

CS 230
Programming Languages

10 / 31 / 2022

Instructor: Michael Eckmann

Today's Topics

- Questions? / Comments?
- More Scheme

Functional Programming

- when writing functions dealing with a list `lis`,
 - if you want to go through the elements one at a time,
 - use `(car lis)` to get the first
 - and recurse on `(cdr lis)`
 - also, you'll need to tell your function when to stop, that is, when the list is null, so `(null? lis)` should be the first thing you check.
- In general recursive functions need to
 - 1st check the base case (the case that will be true when the recursion should stop)
 - and in the recursive step(s) make sure that the function is working towards the base case (What do I mean here?)

Functional Programming

- Let's write functions to:
 - remove the first occurrence of an element of a list
 - remove all occurrences of an element from a list
 - replace the first occurrence of an element of a list with another element
 - replace all occurrences of an element of a list with another element

Functional Programming

- Scheme
- logical functions
 - and or not
- some more functions available in scheme:
 - number? list? boolean?
- for numbers:
 - real? integer? rational? complex? odd? even?
 - +, -, *, /, abs, sqrt, remainder, min, max, round, truncate, ceiling, floor
- some functions on lists
 - length, reverse
 - list-ref

Functional Programming

- Scheme
- examples:
 - (define mylist '(a b c))
 - mylist ; gives (a b c)
 - (list-ref mylist 2) ; gives c
 - (length mylist) ; gives 3
 - (reverse mylist) ; gives (c b a), but mylist is still (a b c)
 - reverse is not a deep reverse (i.e. it does not reverse any sublists)

Functional Programming

- Scheme
 - recall how I said cons always takes a list as the second parameter and returns the new list with the first parameter value attached to the front of the second parameter list
 - and car returns the first element of a list
 - and cdr returns the remainder of the list as a list
 - This is true for what are called PROPER LISTS

Functional Programming

- Scheme

- PROPER LISTS are those lists whose last cdr is the empty list.
- IMPROPER LISTS are those lists whose last cdr is something other than the empty list
- see handout (p. 18,19 from Dybvig's "The Scheme Programming Language")
- Anything created by consing two things together is a pair
- (cons 'a 'b) ; creates a dotted pair (an improper list)
- (cons 'a '(b)) ; creates a proper list (which is also a pair)
- (cons 'a (cons 'b '(c)))

Functional Programming

- Scheme

- PROPER LISTS are displayed in parens with a space between the elements

- e.g.

- (define proplist (cons 'a (cons 'b '(c))))

- proplist ; displays as (a b c)

- but more explicitly that list (a b c) can be thought of as:

(a . (b . (c . ())))

- where it is a nested set of pairs with dots between the car and the cdr of the pair

- Notice that the last cdr is the empty list.

Functional Programming

- IMPROPER LISTS are displayed with dots.
 - e.g.
 - (define improplist (cons 'a 'b))
 - improplist ; displays as (a . b)
 - (define improplist2 (cons 'a (cons 'b 'c)))
 - improplist2 ; displays as (a b . c)
 - but more explicitly improplist2 can be thought of as
(a . (b . c))
- (a b . c) is shorthand for that
 - (define improplist3 (cons (cons 'a 'b) 'c))
 - improplist3 ; displays as ((a . b) . c)

Functional Programming

- Scheme

- list? ; determines if the argument is a proper list

- pair? ; determines if the argument is a pair

- (list? '()) returns #t but (pair? '()) returns #f

- list? and pair? will return #t for non-empty proper lists

- list? will return #f for improper lists

- pair? will return #t for improper lists

- list? and pair? will return #f for non-pairs like

- (pair? 'a)

- (list? 'a)

Functional Programming

- Scheme

- associative lists can be created by creating a list of dotted pairs
- e.g.
- (define cardvalues '((ace . 1) (two . 2) (three . 3) (four . 4) (five . 5) (six . 6) (seven . 7) (eight . 8) (nine . 9) (ten . 10) (jack . 10) (queen . 10) (king . 10)))
- we can use assoc that takes in a key and an associative list and returns the dotted pair whose car is the key
- (assoc 'six cardvalues) ; would evaluate to (six . 6)
- also, if 'six wasn't the car of a pair then it would return #f
- to get the value 6 out of there what would we do?

Functional Programming

- Scheme

- associative lists can be created by creating a list of dotted pairs
- e.g.

```
(define cardvalues '((ace . 1) (two . 2) (three . 3) (four . 4)
(five . 5) (six . 6) (seven . 7) (eight . 8) (nine . 9) (ten . 10) (jack .
10) (queen . 10) (king . 10)))
```

- we can use `assoc` that takes in a key and an associative list and returns the dotted pair

```
(assoc 'six cardvalues) ; would evaluate to (six . 6)
```

- to get the value 6 out of there what would we do?

```
(cdr (assoc 'six cardvalues))
```

Functional Programming

- Let's add the cards code in DrRacket and try it out.

Functional Programming

- Scheme

- let

- creates local variables
 - Syntax:

- (let ((var val) ...) exp1 exp2 ...)

- e.g.
 - (let ((x 5) (y 7)) (+ x y)) ; x and y are local
 - (let ((z (* 6 6))) (+ z z)) ; z is local
 - without let we would write:
 - (+ (* 6 6) (* 6 6)) ; and the * would be performed twice

Functional Programming

- Scheme

- let

```
(let ((x 2))
```

```
  (display (+ x 3))
```

```
  (display "\n")
```

```
  (display x)
```

```
)
```

; x's value is not changed throughout the code above except

; when it is first given the value 2

(display x) ; x is local to the above code so this fails

Functional Programming

- Scheme

- set!

- performs assignments on previously declared variables
 - not a pure functional language feature
 - let's look at the function on the handout written with and without set!
 - (p. 46,47 from Dybvig's "The Scheme Programming Language")