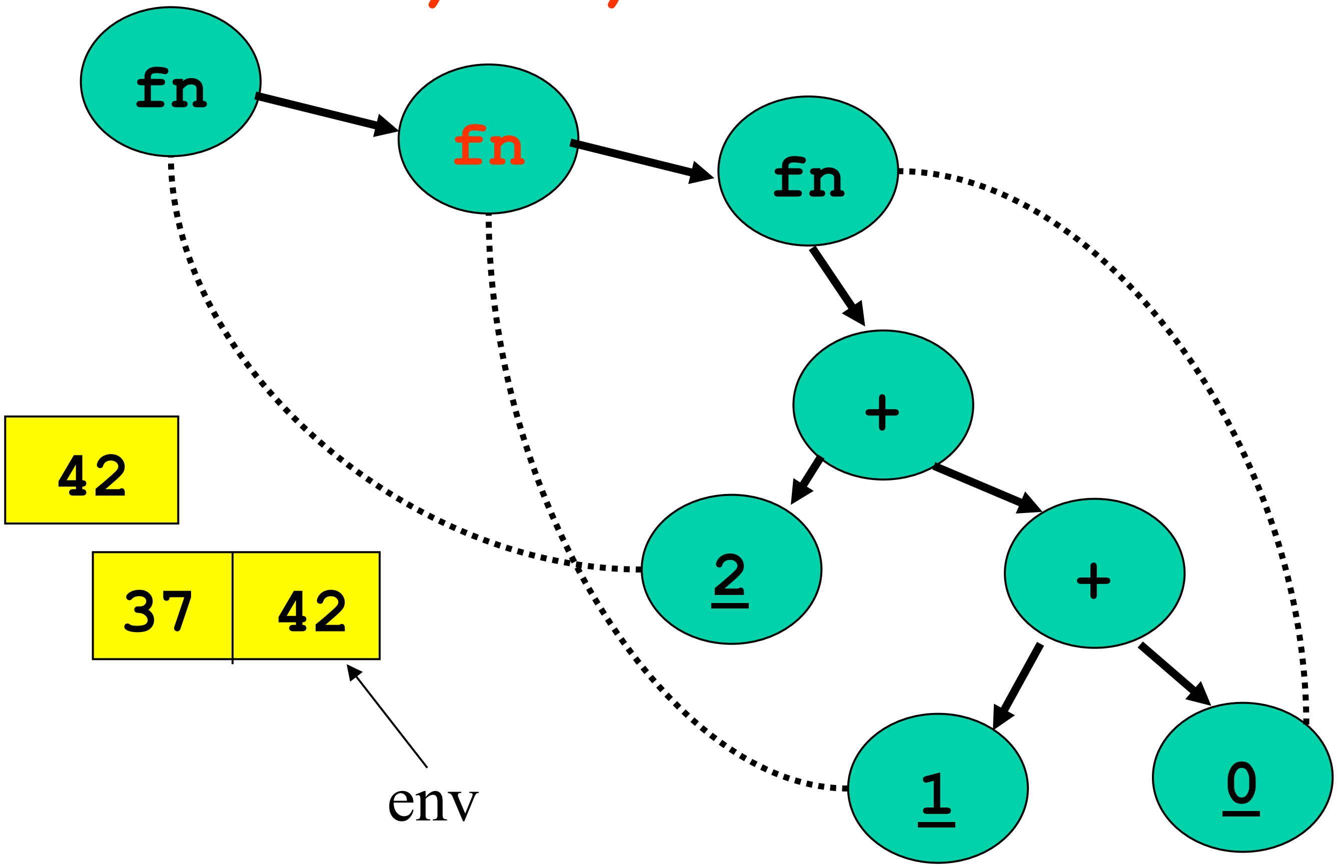
Flat Environments

```
(((fun -> fun -> fun -> 2 + (1 + 0))
              42) 37)  21
```



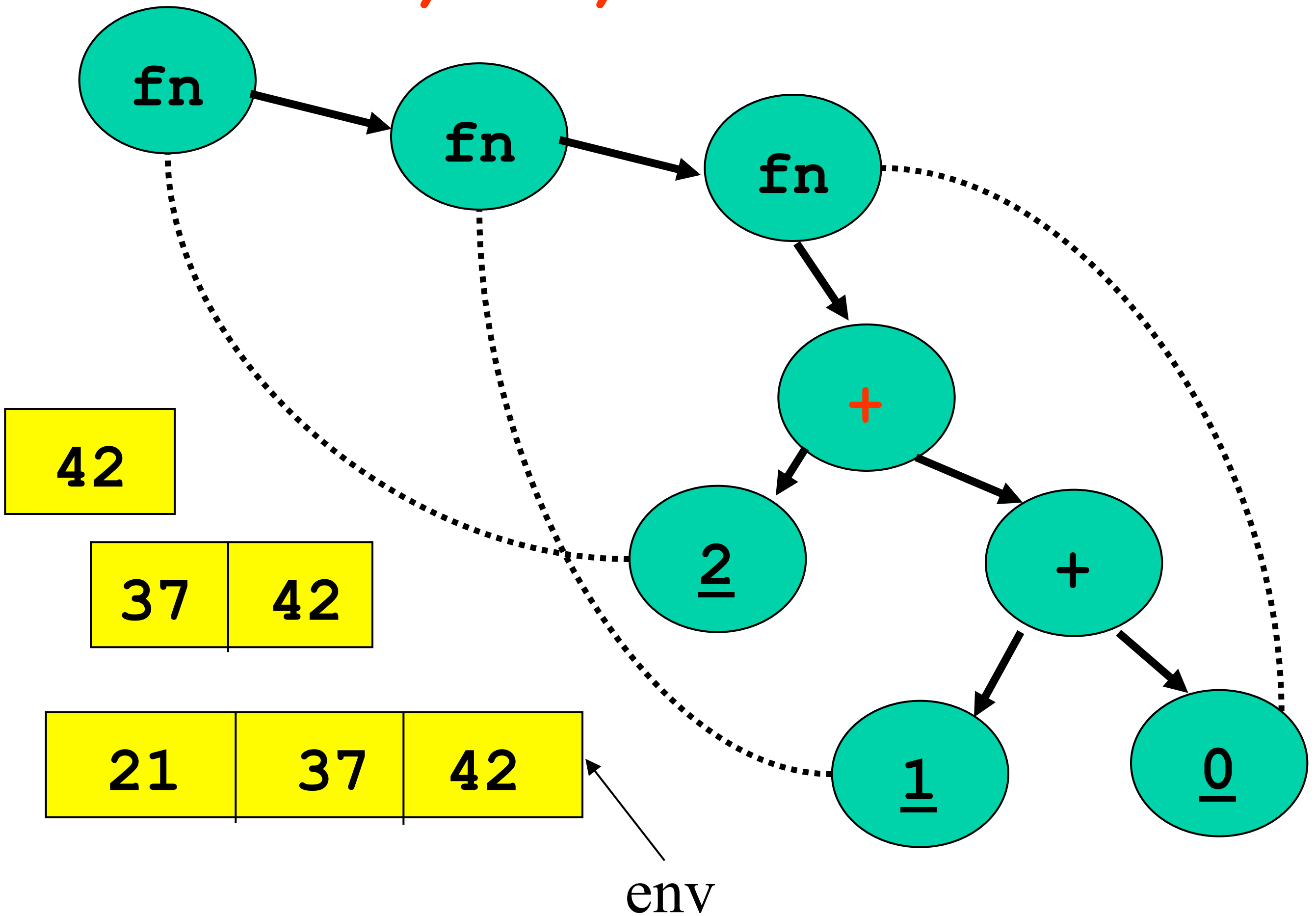
Flat Environments

```
(((fun -> fun -> fun -> 2 + (1 + 0))
            42) 37)  21
```



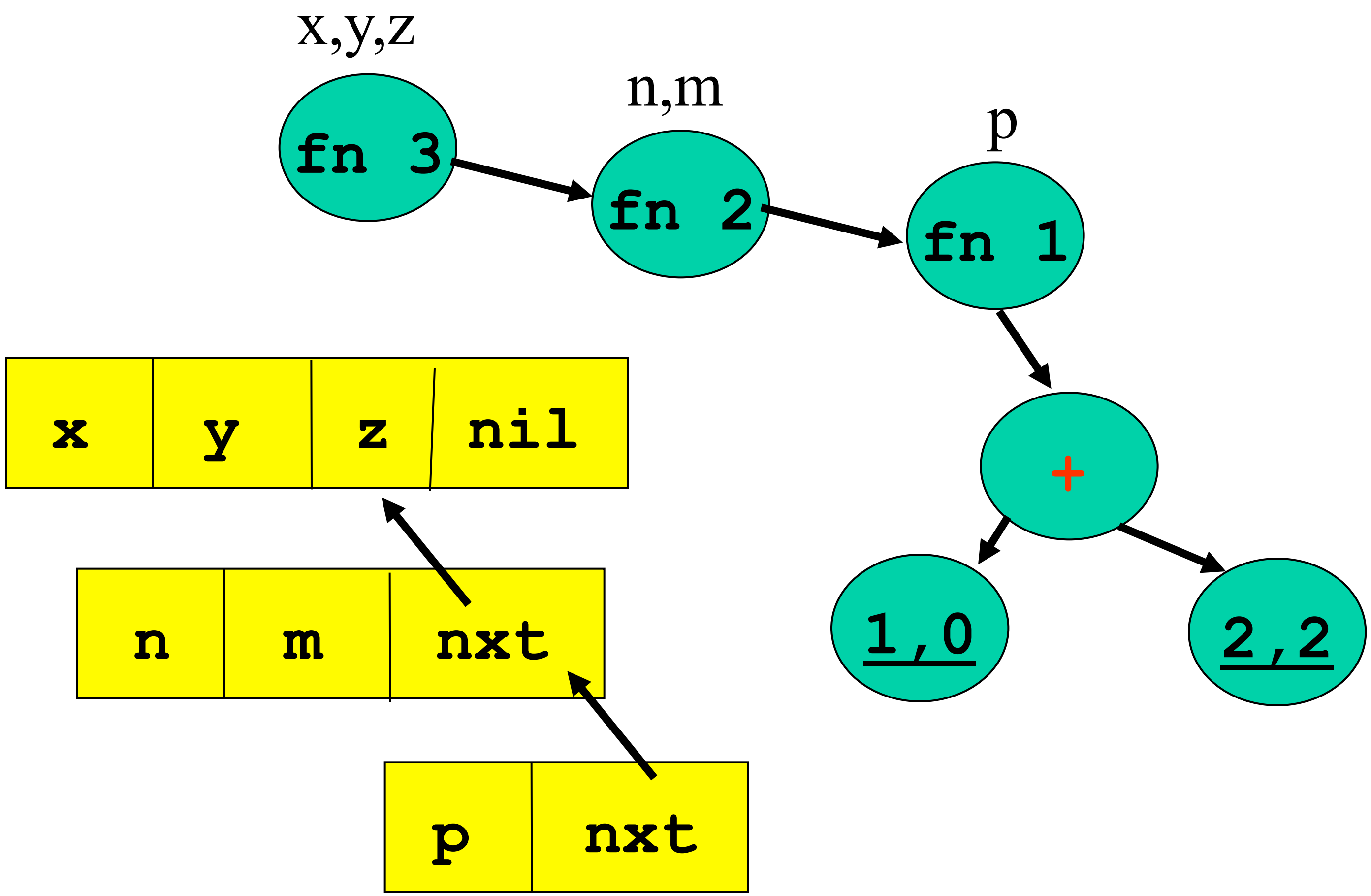
Flat Environments

```
(((fun -> fun -> fun -> 2 + (1 + 0))
              42) 37)   21
```



Schemeish Environments

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



Scish

```
type exp = Int of int | Var of var |
    PrimApp of primop * exp list |
    Lambda of var * exp |
    App of exp * exp |
    If of exp * exp * exp
```

Cish Extended

*exp : reads word at address denoted by exp.
The address must be word-aligned.

*exp1 = exp2 : places value of exp2 in
memory at address denoted by exp1.
Returns value of exp2 as result.
The address must be word aligned.

Cish Extended

exp(exp1,...,expn) : function names are now first-class values

malloc(n) : allocate n bytes of storage, and return (word-aligned) pointer to the storage.

Compiling Scish to Cish

Lambda expression:

- Generate a new Cish function f that takes one parameter -- the environment
- Produce as a value a closure (pair of the function name f and the current env).

Compiling Scish to Cish

Application (e1 e2):

- Evaluate e1 and e2 to values v1 and v2 (where v1 is a closure)
- Extract the function pointer and env of the closure, extend the env by adding in v2, then invoke the function with extended env.