CS 230
Programming Languages

11 / 01 / 2022

Instructor: Michael Eckmann

Today's Topics

- Questions? / Comments?
- Handouts
- More Scheme
- Introduce the parser homework

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")

Functional Programming

- Scheme
  - let's write more functions
  - count the number of #t's in a list of booleans

```
(count-t '(#t #t #f #f #t #t #f))  ; should return 4
```

Map

- map
  - a function that takes two parameters which are a function and a list
  - map applies the function to each element of the list and returns a list of the returned values
  - the following returns (#t #t #t #f #f #f #f #t)
  - (map number? '(1 2 3 a b r tttt 4.5))

  - another example:
  - (map floor '(1 2 3 4 4.5 4.6 7.8))

Apply

- apply
  - a function that takes two parameters which are a function and a list
  - it applies the function to all elements of the list and returns whatever the function would have returned if called using those elements of this list as individual parameters.

  - only functions that take a collection of arguments, not as a list, individually (like + and <) can be passed to apply

  - (apply < '(3 4 5 1)) ; ok, because (< 3 4 5 1) is a valid call.
  - (apply + '(4 5 6 7)) ; ok, because (+ 4 5 6 7) is a valid call.