Functional Programming with Scheme

Keep reading: Scott, Chapter 11.1-11.3, 11.5-11.6, Scott, 3.6
Lecture Outline

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

- Scoping, revisited
Exercises

Write \( \text{rev2} \), which reverses a list, using a single call to \( \text{foldr} \):

\[
\text{(define (rev2 lis) (foldr ...))}
\]
Write `len3`, which computes length of list, using a single call to `foldl`

\[
\text{(define (len3 lis) (foldl \ldots))}
\]
Exercises

\[(\text{foldl} \ op \ \text{lis} \ \text{id}) \rightarrow (1 \ 2 \ 3)\]

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

- Write \text{flatten3} using \text{map} and \text{foldl/foldr}

\[
\text{(define (flatten3 lis)} \rightarrow \text{else (foldl append (map flatten3 lis) '()))}
\]

- Write \text{flatten4} this time using \text{foldl} but not \text{map}.
Ref. Implementation:

(define (flatten lis)
  (cond ((null? lis) '())
        ((atom? lis) (cons lis '()))
        (else (append (flatten (car lis))
                      (flatten (cdr lis))))))

(define (flatten4 lis)
  (cond ((null? lis) '())
        ((atom? lis) (list lis))
        (else (fold1 (lambda (x y)
                      (append x (flatten4 y)))
                     lis)))
Exercises

- Write a function that counts the appearances of symbols a, b and c in a list 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 (or map and reduce), are the foundation of Google’s MapReduce model
  - Canonical MapReduce example [Dean and Ghemawat OSDI’04] is WordCount
A tail expression is an expression that occurs in tail context. Defined inductively as follows:

- The body of function is a tail expression
- If (if e1 e2 e3) is a tail expression, then e2 and e3 are tail expressions

Examples

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

Programming Languages CSCI 4430, A Milanova/B. G. Ryder
A tail call is a tail expression that is a function call. E.g.,

\[
(\text{define } (\text{foldl } \text{op } \text{lis } \text{id})
  \quad (\text{if } (\text{null? } \text{lis}) \text{id}
    \quad (\text{foldl } \text{op } (\text{cdr } \text{lis}) (\text{op } \text{id } (\text{car } \text{lis})))))))
\]

A tail recursive function is a function whose “leaf” tail expressions are either returns or tail calls to itself (still informal)

Tail calls give rise to efficient implementation of Continuation Passing Style (CPS)
Tail Recursion, A Bit More

foldl, normal

foldl (a bc)
foldl (bc)
foldl (c)
foldl ()

foldl, optimized

foldl (id (op id a))
foldl (bc)
foldl (c)

proper tail recursion.
Lecture Outline

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

- Scoping, revisited
Let Expressions

Let-expr ::= \( \text{let} \ ( \text{Binding-list} \ ) \ S\text{-expr1} \ )

Let*-expr ::= \( \text{let*} \ ( \text{Binding-list} \ ) \ S\text{-expr1} \ )

Binding-list ::= \( \text{Var} \ S\text{-expr} \ ) \ \{ \ ( \text{Var} \ S\text{-expr} \ ) \ \} \\

- \text{let} and \text{let*} expressions define a binding between each Var and the S-expr value, which holds during execution of S-expr1
- \text{let} evaluates the S-exprs in Binding-list in current environment “in parallel”
- \text{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 Expressions

Letrec-expr ::= ( letrec ( Binding-list ) S-expr1 )
Binding-list ::= ( Var S-expr ) { ( Var S-expr ) }

- letrec Vars are bound to fresh locations holding undefined values; S-exprs are evaluated “in parallel” in augmented environment
- letrec allows for definition of mutually recursive functions

(letrec (( even? (lambda (n) (if (zero? n) #t (odd? (- n 1)))))
         ( odd? (lambda (n) (if (zero? n) #f (even? (- n 1)))))
         )
         (even? 88) )
Regions (Scopes) in Scheme

- **let**, **let** and **letrec** give rise to block structure.
- They have the same syntax but define different regions (scopes).
- **let**
  - Region where binding is active: body of **let**

\[
( \text{let ( ((v_1 \ S-expr_1) (v_2 \ S-expr_2)) } \ S-expr) )
\]
Regions (Scopes) in Scheme

- `let`, `let*` and `letrec` give rise to block structure.
- They have the same syntax but define different regions (scopes).
- `let*`
  - Region: all bindings to the right plus body of `let*`
Regions (Scopes) in Scheme

- `let`, `let*`, and `letrec` give rise to block structure.
- They have the same syntax but define different regions (scopes).
- `letrec`
  - Region: entire `letrec` expression.

```
(letrec ((v1 S-expr1) (v2 S-expr2)) S-expr)
```

Restriction: `v1`, `v2` cannot be used as values in `S-expr1` or `S-expr2`.
Let Introduces Nested Scopes

```
(let ((x 10)) ;causes x to be bound to 10
  (let ((f (lambda (a) (+ a x)))) ;causes f to be bound to a lambda expression
    (let ((x 2)) (f 5))))
```

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

```
(let ((x 2)) (f 5))
```

vs

```
(let ((x 2)) (f 5))
```

15
What does this function do?

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

With static scoping it evaluates to

\[
(* \ x \ ((\text{lambda} (a)(+ a x)) 3)) \rightarrow
(* 2 \ ((\text{lambda} (a)(+ a 10)) 3) ) \rightarrow ??? \ 26
\]

With dynamic scoping it evaluates to

\[
(* \ x \ ((\text{lambda} (a)(+ a x)) 3)) \rightarrow
(* 2 \ ((\text{lambda} (a)(+ a 2)) 3) ) \rightarrow ??? \ 10
\]
Scheme Chose Static Scoping

(let ((x 10))
  (let ((f (lambda (a) (+ a x))))
    (let ((x 2))
      (* x (f 3))))

f is a closure:
The function value: (lambda (a) (+ a x))
The environment: { x ® 10 } Ref. Environment is just the static link.

Scheme chose static scoping:

(* x (lambda (a)(+ a x) 3)) -->
(* 2 ((lambda (a)(+ a 10) 3)) ) -->(
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 so the 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”

```scheme
(let ((x 5))  
  (let ((f (let ((x 10)) (lambda () x)))  
    (list x (f x (f)))))  
(5 10 5 10)
```
Lecture Outline

- Scheme
  - Exercises with map, foldl and foldr
  - 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
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).

Functions as 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!
Functions as Third-Class Values and Static Scoping

```
program
  a, b, c: integer;
  procedure P
    c: integer;
    procedure S
      c, d: integer;
      procedure R
        ...
        end R;
      end S;
      R();
    end S;
    R();
  end P;
  procedure R
    a: integer;
    = a, b, c;
  end R;
  ...; P(); ...
end program

main
  ---
  ---
  a
  b
  c
  main.P
  c
  main.R
  a

Static Scoping:
  a bound to R.a,
  b to main.b,
  c to main.c
```
Scoping, revisited

- 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!)
Scoping, revisited

- In functional languages local variables typically have **unlimited extent**
- In imperative languages local variables typically have **limited extent** (i.e., when stack frame is popped, local variables disappear)
  - 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
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
    - 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
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

\[ v : \text{integer} := 10 \]

people : database

\[
\text{print\_routine}\ (p : \text{person}) \quad \text{if}\ p\text{.age} > v \quad \text{write\_person}(p)
\]

\[
\text{other\_routine}\ (db : \text{database}, P : \text{procedure}) \quad v : \text{integer} := 5
\]

\[
\text{foreach}\ \text{record}\ r\ \text{in}\ db \quad P(r)
\]

\[
\text{other\_routine}(\text{people, print\_routine}) \quad /* \text{call in main} */
\]
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? Dynamic scoping and deep binding?
Evaluation Order

(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
Evaluation Order

(define (square x) (* x x))

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

- Scheme uses applicative-order evaluation
So Far

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

- Scheme
Coming Up

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