Programare Funcţională – Laborator 1
Introducere

Isabela Drămnesc

February 26, 2023

1 Concepte

- Programare funcţională
- Racket
- read-eval-print
- Variabile locale, globale, constante
- Egalitate
- If, Cond
- Definire funcţii

2 Software

- Descărcaţi de aici.

3 Introducere in Racket

La lansarea interpretorului este afişat un prompter (>), iar funcţionarea interpretorului se bazează pe repetarea ciclului de bază read-eval-print.

1. read: citeşte o expresie simbolică;
2. eval: evaluează expresia simbolică introdusă;
3. print: afişează rezultatul obţinut în urma evaluării expresiei.

3.1 Aritmetică

- *Numere întregi*: 5 999999999999 -18 ;
- *Numere float*: 2.03 sau in notatia: 6.023e+23;
- *Numere raţionale*: 3/4 12345678912345678/23 -17/3;
- *Numere complexe*: 2.0+3.03i 1+2i 3/4-1/2i

```
> 10
10

> "Hello, world!"
"Hello, world!"

> 13/52
1/4

> 3.14
```

In Racket, funcţiile $f[x, y]$ sunt definite ca: (f x y)
$x + y$ este defapt $+[x, y]$, scris ca (+ x y)
Exemplu: (+ 4 6)

```
> (+ 4 6)

> (+ 2 (* 3 4))

> (+ 3.14 2.71)

> (- 23 10)

> (- 10 23)

> (/ 30 3)

> (/ 25 3)

> (/ 3 6)
1/2         ;numar rational

> (/ 3 6.0)
0.5         ;numar de tip float

; cateva functii predefinite

> (max 4 6 5)

> (max 4 6 5 10 9 8 4 90 54 78)

> (min 8 7 3)

> (min 4 6 5 10 9 8 2 90 54 78)

> (expt 5 2)

> (expt 10 4)

> (sqrt 25)
```

```
> (sqrt 25.0)

> (sqrt -25)

> (sqrt -25.5)

> (abs -5)

> (+ (* 2 3 5) (/ 8 2))

> pi

> (truncate 17.678)
; retureaza componentul intreg al unui numar real

> (round 17.678)

> (remainder 14 5)

> (quotient 14 5)

   > (+ 2.5+2i 3+5i)
5.5+7.0i

; #t    #T    #true          pentru valoarea de adevar true
; #f    #F    #false         pentru valoarea de adevar false
```

3.2 Apostrof '

```
> 3
3       ; un numar se evalueaza la numarul insusi

> "hello"
HELLO       ; un sir de caractere se evalueaza la el insusi

> (+ 2 3)
5           ; se aplica + la 2 si 3

> a
ERROR: variable A has no value
;cauta sa evalueze pe a
```

Pentru a stopa evaluarea se foloseşte apostrof:

```
> '3

> '(+ 2 3)

> 'a
```

```
> (eval '(+ 2 3))   ; eval forteaza evaluarea

> '(2 3 4)

> (+ 10 20 30 40 50)

> '(eval '(+ 3 4))

>''3
```

Ce se întâmplă dacă scriem (2 3 4) în interpretorul Racket?
Cum putem afişa lista (2 3 4) în Racket?

3.3 Predicate predefinite

```
> (number? 2)

> (number? 22222222222222222222222222222222222222222222222222)

> (number? 'dog)

> (symbol? 'dog)

> (symbol? +)

> (symbol? '+)

> (symbol? '9)

> (rational? 3/4)

> (real? 3.141592)

> (complex? 3+2i)
```

and, or, not

```
> (and)
#t

> (or)
#f

> (not)
. . not: arity mismatch;
the expected number of arguments does not match the given number
expected: 1
given: 0
```

```
> (not #f)
#t

> (and (< 1 2) 3 'lala)
'lala

> (or (< 1 2) 3 'lala)
#t

> (not 3)
#f

> (and (= 1 2) (expt 2 50))
#f

> (or (= 1 2) (expt 2 50))
1125899906842624
```

Alte predicate

```
> (> 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)
```

3.4 Variabile, constante

(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)))
```

Care este rezultatul obtinut dupa evaluarea urmatoarelor expresii? Explicati cum a fost obtinut rezultatul. Redenumiti variabilele in asa fel incat sa se inteleaga mai bine cum se face legarea lor.

```
(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 (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* ([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 Egalitatea - netriviala în Racket

```
; (eq? obj1 obj2)

> (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)
```

```
> (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)

```
(= 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)

```
(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)
```

5 if, cond, when, unless

Forma specială pentru *if* este:
(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)))
```

Forma specială pentru *cond* este:

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

> (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))
```

Intrebam asa:

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

Daca x=true (atunci returneaza b)

Daca x=false, y = true (atunci returneaza c)

Daca x=false, y=false (atunci returneaza d)

Alt exemplu:

```
(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))
))
```

Daca x=true (atunci returneaza 3)Cine e x?($x = 1$)

Daca x=false, y = true (atunci returneaza 4) Cine sunt x si y? (false si 2)

Daca x=false, y=false (atunci returneaza 0) Cine sunt x si y? (0 si 0)

Explicati urmatorul cod. Cum se face evaluarea?

```
(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))
        ))
```

when, unless

```
(when pred block)   ; syntactic sugar pentru (cond [pred block]),

(unless pred block)    ; syntactic sugar pentru
```

Exemple:

```
> (when (< 1 2) "a")

> (unless (< 1 2) "a")

> (when (> 1 2) "a")

> (unless (> 1 2) "a")
```

6 Liste (car, cons, cdr)

Reprezentarea internă a listelor este dată de o structură arborescentă.
Listele sunt reprezentate ca:

Head (Cap) şi

Tail (Coadă).

Capul este un element, iar coada este o listă.
In Racket sunt 3 operaţii fundamentale pe liste:

```
Head(a b c d)=a —un element
Tail(a b c d)=(b c d) — o lista
Insert[a, (b c d)]=(a b c d)
```

- Head CAR
- Tail CDR
- Insert CONS

Construirea listelor utilizând:

- cons
- list
- append

cons:

```
> (cons 'a '())

> (cons 'a 'b)
;reprezentare in celule

> (cons 1 2 '())
ERROR          ; doar doua argumente poate avea cons

> (cons 32 (cons 25 (cons 48 '())))

> (cons 'a (cons 'b (cons 'c 'd)))

> (cons 'a (cons 'b (cons 'c '(d))))
```

list:

```
> (list 'a)

> (list 'a 'b)
```

```
>(list 32 25 48)
>(list a b c)
>(list 'a 'b 'c)
```

append:

```
> (append '(a) '(b))
```

car, cdr, cons:

```
> (car '(a b c))
> (cdr '(a b c))
> (car (cdr '(a b c d)))
> (car (cdr (car '((a b) c d))))
> (cdr (car (cdr '(a (b c) d))))
> (cdr (cons 32 (cons 25 (cons 48 '()))))
> (car (cons 32 (cons 25 (cons 48 '()))))
> (cdr (cdr (cons 32 (cons 25 (cons 48 '())))))
> (cdr (cdr (cdr (cons 32 (cons 25 (cons 48 '()))))))
> (cdr (cdr (cdr (cons 32 (list 32 25 48)))))
> (cdr (cdr (cdr (list 32 25 48))))
> (cddr '(astazi este soare))
> (caddr '(astazi este soare si cald))
> (cdr (car (cdr '(a (b c) d)))) ; echivalent cu (cdadr '(a (b c) d))
> (nthcdr 0 '(a b c d e)) ; aplica cdr de 0 ori
> (nthcdr 1 '(a b c d e)) ;aplica cdr o data
```

Alte exemple:

```
> (cons '+ '(2 3))
> (eval (cons '+ '(2 3)))
```

```
> (length '(1 2 d f))

> (reverse '(3 4 5 2))

> (append '(2 3) (reverse '(p l f)))

> (first '(s d r))

> (rest '(p o m))

> (last '(p o m))

> (member 'om '(un om citeste))

> (car (member 'sapte '(o saptamana are sapte zile)))
```

7 Definirea de noi functii

(define (<func-name> <param-list>)
(<expr-1> <expr-2> ... <expr-n>))
where
<expr-i>, i = 1, . . . , n reprezinta corpul functiei.

```
(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)
```

```
; sau alta versiune

(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))
```

8 Exerciţii:

1. Definiţi o funcţie even-nb-divisible-by-7 care se comportă astfel:

```
> (even-nb-divisible-by-7 7)
#t

> (even-nb-divisible-by-7 0)
#t

> (even-nb-divisible-by-7 10)
#t

> (even-nb-divisible-by-7 11)
#f

> (even-nb-divisible-by-7 14)
#t
```

2. Definiţi o funcţie care primeşte doi parametri şi returnează true dacă prima listă e mai lungă decât a doua.

```
> (longer-listp '(1 2 3) '(5 6))
#t
```

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

3. Utilizaţi car, cdr, şi combinaţii ale lor pentru a returna:

```
LISTA de input: (A (L K (P O)) I)
rezultat: O
respectiv  (O)

LISTA de input: (A ((L K) (P O)) I)
rezultat: O
respectiv  (K)

LISTA de input: (A (B C . D) (HELLO TODAY) I AM HERE)
rezultat    HELLO
respectiv    AM
```