Functional Programming – Laboratory 2

Define new functions

Isabela Drămnesc

March 5, 2014

1 Concepts

- Local, global variables
- Assert, Equalities
- If, Cond
- Define new functions

2 Questions from laboratory 1

- What is the difference between cons, list, append?
- What is the internal representation of lists? For the list (today (is 5) March)
- What is the result for: > (cdar '((a (b c)) d ((e f) g) h))

3 Local and global variables

(let ((var expr) ...) body1 body2 ...)

```
>((let ((x 2) (y 3)) (+ x y))

> x

> y

> (let ([f +]) (f 10 20))

> (let ([+ *])
  (+ 20 30))

> (let ([x 1])
  (let ([x (+ x 1)])
    (+ x x)))
; variables defined in let are bound only inside the body of let
```

```
> (let ([x 1])
  (let ([new-x (+ x 1)])
    (+ new-x new-x)))
```

What is the result obtained after evaluating the following expression? Explain how the result is obtained. Rename the variables such that we will understand better the bindings.

```
(let ([x 9])
  (* x
     (let ([x (/ x 3)])
       (+ x x))))

(let ([x 'a] [y 'b])
  (list (let ([x 'c]) (cons x y))
        (let ([y 'd]) (cons x y))))
```

DEFINE:
(define (var0 var1 ... varn) e1 e2 ...)

```
(define abc '(a b c))

> abc

(define abcde '(a b c d e))

> abcde

(set! abcde (cdr abcde))

(let ([abcde '(a b c d e)])
  (set! abcde (reverse abcde))
   abcde)

> abcde

(let ([d 0]) (set! d '(a b c)) d)

> d

(define d '(a b c))

> d
```

(let* ((var expr) ...) body1 body2 ...) returns: the values of the final body expression

let* is similar to let except that the expressions expr ... are evaluated in sequence from left to right, and each of these expressions is within the scope of

the variables to the left. Use let* when there is a linear dependency among the values or when the order of evaluation is important.

```
 (let* ([x (* 5.0 5.0)]
        [y (- x (* 4.0 4.0))])
  (sqrt y))

(let ([x 0] [y 1])
  (let* ([x y] [y x])
    (list x y)))
```

4 Equality - nontrivial in Racket

```
; (eq? obj1 obj2)
; returns: true if obj1 and obj2 are identical, false otherwise

> (eq? 'a 3)

> (eq? #t 't)

> (eq? "abc" 'abc)

> (eq? "hello" '(hello))

> (eq? #f '())

> (eq? 9/2 4.5)

> (eq? 3 3.)

> (eq? '(a b c) '(a b c))

> (eq? 9/2 9/2)

> (let ([x (* 12345678987654321 2)])
  (eq? x x))

> (eq? #\a #\b)

> (eq? #\a #\a)

> (let ([x (string-ref "hi" 0)])
  (eq? x x))

> (eq? #t #t)

> (eq? #f #f)

> (eq? #t #f)
```

```
; The predicate null? returns true if its argument is the
; empty list () and false otherwise.

> (null? '())

> (null? 'abc)

> (null? '(x y z))

> (null? (cdddr '(x y z)))

> (eq? (null? '()) #t)

> (eq? (null? '(a)) #f)

> (eq? (cdr '(a)) '())

> (eq? 'a 'a)

> (eq? 'a 'b)

> (eq? 'a (string->symbol "a"))

> (eq? '(a) '(b))

> (eq? '(a) '(a))

> (let ([x '(a . b)]) (eq? x x))
```

(eqv? obj1 obj2) returns: true if obj1 and obj2 are equivalent, false otherwise

eqv? is similar to eq? except eqv? is guaranteed to return true for two characters that are considered equal by char=? and two numbers that are (a) considered equal by = and (b) cannot be distinguished by any other operation besides eq? and eqv?. A consequence of (b) is that (eqv? -0.0 +0.0) is false even though (= -0.0 +0.0) is true in systems that distinguish -0.0 and +0.0, such as those based on IEEE floating-point arithmetic.

Similarly, although 3.0 and 3.0+0.0i are considered numerically equal, they are not considered equivalent by eqv? if -0.0 and 0.0 have different representations.

```
(= 3.0+0.0i 3.0)

(eqv? 3.0+0.0i 3.0)

(eqv? #t #t)

(eqv? #f #f)

(eqv? #t #f)

(eqv? (null? '()) #t)
```

```
(eqv? (null? '(a)) #f)

(eqv? (cdr '(a)) '())

(eqv? 'a 'a)

(eqv? 'a 'b)

(eqv? 'a (string->symbol "a"))

(eqv? "abc" "cba")

(eqv? "abc" "abc")
```

(equal? obj1 obj2) returns: true if obj1 and obj2 have the same structure and contents, false otherwise

Two objects are equal if they are equivalent according to eqv?, strings that are string=?, bytevectors that are bytevector=?, pairs whose cars and cdrs are equal, or vectors of the same length whose corresponding elements are equal.

```
(equal? 'a 3)

(equal? #t 't)

(equal? "abc" 'abc)

(equal? "hi" '(hi))

(equal? #f '())

(equal? 9/2 7/2)

(equal? 3.4 53344)

(equal? 3 3.0)

(equal? 1/3 #i1/3)


(equal? 9/2 9/2)

(equal? 3.4 (+ 3.0 .4))

(let ([x (* 12345678987654321 2)])
  (equal? x x))

(equal? #\a #\b)

(equal? #\a #\a)
```

```
(let ([x (string-ref "hi" 0)])
  (equal? x x))

(equal? #t #t)

(equal? #f #f)

(equal? #t #f)

(equal? (null? '()) #t)

(equal? (null? '(a)) #f)

(let ([x '(a . b)]) (equal? x x))

(let ([x (cons 'a 'b)])
  (equal? x x))

(equal? (cons 'a 'b) (cons 'a 'b))

(equal? car car)
```

Conclusions:

- EQ? returns true if its arguments are the same and false otherwise.
- EQL?: two elements are EQL if they are EQ or if they are numbers of the same type;
- EQUAL?: two elements are EQUAL if they have the same structure and contents

5 Predicates with multiple arguments

```
> (> 77 10)

> (> 10 77)

> (>= 25 25)

> (<= 25 25)

> (>= 100 6)

> (>= 6 100)

> (< 1 2 3 4 5 6 7 8 9 10 11 13 17)

> (< 1 2 3 4 5 6 7 8 9 10 19 13 17)
```

6 If, Cond

if is used as follows:
(if <test> <then-expression> [<else-expression>])

```
(if #t 'true 'false)

(if #f 'true 'false)

(if '() 'true 'false)

(if 1 'true 'false)

(if '(a b c) 'true 'false)

>(if (> 3 2) (+ 4 5) (* 3 7))

> (if (< 3 2) (+ 4 5) (* 3 7))

> (if (+ 2 3) 1 2)


>(* 5 (if (null? (cdr '(x)))
        0
        (+ 11 12)))


>(* 5 (if (null? (cdr '(x y)))
         0
         (+ 11 12)))
```

cond :

```
(cond
    (<test_1> <consequence_1_1> <consequence_1_2> ...)
    (<test_2>)
    (<test_3> <consequence_3_1> ...)
    ...
)
```

Regarding if or cond, any expression <test-n> which is not false, evaluates to true, even if (equal? <test-n> t) is false.

```
> (cond ((= 2 3) 1) ((< 2 3) 2))

> (cond ((= 2 3) 1) ((> 2 3) 2) (#t 3))

> (cond ((= 2 3) 1) ((> 2 3) 2) (3))

> (cond ((= 2 2) (print 1) 8) ((> 2 3) 2) (#t 3))
```

Ask like this:

```
(cond (x 'b) (y 'c) (t 'd))
```

If x=true (then returns b)

If x=false, y = true (then returns c)

If x=false, y=false (then returns d)

Other example:

```
(let ([x 0] [y 0] [z 0])
(cond (x (set! x 1) (+ x 2))
      (y (set! y 2) (+ y 2))
      (#t (set! x 0) (set! y 0))
))
```

If x=true (then returns 3)Who is x?($x = 1$)

If x=false, y = true (then returns 4) Who is x and y? (false and 2)

If x=false, y=false (then returns 0) Who is x and y? (0 and 0)

Explain the evaluation of the following:

```
(let ([x 0] [y 0] [z 0])
(cond ((< x 0) (set! x 1) (+ x 2))
      ((< y 0) (set! y 2) (+ y 2))
      (#t (set! x 0) (set! y 0) (+ x y))
))
```

7 Define new functions

(define (<func-name> <param-list>)
(<expr-1> <expr-2> ... <expr-n>))
where
<expr-i>, i = 1, . . . , n represent the body of the function.

```
(define f1 (+ 2 3))
```

```
> f1
```

```
(define (f2 vara varb)
   (+ vara varb))
```

```
> (f2 3 4)
(define (sum-of-squares x y)
  (+ (* x x) (* y y)))
```

```
> (sum-of-squares 10 20)
```

```
(define (quadric-eq a b c)
```

```
    (let ([root1 0] [root2 0] [minusb 0] [radical 0] [divisor 0])
      (set! minusb (- 0 b))
      (set! radical (sqrt (- (* b b) (* 4 (* a c)))))
      (set! divisor (* 2 a))
      (set! root1 (/ (+ minusb radical) divisor))
      (set! root2 (/ (- minusb radical) divisor))
      (cons root1 root2)))

> (quadric-eq 2 -4 -6)

; or another version

(define (quadric-eq2 a b c)
    (let ([minusb (- 0 b)]
          [radical (sqrt (- (* b b) (* 4 (* a c))))]
          [divisor (* 2 a)])
      (let ([root1 (/ (+ minusb radical) divisor)]
            [root2 (/ (- minusb radical) divisor)])
        (cons root1 root2))))

> (quadric-eq2 2 -4 -6)

(define (is-even x)
             (equal? (modulo x 2) 0))

> (is-even 6)

> (is-even 7)

(define (abs n)
  (if (< n 0)
      (- 0 n)
      n
     )
  )
```

Write a function even-nb-divisible-by-7 such that:

```
> (even-nb-divisible-by-7 7)
true

> (even-nb-divisible-by-7 0)
true

> (even-nb-divisible-by-7 10)
true

> (even-nb-divisible-by-7 11)
false
```

```
> (even-nb-divisible-by-7 14)
true

> (define (the-third list)
          (car (cdr (cdr list))))

> (the-third '(r t y))

> (the-third '(r t y h))


>(length '(c b a))

; Define our "len" function which returns the length of a list

>(len '(c b a))

>(define (len list)
    (+ 1 (len (cdr list))))

>(len '(1 2 3 4))

;;; what is missing here???

;;; Let's define the recursive function correctly!

>(define (len list)
  (if (null? list)
        0                          ; base case
        (+ 1 (len (cdr list)))))  ; inductive case

>(len '(1 2 3 4))

>(len '(1 2 3 4))

>(len ())

>(len '(a))

>(len '(b a))

>(len '(c b a))

>(len '((a b c d e f) g h (i j) k))

>(len (car '((a b c d e f) g h (i j) k)))

; A function with two parameters which returns true
;if the first list is longer than the second list
```

```
> (define (longer-listp list1 list2)
(if
          (> (length list1) (length list2))
           #t
           #f
       )
)

> (longer-listp '(1 2 3) '(5 6))

> (longer-listp '(1 2 3) '(5 6 1 1 1 1))
```

8 Homework

Problem 1

Write a function which returns x to the power of y.

Problem 2

Write a function which returns $x * x * y * y$.

Problem 3

Write a function which counts the numbers from a list.

Problem 4

Write a function THE-SAME-ELEM with one parameter (a list) and which returns true if all the elements from a list are equal and false otherwise.

Problem 5

Write a function which takes two parameters (two lists) and which returns a new list obtained by appending the two lists. (Do not use append).

Problem 6

Write a function which calculates the factorial of a natural number.
Deadline: next lab.