Third Racket Programming Assignment

Abstract

This programming assignment mainly focuses on interacting with various methods and concepts from Historical Lisp and Racket. In the Historical Lisp section we look at interactions of older Lisp methods and concepts within a Racket session. The rest of the document contains definitions and interactions of basic list processing presented within various Racket sessions.

Task 1 - Historical Lisp

Quote and Eval:

Interactions – Constants 9 and "red" 'red

```
> 9
9
> "red"
"red"
> 'red
'red
```

Interactions – Variants of the quote special form

```
> (quote red)
'red
> 'red
'red
```

Interactions – Illustrating the "unbound variable" error

```
> red
red: undefined;
 cannot reference an identifier before its definition
> (quote red)
'red
```

Interactions - Examples of standard form evaluation

```
> (+ 3 4)
7
> (+ (/ 3 1) (* 2 2))
7
> (+ 1 2 3 4 5 6 7 8 9 10)
55
> (/ (* 10 (+ 10 1)) 2)
55
```

Interactions - Illustrating the "unbound function" error


```
> (red yellow blue)
```
red: undefined;
cannot reference an identifier before its definition

Car, Cdr, and Cons:

Interactions - Examples of the car function

```
> (car '(apple peach cherry))
'apple
> (car '((lisp 1959) (prolog 1971) (haskell 1990)))
'(lisp 1959)
```

Interactions - Examples of the cdr function

```
> (cdr '(apple peach cherry))
'(peach cherry)
> (cdr '((lisp 1959) (prolog 1971) (haskell 1990)))
'((prolog 1971) (haskell 1990))
```

Interactions - Examples of the cons function

```
> (cons 'apple '(peach cherry))
'(apple peach cherry)
> (cons '(lisp 1959) '((prolog 1971) (haskell 1990)))
'((lisp 1959) (prolog 1971) (haskell 1990))
```

Eq and Atom:

Interactions - Examples of the eq? function

```
> (eq? 'a 'b)
#f
> (eq? 'a 'a)
#t
```

Interactions - Examples of the atom? function

```
> (define (atom? x) (not (or (pair? x) (null? x))))
> (atom? 'a)
#t
> (atom? '(a b c))
#f
> (atom? 4)
#t
> (atom? '(a . b))
#f
```

Lambda:

Interactions - Interactions featuring lambda function application

```
> ( ( lambda (x) ( * x x ) ) 5 )
25
> ( ( lambda (x) ( * x x ) ) 9 )
81
> ( ( lambda ( x y ) ( cons x ( cons x ( cons y ( cons y '() ) ) ) ) ) 1 2 )
'(1 1 2 2)
> ( ( lambda ( x y ) ( cons x ( cons x ( cons y ( cons y '() ) ) ) ) ) 'hey 'now )
'(hey hey now now)
> ( ( lambda ( a b c )
( define s ( / ( + a b c ) 2.0 ) )(* s (- s a ) ( - s b ) (- s c ) ) ) 3 4 5)
36.0
```

Define:

Definitions - Defining four items, two variables and two functions

```
( define lisp-born 1959 )
( define favorite-pies '(cherry peach apple))
( define square ( lambda ( x ) ( * x x ) ) )
( define seeing-double
( lambda ( x y ) ( cons x ( cons x ( cons y ( cons y '() ) ) ) ) ))
```

Interactions - Referencing the two variables and applying the two functions

```
> lisp-born
1959
> favorite-pies
'(cherry peach apple)
> (square 5)
25
> (square 11)
121
> (seeing-double 'meow 'woof)
'(meow meow woof woof)
> (seeing-double 'oh 'no)
'(oh oh no no)
```

Definitions - Redefining the two functions

```
( define ( square x ) ( * x x ) )
( define ( seeing-double x y )
( cons x ( cons x ( cons y ( cons y '() ) ) ) ))
```

Interactions -Illustrating the application of these functions

```
> (square 6)
36
> (square 12)
144
> (seeing-double 'water 'ice)
'(water water ice ice)
> (seeing-double 'flag 'pole)
'(flag flag pole pole)
```

Definitions - Defining the area-of-circle function

```
( define ( square x ) ( * x x ) )
( define ( area-of-circle diameter )
( define radius ( / diameter 2 ) )
( define radius-squared ( square radius ) )
( define the-area ( * pi radius-squared ) )
the-area
)
```

Interactions - Testing the area-of-circle function

```
> (area-of-circle 20)
314.1592653589793
```

Cond:

Definitions - Defining the rgb, determine, and got-milk?
functions

```
( define ( rgb color-name )
   ( cond
      ( ( eq? color-name 'red ) '( 255 0 0 ))
      ( ( eq? color-name 'green ) '( 0 255 0 ))
      ( ( eq? color-name 'blue ) '( 0 0 255 ))
      ( ( eq? color-name 'purple ) '( 106 13 173 ))
      ( ( eq? color-name 'yellow ) '( 255 255 0 ))
      ( else 'unknown-color-name)))

( define ( determine operator operand )
   ( cond
      (( eq? operator 'difference )
       ( define maximum ( max ( car operand ) ( cadr operand ) ( caddr operand ) ) )
       ( define minimum ( min ( car operand ) ( cadr operand ) ( caddr operand ) ) )
       ( - maximum minimum ))
      (( eq? operator 'average )
       ( define sum ( + ( car operand ) ( cadr operand ) ( caddr operand ) ) )
       ( / sum ( length operand ) ) ) ) ) )

( define ( got-milk? list )
   ( cond
      ( ( null? list ) #f )
      ( ( eq? 'milk ( car list ) ) #t )
      ( else ( got-milk? ( cdr list ) ) ) ) ) )
```

Interactions - Mimicking the demo illustrating application of
the three functions

```
> (rgb 'blue)
'(0 0 255)
> (rgb 'yellow)
'(255 255 0)
> (rgb 'purple)
'(106 13 173)
> (rgb 'orange)
'unknown-color-name
> (determine 'difference '(11 100 55))
89
> (determine 'difference '(5 20 -1))
21
> (determine 'average '(1 2 9))
4
> (determine 'average '(9 5 22))
12
> (got-milk? '(coffee))
#f
> (got-milk? '(coffee with cream))
#f

> (got-milk? '(coffee with milk))
#t
```

Task 2 - Referencers and Constructors

Racket Session featuring CAR, CDR, and CONS:

Interactions - Applying CAR, CDR, and CONS
```
> ( car '( red green blue ))
'red
> ( cdr '( red green blue ) )
'(green blue)
> ( car '( ( 1 3 5 ) seven nine ) )
'(1 3 5)
> ( cdr '( ( 1 3 5 ) seven nine ) )
'(seven nine)
> ( car '( "Desde El Alma" ) )
"Desde El Alma"
> ( cdr '( "Desde El Alma" ) )
'()
> ( cons 'ESPRESSO '( LATTE CAPPUCCINO ) )
'(ESPRESSO LATTE CAPPUCCINO)
> ( cons '( a b c ) '( 1 2 3) )
'((a b c) 1 2 3)
> ( cons 'SYMBOL '() )
'(SYMBOL)
```

Referencing a list element:

Interactions - Referencing a list element from scratch
```
> ( define animals '(ant bat cat dog eel) )
> ( define questions '(who what when where why) )
> animals
'(ant bat cat dog eel)
> questions
'(who what when where why)
> ( car ( cdr ( cdr ( cdr animals ) ) ) )
'dog
> ( car ( cdr ( cdr ( cdr questions ) ) ) )
'where
```

Interactions - Referencing a list element from using list-ref

```
> (define animals '(ant bat cat dog eel ) )
> ( define questions '(who what when where why) )
> animals
'(ant bat cat dog eel)
> questions
'(who what when where why)
> (list-ref animals 3)
'dog
> (list-ref questions 3)
'where
```

Creating a list:

Interactions - Creating a list from scratch

```
> ( define a ( random 10 ) )
> ( define b ( random 10 ) )
> ( define c ( random 10 ) )
> (cons a (cons b (cons c '())))
'(1 8 2)
```

Interactions - Creating a list using list

```
> ( define a ( random 10 ) )
> ( define b ( random 10 ) )
> ( define c ( random 10 ) )
> (list a b c)
'(1 3 9)
```

Appending one list to another list:

Interactions - Appending two lists from scratch

```
> (define x '(one fish))
> (define y '(two fish))
> x
'(one fish)
> y
'(two fish)
> ( cons ( car x ) ( cons ( car ( cdr x ) ) y ) )
'(one fish two fish)
```

Interactions - Appending two lists using append

```
> (define x '(one fish))
> (define y '(two fish))
> x
'(one fish)
> y
'(two fish)
> (append x y)
'(one fish two fish)
```

Redacted Racket Session Feature Referencers and Constructors:

Interactions - Redacted Session
> (define languages '(racket prolog haskell rust))
> languages
> '(racket prolog haskell rust)

> 'languages
> 'languages

> (quote languages)
> 'languages

> (car languages)
> 'racket

> (cdr languages)
> '(prolog haskell rust)
> (car (cdr languages))
> 'prolog

> (cdr (cdr languages))
> '(haskell rust)

> (cadr languages)
> 'prolog

> (cddr languages)

```
> '(haskell rust)

> ( first languages )
> 'racket

> ( second languages )
> 'prolog

> ( third languages )
> 'haskell

> ( list-ref languages 2 )
> 'haskell

> ( define numbers '(1 2 3) )
> ( define letters '(a b c) )
> ( cons numbers letters )
> '((1 2 3) a b c)

> ( list numbers letters )
> '( (1 2 3) (a b c))

> ( append numbers letters )
> '(1 2 3 a b c)

> ( define animals '(ant bat cat dot eel) )
> ( car ( cdr ( cdr ( cdr animals ) ) ) )
> 'dot

> ( cadddr animals )
> 'dot

> ( list-ref animals 3 )
> 'dot
```

```
> ( define a 'apple )
> ( define b 'peach )
> ( define c 'cherry )
> ( cons a ( cons b ( cons c '() ) ) )
> '(apple peach cherry)

> ( list a b c )
> '(a b c)

> ( define x '(one fish) )
> ( define y '(two fish) )
> ( cons ( car x ) ( cons ( car ( cdr x ) ) y ) )
> '(one fish two fish)

> ( append x y )
> '(one fish two fish)
```

Task 3 - Random Selection

Definitions - Defining the sampler program

```
(define (sampler)
  (display "(?): ")
  (define the-list (read))
  (define the-element (list-ref the-list (random (length the-list))))
  (display the-element ) (display "\n")
  (sampler))
```

Interactions - Mimicking the sampler program
```
> (sampler)
(?): ( red orange yellow green blue indigo violet )
green
(?): ( red orange yellow green blue indigo violet )
red
(?): ( red orange yellow green blue indigo violet )
green
(?): ( red orange yellow green blue indigo violet )
green
(?): ( red orange yellow green blue indigo violet )
yellow
(?): ( red orange yellow green blue indigo violet )
orange
(?): ( aet ate eat eta tae tea )
tea
(?): ( aet ate eat eta tae tea )
eta
(?): ( aet ate eat eta tae tea )
aet
(?): ( aet ate eat eta tae tea )
tea
(?): ( aet ate eat eta tae tea )
eta
(?): ( aet ate eat eta tae tea )
ate
(?): ( 0 1 2 3 4 5 6 7 8 9 )
1
(?): ( 0 1 2 3 4 5 6 7 8 9 )
8
(?): ( 0 1 2 3 4 5 6 7 8 9 )
1
(?): ( 0 1 2 3 4 5 6 7 8 9 )
4
(?): ( 0 1 2 3 4 5 6 7 8 9 )
2
(?): ( 0 1 2 3 4 5 6 7 8 9 )
7
```

Task 4 - Playing Card Programming Challenge

Definitions - Programming the card playing functionality

```
(define (ranks rank)
  (list
    (list rank 'C)
    (list rank 'H)
    (list rank 'D)
    (list rank 'S)))

(define (deck)
  (append
    (ranks 2 )
    (ranks 3 )
    (ranks 4 )
    (ranks 5 )
    (ranks 6 )
    (ranks 7 )
    (ranks 8 )
    (ranks 9 )
    (ranks 'X )
    (ranks 'J )
    (ranks 'Q )
    (ranks 'K )
    (ranks 'A )))

(define (pick-a-card cards)
  (list-ref cards (random (length cards))))

(define (show card)
  (display (rank card))
  (display (suit card)))

(define (rank card)
   (car card))

(define (suit card)
   (cadr card))

(define (red? card)
   (or
     (equal? (suit card) 'D)
     (equal? (suit card) 'H)))

(define (black? card)
   (not (red? card)))

(define (aces? card1 card2)
   (and
     (equal? (rank card1) 'A)
     (equal? (rank card2) 'A)))
```

Interactions - Mimicking the card playing functionality

```
> (define c1 '(7 C))
> (define c2 '(Q H))
> c1
'(7 C)
> c2
'(Q H)
> (rank c1)
7
> (suit c1)
'C
> (rank c2)
'Q
> (suit c2)
'H
> (red? c1)
#f
> (red? c2)
#t
> (black? c1)
#t

> (black? c2)
#f
> ( aces? '( A C ) '( A S ) )
#t
> ( aces? '( K S ) '( A C ) )
#f
> ( ranks 4 )
'((4 C) (4 H) (4 D) (4 S))
> ( ranks 'K )
'((K C) (K H) (K D) (K S))
> (length (deck))
52
> (display (deck))
((2 C) (2 H) (2 D) (2 S) (3 C) (3 H) (3 D) (3 S) (4 C) (4 H) (4 D) (4 S) (5 C) (5 H) (5 D) (5 S)
(6 C) (6 H) (6 D) (6 S) (7 C) (7 H) (7 D) (7 S) (8 C) (8 H) (8 D) (8 S) (9 C) (9 H) (9 D) (9 S)
(X C) (X H) (X D) (X S) (J C) (J H) (J D) (J S) (Q C) (Q H) (Q D) (Q S) (K C) (K H) (K D) (K S)
(A C) (A H) (A D) (A S))
> (pick-a-card (deck))
'(4 H)
> (pick-a-card (deck))
'(Q H)
> (pick-a-card (deck))
'(3 H)
> (pick-a-card (deck))
'(K S)
> (pick-a-card (deck))
'(Q D)
> (pick-a-card (deck))
'(2 H)
```