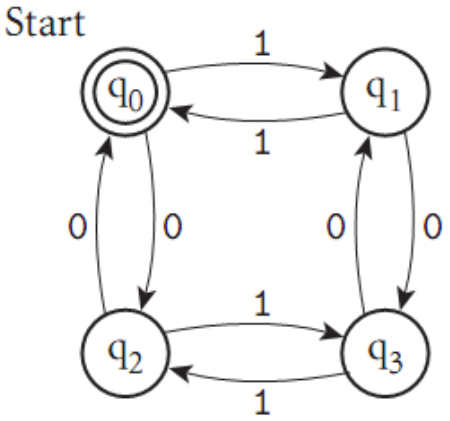
Functional Programming Demonstration

Brad Vander Zanden

A Review/Overview of Scheme

Example program - Simulation of DFA




```
(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.

A Review/Overview of Scheme

Example program - Simulation of 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
```

Figure 10.1 **Scheme program to simulate the actions of a DFA.** Given a machine description and an input symbol *i*, function `move` searches for a transition labeled *i* from the start state to some new state *s*. It then returns a new machine with the same transition function and final states, but with *s* as its "start" state. The main function, `simulate`, tests to see if it is in a final state. If not, it passes the current machine description and the first symbol of input to `move`, and then calls itself recursively on the new machine and the remainder of the input. The functions `cadr` and `caddr` are defined as `(lambda (x) (car (cdr x)))` and `(lambda (x) (car (cdr (cdr x))))`, respectively. Scheme provides a large collection of such abbreviations.