Announcements

- HW6 due today
- HW7 is out
  - A team assignment
  - Submitty page will be up tonight
  - Functional correctness: 75%, "Comments": 25%

Last class

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

Functional Programming with Scheme

Read: Scott, Chapter 11, Scott, 3.6

Today's Lecture Outline

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

Exercises

(foldr op lis id)

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

Resₙ

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

Exercises

(foldl op 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

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

- Write `flatten3` using `map` and `foldl/foldr`.
- Write `flatten4` this time using `foldl` but not `map`.

#### foldr vs. foldl

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

- Compare underlined portions of these two 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.
- Scheme implementations are required to be "properly tail-recursive".

### Tail Recursion: Two Definitions of Length

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

- `len` is tail recursive.
- `acc` accumulates the length.
Tail Recursion: Two Definitions of Factorial

(define (factorial n)
  (cond ((zero? n) 1)
        ((eq? n 1) 1)
        (else (* n (factorial (- n 1))))))

(define (fact2 n acc)
  (cond ((zero? n) 1)
        ((eq? n 1) acc)
        (else (fact2 (- n 1) (* n acc)))))

(define (factorial n)
  (fact2 n 1))

fact2 is tail recursive

Let Expressions

Let-expr ::= \[ \text{let} \ \{ \text{Binding-list} \} \ S-expr1 \]
Let*-expr ::= \[ \text{let*} \ \{ \text{Binding-list} \} \ S-expr1 \]
Binding-list ::= \[ \text{Var} \ S-expr \]

- let and let* expressions define a binding between each Var and the S-expr value, which holds during execution of S-expr1
- let evaluates the S-exprs in current environment "in parallel"; Vars are bound to fresh locations holding the results
- let* evaluates the S-exprs from left to right
- Associate values with variables for the local computation

Questions

(let ((x 2)) (* x x)) yields 4
(let ((x 2))
  (let ((y 1)) (+ x y)) yields what?
(let ((x 10) (y (* 2 x))) (* x y)) yields what?
(let* ((x 10) (y (* 2 x))) (* x y)) yields what?

Let Introduces Nested Scopes

(let ((x 10))
  (let ((f (lambda (a) (+ a x)))
        (let ((x 2)) (f 5)))))

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

\[
(\text{define} \ f \ z)\\
(\text{let*} \ (x \ 5) \ (f \ (\lambda \ (z) \ (** \ x \ z))))\\
(map \ f \ z)
\]

What does this function do?

Answer: takes a list of numbers, \( z \), and maps it to the \( x^5 \) list. E.g., \((f \ '(1 \ 2 \ 3))\) yields \((5 \ 10 \ 15)\).

Scheme Chose Static Scoping

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

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

Scheme chose static scoping:
\((** \ x \ ((\lambda \ a) (+ \ a \ x)) ) \rightarrow (** \ x \ ((\lambda \ a) (+ \ a \ 3)) ) \rightarrow (** \ 2 \ ((\lambda \ a) (+ \ a \ 3)) ) \rightarrow ...\)

With static scoping it evaluates to
\((** \ x \ ((\lambda \ a) (+ \ a \ x)) ) \rightarrow (** \ x \ ((\lambda \ a) (+ \ a \ 3)) ) \rightarrow (** \ 2 \ ((\lambda \ a) (+ \ a \ 3)) ) \rightarrow ...\)

Closures

- A closure is a function value plus the environment in which it is to be evaluated
  - Function value: e.g., \((\lambda \ a \ (+ \ a \ y))\)
  - 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

Lecture Outline

- Scheme
  - Exercises with map, foldl and foldr
  - Tail recursion
    - Binding with let, let*, and letrec
  - Scoping in Scheme
  - Closures
  - Scoping, revisited
Scoping, revisited (Scott, Ch. 3.6)

- We discussed the two choices for mapping non-local variables to locations
  - Static scoping (early binding)
  - 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)

- Most languages choose static scoping

Functions as Third-Class Values and Static Scoping

- Functions as first-class values
- Static scoping is more involved. Function value may outlive static referencing environment!
- Therefore, 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!)

More on Dynamic Scoping

- Shallow binding vs. deep binding
- Dynamic scoping with shallow binding
  - Reference environment for function/routine is not created until the function is called
  - Imperative languages (Fortran, Pascal, C) disallow truly first-class function values
  - More and more languages do allow first-class functions, e.g., Java 8, C++11

- Shallow binding is usually the default in languages with dynamic scoping
- All examples of dynamic scoping we saw so far used shallow binding
More on Dynamic Scoping

- **Dynamic scoping with 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).

**Example**

```scheme
v : integer := 10
people : database
print_routine (p : person)
  if p.age > v
    write_person(p)
other_routine (db : database, P : procedure)
  v : integer := 5
  foreach record r in db
    P(r)
other_routine(people, print_routine) /* call in main */
```

**Exercise**

```scheme
(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? Dynamic scoping and deep binding?

**Evaluation Order**

```scheme
(define (square x) (* x x))
```

- **Applicative-order (also referred to as eager) evaluation**
  - Evaluates arguments before function value
    - (square (+ 3 4)) =>
    - (square 7) =>
    - (* 7 7) =>
    - 49

**So Far**

- Essential functional programming concepts
- Reduction semantics
- Lists and recursion
- Higher-order functions
  - Map and fold (also known as reduce)
  - Evaluation order

- Scheme
Coming Up

- Lambda calculus: theoretical foundation of functional programming
- Haskell
  - Algebraic data types and pattern matching
  - Lazy evaluation
  - Type inference
  - Monads