Lecture 6

Thursday, September 9, 2021 2:03 PM

```
let even4 n =
    let rec unwind stack acc =
      match stack with
      | [] -> acc
      | hd :: tl -> unwind tl (hd acc) in
    let rec _even stack n =
      if n = 0 then (unwind stack true) else _even (not :: stack) (n - 1) in
    _even [] n
```

---

③ So n must be an integer

```
let even4 n =
    let rec unwind stack acc =
      match stack with
      | [] -> acc
      | hd :: tl -> unwind tl (hd acc) in
    let rec _even stack n =
      if n = 0 then (unwind stack true) else _even (not :: stack) (n - 1) in
    _even [] n
```

② So this must be an integer

① This n is an integer

```
let even4 n =
    let rec unwind stack acc =
      match stack with
      | [] -> acc
      | hd :: tl -> unwind tl (hd acc) in
    let rec _even stack n =
      if n = 0 then (unwind stack true) else _even (not :: stack) (n - 1) in
    _even [] n
```

① So acc must have type bool

① stack must be a list. But of what?

② The head of the stack is applied to something

So stack : (bool → bool) list

③ We pass (hd acc) back to unwind

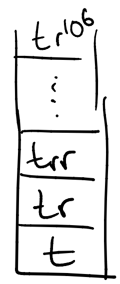
hd : bool → bool

③ unwind : (bool -> bool) list -> bool -> bool

④ _even : (bool -> bool) list -> int -> bool

⑤ even4 : int -> bool

Trees




```
type tree = Leaf | Node of int * tree * tree
```

```
let rec inorder1 t =
match t with
| Leaf -> []
| Node(c, tl, tr) -> (inorder1 tl) @ [c] @ (inorder1 tr)
```

Question: Is inorder1 tail recursive?

No.

Question (For extra credit / HW1)

Can you provide a tail recursive variant of inorder1?

Mutual recursion

```
let rec even x = if x = 0 then true else odd (x - 1)

let rec odd x = if x = 0 then false else even (x - 1)
```

X How to define even without having previously defined previously defined

odd?

```
let rec even x = if x = 0 then true else odd (x - 1) and
odd x = if x = 0 then false else even (x - 1)
```

Define them together!

```
type tree = Leaf | Node of node_rec and
node_rec = { value : int; left : tree; right : tree; color : bool }
```

Mutual recursion in ADT definitions

Scoping Rules

"What does a variable refer to?"

```
let f = fun x -> x in f 0
```

```
let x = 1 in
let f = fun y -> x in
let x = 2 in
f 0
```

Unused

Our expectation of result = 1.

Emacs Lisp

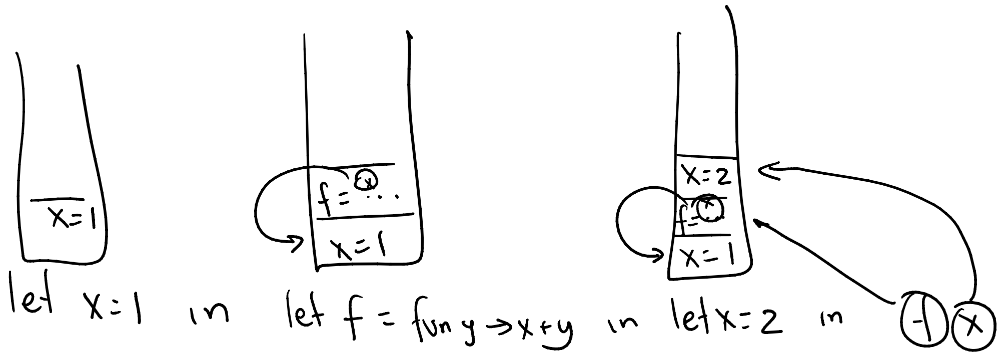
(There are completely reasonable languages where this would evaluate to 2.)

```
let x = 1 in
let f = fun y -> x + y in
let x = 2 in
f x
```

Expected result: 3

How to implement this behavior?

1. Maintain a stack



let x=1 in let f = fun y => x+y in let x=2 in f(x)

2. Maintain an "environment"

env : (var name) $\longrightarrow$ value

Maybe maintain as a stack.

eval env exp $\longrightarrow$ value

At the top, eval [] exp.

---

eval env (let x = e_1 in e_2)

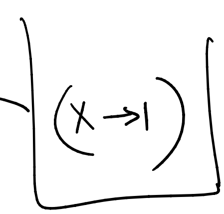
= let v_x = eval env e_1 in
  let env' = (x, v_x) :: env in
     eval env' e_2

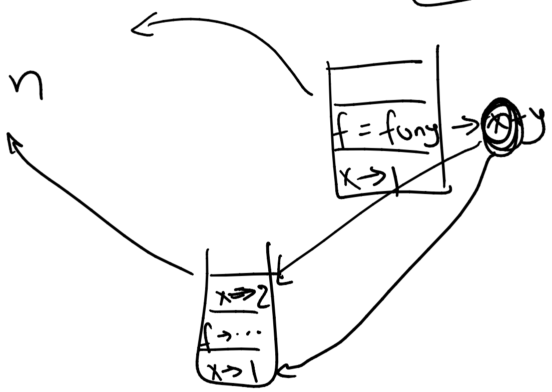
```
eval env (f1 f2) =
  let v1 = eval env f1 in
  let v2 = eval env f2 in

  (* v1 must be a fn-like thing *)
  (* Replace all occurrences of its input
     argument with v2 *)

  (* eval env resulting expression in
     same environment *)
```

```
let x = 1 in
let f = fun y -> x+y in
let x = 2 in
f x
```





f preserves a "photograph" of the environment from when it was defined.

from when it was defined.

"Closure" ← Static scoping
Lexical scoping

Emacs LISP ← Dynamic scoping

```
foo() {
  ⋮
  raise ——
  ⋮
}

bar () {
  ⋮
  catch ——
  ⋮
}
```

Who catches this exception?

```
—— baz () {
  ⋮
  int foo () {
    ⋮
    raise exception
    ⋮
  }
}

—— bar () {
  f = baz()
  f()
  catch exception
}
```

}

}

}

}

catch exception

return foo

}

Lexical scope vs. Dynamic scope

```
① let e1 = let x=1 in
            let f y = let x = y+1 in
                      fun z -> x+y+z in
            let x=3 in
            let w = (f 4) 6 in
            w
```

- Lexical scoping makes variable renaming/refactoring easy.
- Makes type inference easy.