Chapter 10

(10.3, 10.5)

Higher-Order Functions
Examples

Higher Order Functions

A function that either takes a function as an argument or returns a function as a result

(Functions are "first-class" values)

apply: Applies a function to a list of arguments (iterator)

```
(apply + '(1 2 3 4))
(define x '(2 4 6 5 9 1))
(apply max x)
```

Higher Order Functions

map: Returns a list which is the result of applying its first argument to each element of its second argument

```
(map odd? '(2 3 4 5 6))
(map (lambda (x) (* x x)) '(1 2 3 4 5))
(map car '((1 2) (3 4) (5 6)))
; binary ops need two lists of elements
(map + '(1 2 3 4) '(5 6 7 8))
```

Higher Order Functions

```
(define double-any (lambda (f x)
   (f x x)))

(double-any + 10)
(double-any cons 'a)
```

Compose

A function that takes two functions f and g as arguments and returns a new function that is the composition $f \circ g$

```
(define compose (lambda (f g)
  (lambda (x)
   (f (g x)))))

((compose car cdr) '(1 2 3 4 5))
((compose (lambda (x) (apply + x)) cdr) '(1 2 3 4 5))
```

Filtering Data

A filter function that takes a predicate and a list as arguments and returns a list of all elements satisfying the predicate.

```
(define filter (lambda (f x)
  (cond ((null? x) '())
     ((f (car x)) (cons (car x) (filter f (cdr x))))
     (else (filter f (cdr x))))))

(filter number? '(5 "hello" #t 9 '(1 2 3)))
(filter (lambda (x) (> x 0)) '(0 -1 3 -3 2 5 -1))
```

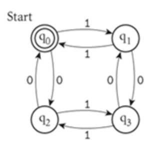
Towers of Hanoi

```
(define (hanoi n)
  (transfer 'A 'B 'C n)
)
(define (print-move from to)
  (display "Move disk from")
  (display from)
  (display " to ")
  (display to)
  (newline)
)
```

```
(define (transfer from to via n)
  (cond
    ((= n 1) (print-move from to))
      (else (transfer from via to
                      (- n 1))
         (print-move from to)
         (transfer via to from
                      (- n 1))
      )
   )
)

; Use:  (hanoi n)
```

Example program - Simulating a DFA

```
(define simulate
  (lambda (dfa input)
    (cons (current-state dfa)                ; start state
          (if (null? input)
              (if (infinal? dfa) '(accept) '(reject))
            (simulate (move dfa (car input)) (cdr input))))))

;; access functions for machine description:
(define current-state car)
(define transition-function cadr)
(define final-states caddr)
(define infinal?
  (lambda (dfa)
    (memq (current-state dfa) (final-states dfa))))

(define move
  (lambda (dfa symbol)
    (let ((cs (current-state dfa)) (trans (transition-function dfa)))
      (list
       (if (eq? cs 'error)
           'error
         (let ((pair (assoc (list cs symbol) trans)))
           (if pair (cadr pair) 'error)))   ; new start state
       trans                                ; same transition function
       (final-states dfa)))))               ; same final states
```




```
(define zero-one-even-dfa
 '(q0                                                    ; start state
   (((q0 0) q2) ((q0 1) q1) ((q1 0) q3) ((q1 1) q0)      ; transition fn
    ((q2 0) q0) ((q2 1) q3) ((q3 0) q1) ((q3 1) q2))
   (q0)))                                                ; final states
```

Figure 10.2 DFA to accept all strings of zeros and ones containing an even number of each. At the bottom of the figure is a representation of the machine as a Scheme data structure, using the conventions of Figure 10.1.

```
(simulate zero-one-even-dfa '(0 1 1 0 1))
```

Symbolic Computation

Scheme is excellent for symbolic computation

Write a *differentiate* function which takes an expression and a variable as input

- Must support *addition*, *subtraction*, *multiplication*, *division*

$$\frac{d}{dx}(c) = 0$$
$$\frac{d}{dx}(x) = 1$$
$$\frac{d}{dx}(u+v) = \frac{du}{dx} + \frac{dv}{dx}$$
$$\frac{d}{dx}(u-v) = \frac{du}{dx} - \frac{dv}{dx}$$
$$\frac{d}{dx}(u*v) = u\frac{dv}{dx} + v\frac{du}{dx}$$
$$\frac{d}{dx}(u/v) = \left(v\frac{du}{dx} + u\frac{dv}{dx}\right)/v^2$$

Symbolic Differentiation

```
(define (diff x expr)
  (cond ((not (pair? expr)) (if (eq? expr x)  1 0))
        (else (let ((op (car expr))
                    (u (cadr expr))
                    (v (caddr expr)))
                (cond ((eq? op '+) (list '+ (diff x u) (diff x v)))
                      ((eq? op '-) (list '- (diff x u) (diff x v)))
                      ((eq? op '*) (list '+ (list '* u (diff x v))
                                            (list '* v (diff x u))))
                      ((eq? op '/)
                         (list '/ (list '- (list '* v (diff x u))
                                           (list '* u (diff x v)))
                                  (list '* v v))))))))
```