Announcements

- HW5 due today

- HW6 will be out tonight
  - Will be available for submission in HW Server
  - Functional correctness: 75%,
  - "Comments": 25%

---

Last class

- Scheme
  - Equality testing
    - eq? vs. equal?
  - Higher-order functions
    - map, foldr, foldl

---

Functional Programming with Scheme

Read: Scott, Chapter 10.1 – 10.6

Scott, 3.6

---

Today's Lecture Outline

- Scheme
  - Exercises with map, foldl and foldr
  - Tail recursion
  - Binding with let and let*
  - Scoping in Scheme, closures
  - Scoping, revisited

---

Exercises

(foldr op lis id)

( e1 ... en-1 en ) id
( e1 ... en-1 ) res1
...
( e1 ) resn-1

Write rev, which reverses a list, using a single call to foldr
(define (rev lis) (foldr ... ))

Exercises

(foldl op lis id)

id ( e1 e2 e3 ... en )
id1 ( e2 e3 ... en )
id2 ( e3 ... en )
...

Write len, which computes length of list, using a single call to foldl
(define (len lis) (foldl ... ))
Exercises

(define (foldl op lis id)
  (if (null? lis) id
     (foldl op (cdr lis) (op id (car lis)))))

- Write `flatten3` using only `cond`, `atom?`, `null?`, `map`, `append` and `foldl/foldr`
- Write `flatten4` this time using only `cond`, `atom?`, `null?`, `append` and `foldl`.

Exercises

- Write a function that counts the appearances of symbols `a`, `b` and `c` in a set of flat lists
  - `(count-sym '(a b) (c a) (a b d))` yields `(a 3) (b 2) (c 1)`
  - Natural idea: use `map` and `fold`
- `map` and `fold` (also known as `reduce`), are the inspiration for Google’s MapReduce model
- Canonical MapReduce example [Dean and Ghemawat OSDI’04], is counting of appearances of certain words in a set of documents

foldr vs. foldl

(define (foldr op lis id)
  (if (null? lis) id
     (op (car lis) (foldr op (cdr lis) id))
  )

(define (foldl op lis id)
  (if (null? lis) id
     (foldl op (cdr lis) (op id (car lis)))
  )

- Compare underlined portions of these 2 functions
  - `foldr` contains a recursive call, but it is not the entire return value of the function
  - `foldl` returns the value obtained from the recursive call to itself.

Tail Recursion

- If the result of a function is computed without a recursive call OR it is the result of an immediate recursive call, then the function is said to be tail recursive
  - E.g., `fold`
- Tail recursion can be implemented efficiently
  - Result is accumulated in one of the arguments, and stack frame creation can be avoided!

Tail Recursion: Two Definitions of Length

(define (len lis)
  (if (null? lis) 0
     (+ 1 (len (cdr lis)))))

(define (lenh lis tot)
  (if (null? lis) tot
     (lenh (cdr lis) (+ 1 tot))))

(len '(3 4 5))

(lenh '(3 4 5))

Lenh is tail recursive. tot accumulates the length
Tail Recursion: Two Definitions of Factorial

- **fact1**:
  ```scheme
  (define (fact1 n)
    (cond ((zero? n) 1)
          ((eq? n 1) 1)
          (else (* n (fact1 (- n 1))))))
  ```

- **fact2**: Uses an accumulator.
  ```scheme
  (define (fact2 n acc)
    (cond ((zero? n) 1)
          ((eq? n 1) acc)
          (else (fact2 (- n 1) (* n acc))))
  ```

**fact2** is tail recursive.

Higher-order Functions

- **Functions can be return values**
  ```scheme
  (define (plus-2) (lambda (a) (+ a 2)) )
  ```

- **What is plus-2?**
  - A higher-order function that takes 0 arguments and returns a function of 1 argument

- **What does call (plus-2 2) yield?**
  - plus-2: Arity mismatch
  - What does ((plus-2) 2) yield?
    - 4

Let Expressions

- **let** and **let** expressions define a binding between each Var and the S-exp value, which holds during execution of S-exp
- **let** evaluates the S-exprs in parallel; **let** evaluates them from left to right.
- Associate values with variables for the local computation

- **let** expression:
  ```scheme
  (let ((x 2)) (* x x))
  ```
yields 4

- **let** expression:
  ```scheme
  (let ((x 2)) (let ((y 1)) (+ x y)) )
  ```
yields what?

- **let** expression:
  ```scheme
  (let ((x 10) (y (* 2 x))) (* x y))
  ```
yields what?

- **let** expression:
  ```scheme
  (let* ((x 10) (y (* 2 x))) (* x y))
  ```
yields what?

Let Introduces Nested Scope

- **let** expression:
  ```scheme
  (let ((x 10))
    (let ((f (lambda (a) (+ a x))))
      (f 5)))
  ```

causes x to be bound to 10

causes f to be bound to the lambda expression

Assuming that Scheme uses static scoping, what would this expression yield?

Question

```scheme
(define (f z)
  (let* ((x 5) (f (lambda (z) (* x z)))
     (map f z)) )

What does this function do?
```

Answer: takes a list of numbers, z, and maps it to the x5 list. E.g., (f '(1 2 3)) yields (5 10 15).
Scoping in Scheme:
Two Choices

With static scoping it evaluates to

\[ (* x ((\lambda (a)(+ a x)) 3)) \rightarrow (* 2 ((\lambda (a)(+ a 10)) 3)) \rightarrow ??? \]

With dynamic scoping it evaluates to

\[ (* x ((\lambda (a)(+ a x)) 3)) \rightarrow (* 2 ((\lambda (a)(+ a 2)) 3)) \rightarrow ??? \]

Scheme Chose Static Scoping

\[ \text{let } (x \text{ 10}) \]
\[ \text{let } ((f (\lambda (a)(+ a x))))) \]
\[ \text{let } ((x 2)) \]
\[ (* x ((f 3)))) \]

F is a closure:
The function value: \((\lambda (a)(+ a x))\)
The environment: \(\{ x \rightarrow 10 \}\)

Scheme chose static scoping:

\[ (* x ((\lambda (a)(+ a x)) 3)) \rightarrow (* 2 ((\lambda (a)(+ a 10)) 3)) \rightarrow 26 \]

Closures

- A closure is a function value plus the environment in which it is to be evaluated
  - Function value: e.g., \((\lambda (x)(+ x y))\)
  - Environment consists of bindings for variables not local to the function; thus, closure can eventually be evaluated: e.g., \(\{ y \rightarrow 2 \}\)
- A closure can be used as a function
  - Applied to arguments
  - Passed as an argument
  - Returned as a value

Closures

- Normally, when let expression exits, its bindings disappear
- Closure bindings (i.e., bindings part of a closure) are special
  - When let exits, bindings become inactive, but they do not disappear
  - When closure is called, bindings become active
- Closure bindings are "immortal"

\[ \text{let } (x 5) \]
\[ \text{let } ((f (let ((x 10)) (\lambda (y) x)))) \]
\[ \text{list } (f x (f x)) \]

Closures

\[ \text{define } (gg z) \]
\[ \text{let* } ((x 2) (f (\lambda (y)(+ x y))) \text{ (map } f z)) \]

- gg is a closure:
  - Function value: \((\lambda z) (\text{map } f z))\)
  - Environment: \(\{ x \rightarrow 2; f \rightarrow (\lambda (y)(+ x y)) \}\)

\[ > (gg '(1 2 3)) \]

1. gg is evaluated to its function value: \((\lambda z) (\text{map } f z))\)
2. closure environment is expanded with parameter binding: \(\{ x \rightarrow 2; f \rightarrow (\lambda (y)(+ x y)); z \rightarrow '(1 2 3) \}\)
3. evaluation occurs and \((3 4 5)\) is returned

Question

Lisp, the predecessor of Scheme used dynamic scoping. What is one reason why dynamic scoping made sense?
Lecture Outline

- Scheme
  - Exercises with map, foldl and foldr
- Tail recursion
- Binding with `let` and `let*`
- Scoping in Scheme, Closures
- Scoping, revisited

Scoping, revisited (Scott, Ch. 3.6)

- We discussed the two choices for binding of non-local variables
  - Static scoping (early binding) and
  - Dynamic scoping (late binding)
- Most languages choose static scoping

Scoping, revisited

- When we discussed scoping earlier, we assumed that functions were third-class values (i.e., functions cannot be passed as arguments or returned from other functions)
- When functions are third-class values...
  - When functions are third-class values, the function's static reference environment (i.e., closure bindings) is available on the stack. Function cannot outlive its referencing environment!

Scoping, revisited

- Functions as first-class values
  - Static scoping is difficult. Function value may outlive static referencing environment!
  - Need “immortal” closure bindings
  - In languages that choose static scoping, local variables must have “unlimited extent” (i.e., when stack frame is popped, local variables do not disappear!)

Scoping, revisited

- In functional languages local variables typically have unlimited extent
- In imperative languages local variables have limited extent (i.e., when stack frame is popped, local variables disappear)
  - Imperative languages typically disallow truly first-class function values
More on Dynamic Scoping

- Dynamic scoping with shallow binding and deep binding

- Shallow binding
  - Reference environment for function/routine is not created until the function is called
  - i.e., all non-local references are resolved using the most-recent-frame-on-stack rule
  - Shallow binding is usually the default in languages with dynamic scoping
  - All examples of dynamic scoping we saw so far used shallow binding

- Deep binding
  - When a function/routine is passed as an argument, the code that passes the function/routine has a particular reference environment (the current one!) in mind. It passes this reference environment along with the function value (it passes a closure).

Group Exercise

(define A
  (lambda ()
    (let* ((x 2)
           (C (lambda (P) (let ((x 4)) (P))))
           (D (lambda () x))
           (B (lambda () (let ((x 3)) (C D)))))
     B)))

When we call (A) in the interpreter, what gets printed? What would get printed if Scheme used dynamic scoping with shallow binding? With deep binding?