Lab 1: Introduction to Racket

Seven Lewis

4/3/24

Part 2

1

What do car and cdr do? That last line of the above should cause an error. Explain why the single quote necessary and what it does.

```
(car '(11 12 13 14))     ; 11
(car '(a b c d))         ; 'a
(cdr '(11 12 13 14))     ; 11
(cdr '(a b c d))         ; '(b c d)
(car (11 12 13 14))      ; invalid
(cdr (a b c d))          ; invalid

; car returns the first symbol
; cdr returns a sub-list with the first symbol removed
; the single quote indicates that the following data should
; not be evaluated an instead be treated as a list of data.
```

2

Write sequences of cars and cdrs that will pick the symbol 'x out of the following expressions:

```
(car '(x y z m))
(car (cdr '(y x z m)))
(car (cdr (cdr (cdr '(y z m x)))))
(car (car (cdr '((y) (x) (z) (m)))))
(car (cdr (car (cdr '((y z) (m x))))))
```

3

Execute the following statements. Write down the return value for each statement and explain what each of the functions cons, append, and list does.

```
                         ; cons appears to add arg1 to the beginning of
                         ; arg2 (where arg 2 is a list)
(cons 3 '(1 2))          ; '(3 1 2)
(cons '(1 5) '(2 3))     ; '((1 5) 2 3)

                         ; list appears to take all the arguments and put
                         ; them in a list
(list 3 '(1 2))          ; '(3 (1 2))
```

```
(list '(1 5) '(2 3))     ; '((1 5) (2 3))

                         ; append adds to front each element in arg1 (which
                         ; is a list) to arg2's list
(append '(1) '(2 3))     ; '(1 2 3)
(append '(1 5) '(2 3))   ; '((1 5 2 3))

(cons 'x '(1 2))         ; '(x 1 2)
(list 1 2 3 '(4 5))      ; '(1 2 3 (4 5))
(cons '1 '2 '3 '(4 5))   ; INVALID, cons expects 2 arguments
```

4

Execute the following code. Write down the return value for each statement and explain what each of the functions length, reverse and member does.

```
                         ; length returns length of list
(length '(a b c))        ; 3

                         ; reverse returns reverse of list
(reverse '(a b c))       ; '(c b a)

                         ; member appears to return a new list beginning
                         ; with arg1 and succeeded by the rest of the
                         ; list after arg1, if the element is not in the
                         ; list, return #f
(member 'a '(a b c))     ; '(a b c)
(member 'b '(a b c))     ; '(b c)
(member 'c '(a b c))     ; '(c)
(member 'd '(a b c))     ; #f
```

Part 4

1

Write down the return value for each of the statement and explain the difference among equal?, eqv?, and =.

```
                                    ; equal? does a value and type
                                    ; equality comparison and if
                                    ; both are equal, return #t
(equal? '(hi there) '(hi there))    ; #t

                                    ; eqv? does a reference equality
                                    ; comparison (compare locations
                                    ; in memory)
(eqv? '(hi there) '(hi there))      ; #f

                                    ; = does a numeral equality comparison
                                    ; (with type coercion)
(= '(hi there) '(hi there))         ; error -> contract violation (strings
                                    ; cannot be coerced to numerals)
```

```
(equal? '(hi there) '(bye now))          ; #f
(eqv? '(hi there) '(bye now))            ; #f
(equal? 3 3)                             ; #t
(eqv? 3 3)                               ; #t
(= 3 3)                                  ; #t
(equal? 3 (+ 2 1))                       ; #t
(eqv? 3 (+ 2 1))                         ; #t
(= 3 (+ 2 1))                            ; #t
(equal? 3 3.0)                           ; #f
(eqv? 3 3.0)                             ; #f
(= 3 3.0)                                ; #t
(equal? 3 (/ 6 2))                       ; #t
(eqv? 3 (/ 6 2))                         ; #t
(= 3 (/ 6 2))                            ; #t
(equal? -1/2 -0.5)                       ; #f
(eqv? -1/2 -0.5)                         ; #f
(= -1/2 -0.5)                            ; #t
```

2

Enter the following code:

```
(if (equal? 8 3)
    9
    10
)                ; returns 10
```

Modify the condition following if to get a different value to return.

```
(if (equal? 8 (* 3 8/3))
    9
    10
)                ; returns 9
```

3

Enter the following code:

```
(cond
    ((equal? 16 3) (+ 3 8))
    ((equal? 16 8) 12)
    (else (* 6 3))
)
```

Write the return value for the above code. If replace all of the 16's in the above code with 8, what is the return value? What does the cond function do?

```
(cond
    ((equal? 16 3) (+ 3 8))
    (equal? 16 8) 12)
    (else (* 6 3))          ; <-- this line returns with val 18
)


(cond
    ((equal? 8 3) (+ 3 8))
    ((equal? 8 8) 12)        ; <-- this line returns with val 12
    (else (* 6 3))
)


(cond
   ((equal? 3 3) (+ 3 8))   ; <-- this line return s with val 11
   ((equal? 3 8) 12)
   (else (* 6 3))
)

; the cond function iterates over each argument, of which each is
; a (boolean, expression) pair.
; for each argument, if the boolean is #f, go to the next argument,
; if #t, then return the associated expression.
; the else clause will always return its expression if reached.
```

Part 5

1

```
(lambda (x)
    (+ x 1)
)
```

What does Racket return? The lambda function is an anonymous function. In this case, you have defined an anonymous function that takes a parameter x and adds 1 to it. Functions are also called procedures. Without parameter(s), the above function cannot do much. It can be used this way by given arguments:

```
(
    (lambda (x)
        (+ x 1)
    )
    3
)

(lambda (x)      ; returns nothing
    (+ x 1)
)


(                ; returns 4
    (lambda (x)
        (+ x 1)
    )
    3
)
```

2

Define named functions:

```
(define add-one
    (lambda (x)
        (+ x 1)
    )
)
```

Save this code to a file called "add-one.rkt", run it, then type (add-one 5) in the interactions window. The define statement created a pointer, called add-one, which points to the function you just created. Run this, and give the result. Run the following, and give the result. Does it serve the same purpose as the previous one?

```
(define (add-one x)
    (+ x 1)
)
```

```
; yes they serve the same purpose, one defines a symbol to refer the
; procedure (+ x 1) with parameter x, and the other defines a symbol
; to point to a lambda function that accepts one parameter: x
```

3

Add the following lines in the same file and run it. Given the result.

```
(define another-add-one add-one)
(another-add-one 5)
```

At the pointer level, what is happening here? Draw a picture in the answer box indicating what is happening.

```
(another-add-one 5) returns 6

[another-add-one] --> [add-one x] --> (+ x 1)

; we have the symbol another-add-one refer to the procedure add-one,
; this effectively creates an alias for the procedure add-one, then
; we evaluate add-one as normal
```

4

You can declare "local" variables in Racket via the use of the let function. For example, try the following code:

```
(define a 2);;binding a variable to a value
(define b 3)
(define c 4)
(define (strange x)
    (let ((a 1) (b 2))
        (+ x a b)
    )
)
```

After executing this code, what are the values of a, b, and c? What is the return value when you make the call (strange 4)? Explain your answers.

```
(strange 4)     ; the return value of (strange 4) is 7, this is because
                ; for this scope, the symbols 'a' and 'b' have been bound
                ; to '1' and '2' respectively
                ; 4 + 1 + 2 = 7
```