Functional Programming – Laboratory 8
Lambda expressions, Circular lists, Mapping

Isabela Drămnesc

April 16, 2014

1 Concepts

- remove, remove*
- lambda expressions
- apply, funcall
- map, andmap, ormap, for-each, foldl

2 Questions from Laboratory 6

- Write the iterative definition in Lisp for:
  1. gcd(a,b)
  2. factorial(n)
  3. my-reverse(my-list)
  4. length(my-list)

3 Exercises

3.1 remove, remove*

```
(define l '(today is raining))

> (remove 'raining l)

> l

> (remove* '(-1) '(3 -1 3 -1 0 -1 2 4))

> (remove '(-1) '(3 -1 3 -1 0 -1 2 4))
```

3.2 Lambda expresions

We use them:

- when a function is used only once and it is to simple to write a separate definition for it;
- when the function has to be applied dynamically (is impossible to define it using DEFINE).

Syntax:
(lambda l f1 f2 f3 ... fn)
or
((lambda l f1 f2 f3 ... fn) par1 par2 ... parn)

- defines a function used locally;
- l represents the list of parameters; (can be given explicitly (a fixed number of parameters) or the list can have a variable number of parameters);
- f1 f2 ... fn the body of the function;

Exemple:

```
> ((lambda () 20))

> ((lambda (x) (+ 1 x)) 5)

> ((lambda (x y) (+ x y)) 5 7)

> ((lambda (y)
   ((lambda (x) (+ x (* 2 y)
                      (* 12 x y)
                      (* (* x x) (* y y))
                 )
      )
      21
     )
   )
   2
   )

> ((lambda (x y)
       (+ x
              (* 2 y)
              (* 12 x y)
              (* (* x x) (* y y))))
 21 2)
```

Lambda expressions can be considered anonymous functions (without name).

3.3 apply, funcall

There are cases when the number of parameters of a function has to be set dynamically. Applying a function on a set of parameters possibly dynamically synthesized is possible using the function APPLY.

Syntax:
(apply function (list-of-parameters))

Examples:

```
> (apply cons '(a b))

> (apply max '(1 2 3 4 5 6))

> (apply + '(1 2 3 4))

> (apply + 1 2 '(10))
```

Define a function (a version of the function apply) which allows applying a function to a fixed number of parameters.

Examples:

```
(define (funcall fun . args)
  (apply fun args))

> (funcall + 1 2 3 4)

> (apply + 1 2 3 4)
ERROR
> (apply + 1 2 3 '(4))

> (funcall (lambda (a b) (+ a b)) 2 3)

> (funcall newline)

> (apply newline)
ERROR

> (apply newline '())

> (funcall cons 'a 'b)

> (funcall max 1 2 3 4 5 6)

 (define p 'car)

> ((eval p) '(a b c))

  > (funcall (eval p) '(a b c))

 > (apply (eval p) '((a b c)))

(define f1
  (lambda (x) (+ x 3)))

> (funcall f1 5)
```

```
> f1
```

3.4 map, andmap, ormap, for-each, foldl

MAP is a function with global application. It is applied on each of the arguments from the list of arguments, the results are put into a list and returned when calling the function MAP. The evaluation is done when we reach the end of the shortest list.

```
> (map + '(1 2 3) '(1 2 3))

> (map + '(1 2) '(1 2 3))
ERROR

> (map + '(1 2 3) '(1 2))
ERROR

> (map + '(1 2 3) '(1 2 3) '(1 2 3))

> (map (lambda (number)
         (+ 1 number))
       '(1 2 3 4))
  '(2 3 4 5)

> (map (lambda (number1 number2)
         (+ number1 number2))
       '(1 2 3 4)
       '(10 100 1000 10000))

> (andmap positive? '(1 2 3))

> (andmap positive? '(1 2 a))
 ERROR

> (andmap positive? '(1 -2 a))

> (andmap positive? '(1 a -2))
   ERROR

> (andmap + '(1 2 3) '(4 5 6))

> (map + '(1 2 3) '(4 5 6))

> (andmap + '(1 2 3) '(4 5 6 10 20))
  ERROR

> (andmap odd? '(1 2 3))

> (andmap pair? '(1 2 3))
```

```
> (andmap pair? '((1) (2) (3)))

> (map cons '(1 2 3) '(4 5 6))

> (map car '((a b c) (x y z)))

> (map car '((a b c)))

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

> (ormap positive? '(1 2 a))

> (ormap + '(1 2 3) '(4 5 6))
```

```
(for-each (lambda (arg)
            (printf "The element ~a\n" arg)
            23)
          '(1 2 3 4))

(for-each
            (printf "The element \n")
           '(1 2 3 4))
```

ERROR

```
> (foldl cons '() '(1 2 3 4))

> (foldl + 2 '(1 2 3 4))

> (foldl cons '(a) '(1 2 3 4))

> (foldl cons '(a b) '(1 2 3 4))

> (foldl + 0 '(1 2 3 4))

> (foldl (lambda (a b result)
           (* result (- a b)))
         1
         '(1 2 3)
         '(4 5 6))
```

3.5 Circular lists

Study the following example:

```
(define list-len (x)
  (do ((n 0 (+ n 2))            ;Counter
       (fast x (cddr fast))     ;Fast pointer: goes from 2 to 2
       (slow x (cdr slow)))     ;Slow pointer: passes through each cdr
```

```
    (#f)
    ;; If the fast pointer reaches the end, then will return n.
    (when (null? fast) n)
    ;; If cdr of the fast pointer reaches the end, then returns n+1.
    (when (null? (cdr fast)) (+ n 1))
    ;; If the fast pointer catches up the slowly pointer,
    ;; means that we deal with a circular list.
    ;; We return the empty list.
    (when (and (eq? fast slow) (> n 0)) '()))))
```

and modify if needed.

3.6 Our own equality predicate

```
A function which decides the "equality" of two structures
constructed with cons:

(define (our-equal l1 l2)
  (cond ((and (null? l1) (null? l2)) #t)
        ((and (or (symbol? l1) (number? l1))
          (or (symbol? l2) (number? l2))) (equal? l1 l2))
        ((and (or (symbol? l1) (number? l1))
          (not (or (symbol? l2) (number? l2)))) #f)
        ((and (not (or (symbol? l1) (number? l1)))
          (or (symbol? l1) (number? l1))) #f)
        ((our-equal (car l1) (car l2)) (our-equal (cdr l1) (cdr l2)))
        (#t #t)
  )
)
```

4 Homework (deadline: next lab)

1. Evaluate the following expressions:

```
> (cadr '(a b c d e))

> (second '(a b c d e))

> (nth 2 '(a b c d e))

> (cadr (cadr '((a b c) (d e f) (g h i))))

> (cadadr '((a b c) (d e f) (g h i)))

> (cddadr '((a b c) (d e f) (g h i)))

> (last '((a b c) (d e f) (g h i)))

> (third '((a b c) (d e f) (g h i)))

> (cdr (third '((a b c) (d e f) (g h i))))
```

2. Define an iterative function RANGE with has as parameters a list of numbers and returns a list with the length 2 and the returned list contains the minimum element and the maximum element. Use the predicates > and < Ensure that your algorithm is linear. Write another function VALID-RANGE, which returns the same result as RANGE if the elements of the list are all numbers and INVALID otherwise.
Exemplu:

```
> (range '(0 7 8 2 3 -1))
(-1 8)
> (range '(7 6 5 4 3))
(3 7)
> (valid-range '('a 7 8 2 3 -1))
INVALID
> (valid-range '(0 7 8 2 3 -1))
(-1 8)
```

3. Write 2 s-expressions in order to access the symbol C for each of the following lists:

```
(a) (A B C D E)

(b) ((A B C) (D E F))

(c) ((A B) (C D) (E F))
```

4. Change C in SEE for each of the following lists:

```
(a) (A B C D E)

(b) ((A B C) (D E F))

(c) ((A B) (C D) (E F))

(d) (A (B C D) E F)
```