

# Racket CheatSheet

## Laborator1

### Sintaxa Racket

(nume\_functie arg1 arg2 ...)

```
1 (max 2 3)      3
2 (+ 2 3)        5
```

### Tipuri de bază

Valori booleene: #t #f (sau true false)

Numere: 1 2 3 3.14 ...

Simboli (literali): 'a 'b 'abc 'non-alnum?!

### Operatori aritmetici

+ - \* / modulo quotient add1 sub1

```
1 (+ 1 2)      3
2 (- 7 2)      5
3 (* 2 11)     22
4 (/ 5 2)      2.5
5 (quotient 5 2) 2
6 (modulo 5 2)  1
7 (add1 4)      5
8 (sub1 4)      3
```

### Operatori relationali

< <= > >= = eq? equal? zero?

```
1 (< 3 2)      #f
2 (>= 3 2)     #t
3 (= 1 1)      #t      (numere)
4 (= '(1 2) '(1 2)) eroare
5 (equal? '(1 2) '(1 2)) #t      (valori)
6 (eq? '(1 2 3) '(1 2 3)) #f      (referinte)
7
8 (define x '(1 2 3))
9 (eq? x x)     #t
10
11 (zero? 0)     #t (true)
12 (zero? 1)     #f (false)
```

### Operatori logici

not and or

```
1 (not true)    #f
2 (not false)   #t
3 (or true false) #t
4 (and #f #t)   #f
```

### Constructorii liste

null '() cons list

```
1 '()           ()
2 null          ()
3
4 (cons 1 null) (1)
5 (cons 'a '(b c)) (a b c)
6
7 (list 1)       (1)
8 (list 1 2 3 4) (1 2 3 4)
```

### Operatori pe liste

car cdr first rest null? length member reverse  
append

```
1 (car '(1 2 3 4))      1
2 (first '(1 2 3 4))    1
3 (cdr '(1 2 3 4))      (2 3 4)
4 (rest '(1 2 3 4))     (2 3 4)
5 (cadr '(1 2 3 4 5))   2
6 (cdar '(1 2 3 4 5))   eroare
7 (cddr '(1 2 3 4 5))   (3 4 5)
8 (caddr '(1 2 3 4 5))  3
9
10 (null? null)          #t (true)
11 (null? '(1 2))        #f (false)
12
13 (length '(1 2 3 4))   4
14 (length '(1 (2 3) 4)) 3
15
16 (member 'a '(b c a d a e)) 'a d a e
17 (member 'f '(b c a d a e)) #f
18
19 (reverse '(1 (2 3) 4)) (4 (2 3) 1)
20
21 (list? '())           #t
22 (list? 2)             #f
23
24 (append '(1 2 3) '(4) '(5)) (1 2 3 4 5)
25 (append 1 '(2 3 4))        eroare
```

### take și drop

```
1 (take '(1 2 3 4) 2)      '(1 2)
2 (drop '(1 2 3 4 5) 2)    '(3 4 5)
3
4 (take-right '(1 2 3 4) 2) '(3 4)
5 (drop-right '(1 2 3 4 5) 2) '(1 2 3)
```

### Funcții anonime (lambda) și funcții cu nume

(lambda (arg1 arg2 ...) rezultat) (define nume val)

```
1 (lambda (x) x)          functia identitate
2 ((lambda (x) x) 2)      2 aplicare functie
3 (define idt (lambda (x) x)) legare la un nume
4 (define (idt x) x)      sintaxa alternativa
5 (idt 3)                  3
```

### Sintaxa if

```
1 (if test exp1 exp2)
2
3 (if (< a 0)
4     a
5     (if (> a 10) (* a a) 0))
```

### Sintaxa cond

```
1 (cond (test1 exp1) (test2 exp2) ... (else exp))
2
3 (cond
4   ((< a 0) a)
5   ((> a 10) (* a a))
6   (else 0))
```

### AȘA DA / AȘA NU

```
1 DA: (cons x L)      NU: (append (list x) L)
2                      NU: (append (cons x '()) L)
3 DA: (if c vt vf)     NU: (if (equal? c #t) vt vf)
4 DA: (null? L)        NU: (= (length L) 0)
5 DA: (zero? x)        NU: (equal? x 0)
6 DA: test             NU: (if test #t #f)
7 DA: (or ceva1 ceva2) NU: (if ceva1 #t ceva2)
8 DA: (and ceva1 ceva2) NU: (if ceva1 ceva2 #f)
```

### Programare cu funcții recursive

1. După ce variabilă(e) fac recursivitatea? (ce variabilă(e) se schimbă de la un apel la altul?)
2. Care sunt condițiile de oprire în funcție de aceste variabile?(cazurile “de bază”)
3. Ce se întâmplă când problema nu este încă elementară? (Obligatoriu aici cel puțin un apel recursiv)

### Folosiți cu încredere!

<http://docs.racket-lang.org/>

# Racket CheatSheet

## Laborator 2

### Recursivitate pe stivă

```
1 ; suma elementelor unei liste
2 (define (sum-list L)
3
4
5 ; aici nu avem nevoie de funcție auxiliară
6
7
8 (if (null? L)
9     0 ; la sfârșit creăm valoarea inițială
10    (+ (car L) (sum-list (cdr L))))
11 ; ^ construim rezultatul pe revenire
12 ; (după întoarcerea din recursivitate)
13 ))
14 ; fiecare apel recursiv întoarce rezultatul
15 ; corespunzător argumentelor
16
17
18 ; concatenarea a două liste
19 (define (app L1 L2)
20
21
22
23
24
25
26 (if (null? L1)
27     L2 ; când L1 este vidă, întoarcem L2
28     (cons (car L1) (app (cdr L1) L2)))
29 ; ^ construim rezultatul pe revenire))
```

- fiecare apel recursiv se pune pe stivă
- complexitate spațială  $O(n)$
- scriere mai simplă

### Recursivitate pe coadă

```
1 ; suma elementelor unei liste
2 (define (sum-list L)
3   (sum-list-tail 0 L)) ; <-- funcție ajutătoare
4                       ; ^ valoarea inițială pentru sumă
5
6                       ; în sum construim rezultatul
7 (define (sum-list-tail sum L)
8   (if (null? L)
9       sum ; la sfârșit avem rezultatul gata
10      (sum-list-tail
11        (+ sum (car L))
12        ; ^ construim rezultatul pe avans
13        ; (pe măsură ce intrăm în recursivitate)
14        (cdr L))))
15 ; funcția întoarce direct rezultatul apelului
16 ; recursiv -- toate apelurile recursive întorc
17 ; același rezultat, pe cel final
18
19 ; concatenarea a două liste
20 (define (app A B)
21   (app-iter B (reverse A)))
22 ; nevoie de funcție ajutătoare
23 ; rezultatul este construit în ordine inversă
24
25 (define (app-iter B Result)
26   (if (null? B) ; la sfârșit rezultatul e complet
27       (reverse Result) ; inversăm rezultatul
28       (app-iter (cdr B) (cons (car B) Result))))
29 ; construim rezultatul pe avans
```

- apelurile recursive nu consumă spațiu pe stivă – execuția este optimizată știind că rezultatul apelului recursiv este întors direct, fără operații suplimentare.
- complexitatea spațială este dată doar de spațiul necesar pentru acumulator – de exemplu la `sum-list-tail` complexitatea spațială este  $O(1)$ .
- scriere mai complexă, necesită de multe ori funcție auxiliară pentru a avea un parametru suplimentar pentru construcția rezultatului (rol de acumulator), mai ales dacă tipul natural de recursivitate al funcției este pe stivă.
  - **Atenție:** uneori, rolul acumulatorului poate fi preluat de unul dintre parametri, caz în care nu este nevoie nici de funcția suplimentară.
- rezultatul este construit în ordine inversă

### Sintaxa Racket

(nume\_functie arg1 arg2 ...)

```
1 (max 2 3)      3
2 (+ 2 3)        5
```

### AȘA DA / AȘA NU

1 DA: (cons x L)	NU: (append (list x) L)
2	NU: (append (cons x '()) L)
3 DA: (if c vt vf)	NU: (if (equal? c #t) vt vf)
4 DA: (null? L)	NU: (= (length L) 0)
5 DA: (zero? x)	NU: (equal? x 0)
6 DA: test	NU: (if test #t #f)
7 DA: (or ceva1 ceva2)	NU: (if ceva1 #t ceva2)
8 DA: (and ceva1 ceva2)	NU: (if ceva1 ceva2 #f)

### Imagini în Racket

image-height, overlay, flip-vertical

```
1 (overlay
2   (image-height (circle 20 "solid" "red")) ; => 40
3   (image-height (circle 20 "solid" "blue")) ; => 40
4   (image-height (circle 20 "solid" "green")) ; => 60
5   (image-height (circle 20 "solid" "yellow")) ; => 40
6   (image-height (circle 20 "solid" "cyan")) ; => 40
7   (image-height (circle 20 "solid" "magenta")) ; => 40
8   (image-height (circle 20 "solid" "brown")) ; => 40
9   (image-height (circle 20 "solid" "pink")) ; => 40
10  (image-height (circle 20 "solid" "gray")) ; => 40
11  (image-height (circle 20 "solid" "black")) ; => 40
12  (image-height (circle 20 "solid" "white")) ; => 40
13  (image-height (circle 20 "solid" "red")) ; => 40
14  (image-height (circle 20 "solid" "blue")) ; => 40
15  (image-height (circle 20 "solid" "green")) ; => 60
16  (image-height (circle 20 "solid" "yellow")) ; => 40
17  (image-height (circle 20 "solid" "cyan")) ; => 40
18  (image-height (circle 20 "solid" "magenta")) ; => 40
19  (image-height (circle 20 "solid" "brown")) ; => 40
20  (image-height (circle 20 "solid" "pink")) ; => 40
21  (image-height (circle 20 "solid" "gray")) ; => 40
22  (image-height (circle 20 "solid" "black")) ; => 40
23  (image-height (circle 20 "solid" "white")) ; => 40
24  (image-height (circle 20 "solid" "red")) ; => 40
25  (image-height (circle 20 "solid" "blue")) ; => 40
26  (image-height (circle 20 "solid" "green")) ; => 60
27  (image-height (circle 20 "solid" "yellow")) ; => 40
28  (image-height (circle 20 "solid" "cyan")) ; => 40
29  (image-height (circle 20 "solid" "magenta")) ; => 40
30  (image-height (circle 20 "solid" "brown")) ; => 40
31  (image-height (circle 20 "solid" "pink")) ; => 40
32  (image-height (circle 20 "solid" "gray")) ; => 40
33  (image-height (circle 20 "solid" "black")) ; => 40
34  (image-height (circle 20 "solid" "white")) ; => 40
35  (image-height (circle 20 "solid" "red")) ; => 40
36  (image-height (circle 20 "solid" "blue")) ; => 40
37  (image-height (circle 20 "solid" "green")) ; => 60
38  (image-height (circle 20 "solid" "yellow")) ; => 40
39  (image-height (circle 20 "solid" "cyan")) ; => 40
40  (image-height (circle 20 "solid" "magenta")) ; => 40
41  (image-height (circle 20 "solid" "brown")) ; => 40
42  (image-height (circle 20 "solid" "pink")) ; => 40
43  (image-height (circle 20 "solid" "gray")) ; => 40
44  (image-height (circle 20 "solid" "black")) ; => 40
45  (image-height (circle 20 "solid" "white")) ; => 40
46  (image-height (circle 20 "solid" "red")) ; => 40
47  (image-height (circle 20 "solid" "blue")) ; => 40
48  (image-height (circle 20 "solid" "green")) ; => 60
49  (image-height (circle 20 "solid" "yellow")) ; => 40
50  (image-height (circle 20 "solid" "cyan")) ; => 40
51  (image-height (circle 20 "solid" "magenta")) ; => 40
52  (image-height (circle 20 "solid" "brown")) ; => 40
53  (image-height (circle 20 "solid" "pink")) ; => 40
54  (image-height (circle 20 "solid" "gray")) ; => 40
55  (image-height (circle 20 "solid" "black")) ; => 40
56  (image-height (circle 20 "solid" "white")) ; => 40
57  (image-height (circle 20 "solid" "red")) ; => 40
58  (image-height (circle 20 "solid" "blue")) ; => 40
59  (image-height (circle 20 "solid" "green")) ; => 60
60  (image-height (circle 20 "solid" "yellow")) ; => 40
61  (image-height (circle 20 "solid" "cyan")) ; => 40
62  (image-height (circle 20 "solid" "magenta")) ; => 40
63  (image-height (circle 20 "solid" "brown")) ; => 40
64  (image-height (circle 20 "solid" "pink")) ; => 40
65  (image-height (circle 20 "solid" "gray")) ; => 40
66  (image-height (circle 20 "solid" "black")) ; => 40
67  (image-height (circle 20 "solid" "white")) ; => 40
68  (image-height (circle 20 "solid" "red")) ; => 40
69  (image-height (circle 20 "solid" "blue")) ; => 40
70  (image-height (circle 20 "solid" "green")) ; => 60
71  (image-height (circle 20 "solid" "yellow")) ; => 40
72  (image-height (circle 20 "solid" "cyan")) ; => 40
73  (image-height (circle 20 "solid" "magenta")) ; => 40
74  (image-height (circle 20 "solid" "brown")) ; => 40
75  (image-height (circle 20 "solid" "pink")) ; => 40
76  (image-height (circle 20 "solid" "gray")) ; => 40
77  (image-height (circle 20 "solid" "black")) ; => 40
78  (image-height (circle 20 "solid" "white")) ; => 40
79  (image-height (circle 20 "solid" "red")) ; => 40
80  (image-height (circle 20 "solid" "blue")) ; => 40
81  (image-height (circle 20 "solid" "green")) ; => 60
82  (image-height (circle 20 "solid" "yellow")) ; => 40
83  (image-height (circle 20 "solid" "cyan")) ; => 40
84  (image-height (circle 20 "solid" "magenta")) ; => 40
85  (image-height (circle 20 "solid" "brown")) ; => 40
86  (image-height (circle 20 "solid" "pink")) ; => 40
87  (image-height (circle 20 "solid" "gray")) ; => 40
88  (image-height (circle 20 "solid" "black")) ; => 40
89  (image-height (circle 20 "solid" "white")) ; => 40
90  (image-height (circle 20 "solid" "red")) ; => 40
91  (image-height (circle 20 "solid" "blue")) ; => 40
92  (image-height (circle 20 "solid" "green")) ; => 60
93  (image-height (circle 20 "solid" "yellow")) ; => 40
94  (image-height (circle 20 "solid" "cyan")) ; => 40
95  (image-height (circle 20 "solid" "magenta")) ; => 40
96  (image-height (circle 20 "solid" "brown")) ; => 40
97  (image-height (circle 20 "solid" "pink")) ; => 40
98  (image-height (circle 20 "solid" "gray")) ; => 40
99  (image-height (circle 20 "solid" "black")) ; => 40
100 (image-height (circle 20 "solid" "white")) ; => 40
```

### Folosiți cu încredere!

<http://docs.racket-lang.org/>

# Racket CheatSheet

## Laborator4

### let

**Colorată** - zona de vizibilitate pentru id1  
**Valoare de retur** - exprn

```
1 (let ((id1 val1)
2       (id2 val2)
3       ...
4       (idn valn))
5     expr1
6     expr2
7     ...
8     exprn)
9
10 (define a 10)
11
12 (let ((a 1) (b (+ a 1)))
13       (cons a b))                (1 . 11)
```

### letrec

**Colorată** - zona de vizibilitate pentru id2  
**Valoare de retur** - exprn

```
1 (letrec ((id1 val1)
2          (id2 val2)
3          ...
4          (idn valn))
5         expr1
6         expr2
7         ...
8         exprn)
9
10 ;; cand evaluez b, b trebuie sa fi fost definit
11 (letrec ((a b) (b 1))
12         (cons a b))              eroare
13
14 ;; corpul unei inchideri functionale
15 ;; nu se evalueaza la momentul definirii
16 (letrec
17   ((even-length?
18     (lambda (L)
19       (if (null? L)
20           #t
21           (odd-length? (cdr L))))))
22   (odd-length?
23     (lambda (L)
24       (if (null? L)
25           #f
26           (even-length? (cdr L))))))
27   (even-length? '(1 2 3 4 5 6))) #t
```

### Alte funcții

**sort remove assoc andmap findf splitf-at**

```
1 (sort '(5 2 1 6 4) >)                (6 5 4 2 1)
2 (remove 2 '(1 2 3 4 3 2 1))          (1 3 4 3 2 1)
3 (assoc 3 '((1 2) (3 4) (3 6) (4 5))) (3 4)
4 (andmap positive? '(1 2 3))          #t
5 (andmap number? '(1 b 3))            #f
6 (findf (lambda (x) (> x 4)) '(1 3 5 6 4)) 5
7 (findf (lambda (x) (> x 6)) '(1 3 5 6 4)) #f
8 (splitf-at '(1 3 4 5 6) odd?)        (1 3)
9                                       (4 5 6)
10 (splitf-at '(1 3 4 5 6) even?)       ()
11                                       (1 3 4 5 6)
```

### let\*

**Colorată** - zona de vizibilitate pentru id1  
**Valoare de retur** - exprn

```
1 (let* ((id1 val1)
2        (id2 val2)
3        ...
4        (idn valn))
5       expr1
6       expr2
7       ...
8       exprn)
9
10 (define a 10)
11
12 (let* ((a 1) (b (+ a 1)))
13       (cons a b))                (1 . 2)
```

### let-values

**Ca let, pentru expresii care întorc valori multiple**

```
1 ;; val-expr este o expresie care intoarce n valori
2 (let-values ( ((id1 id2 .. idn) val-expr)
3              ... )
4             corp)
5
6 (let-values (((x y) (quotient/remainder 10 3)))
7   (cons x y))                    (3 . 1)
```

### named let

**nume** - apare în **corp** ca un apel recursiv al funcției  
cu parametrii **id1 .. idn** și corpul **corp**

```
1 (let nome ((id1 val1)
2            (id2 val2)
3            ...
4            (idn valn))
5   corp)
6
7 (let loop ((n 5)
8           (fact 1))
9   (if (zero? n)
10       fact
11       (loop (sub1 n) (* n fact)))) 120
```

### Folositi cu incredere!

<http://docs.racket-lang.org/>

# Racket CheatSheet

Laborator 3

## Funcții anonime (lambda) și funcții cu nume

(lambda (arg1 arg2 ...) rezultat)  
(define nume val)

```
1 (lambda (x) x)           funcția identitate
2 ((lambda (x) x) 2)       2   aplicare funcție
3 (define idt (lambda (x) x)) legare la un nume
4 (define (idt x) x)       sintaxa alternativa
5 (idt 3)                  3
6
7 ((if true + -) (+ 1 2) 3) 6   if-ul se evaluează
8                             la o funcție
9
10 (define (comp f g)        funcția de
11   (lambda (x)             compunere (o)
12     (f (g x))))          a altor 2 funcții
13
14 ((comp car cdr) '(1 2 3)) 2   car o cdr
15
16 ((comp (lambda (x) (+ x 1)) 11 inc o dublare
17   (lambda (x) (* x 2)))
18 5)
```

## Funcții curried/ uncurried

```
1 (define add-uncurried    parametri luați
2   (lambda (x y)          simultan
3     (+ x y)))
4
5 (add-uncurried 1 2)      3
6
7 (define add-curried      parametri luați
8   (lambda (x)            succesiv
9     (lambda (y)
10      (+ x y))))
11
12 ((add-curried 1) 2)     3
13
14 (define inc (add-curried 1)) aplicație parțială
```

Perspective asupra funcțiilor binare *curried*:

- Iau parametrii „pe rând”, fiind aplicabile parțial.
- Iau un parametru și întorc o altă funcție de un parametru.

## Funcționala filter

(filter funcție listă)

Păstrează dintr-o listă elementele pentru care funcția NU întoarce false. (filter f (list e<sub>1</sub> ... e<sub>n</sub>)) → (list e<sub>i<sub>1</sub></sub> ... e<sub>i<sub>m</sub></sub>), cu (f e<sub>i<sub>k</sub></sub>) ≠ false.

```
1 (filter even? '(1 2 3))      '(2)
```

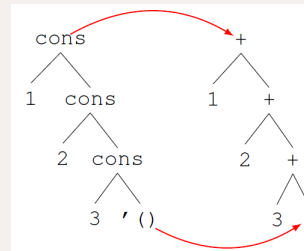
## Funcționalele foldl și foldr

(fold\* funcție acumulator listă)  
funcție → (lambda (element acumulator')  
acumulator')

Îmbină toate elementele unei liste pentru a construi o valoare finală, pornind de la un acumulator inițial. Într-un pas, funcția dată ca parametru combină elementul curent din listă cu acumulatorul, întorcând un nou acumulator. Acumulatorul final este întors ca rezultat al funcționalelor fold\*. Acesta poate fi chiar o listă.

- foldr (*right*) poate fi înțeleasă cel mai ușor prin faptul că funcția dată ca parametru se substituie lui cons, iar acumulatorul inițial, listei vide de la finalul listei. Prin urmare, elementele listei sunt prelucrate de la dreapta la stânga: (foldr f acc (list e<sub>1</sub> ... e<sub>n</sub>)) → (f e<sub>1</sub> (f ... (f e<sub>n</sub> acc)...))
- foldl (*left*) prelucrează elementele de la stânga la dreapta: (foldl f acc (list e<sub>1</sub> ... e<sub>n</sub>)) → (f e<sub>n</sub> (f ... (f e<sub>1</sub> acc)...))

Se pot furniza mai multe liste, caz în care comportamentul este similar cu cel al lui map pe mai multe liste.



```
1 (foldr + 0 '(1 2 3)) 6
2 (foldl + 0 '(1 2 3)) 6
3 (foldr cons '() '(1 2 3)) '(1 2 3) identitate!
4 (foldl cons '() '(1 2 3)) '(3 2 1) inversare
5 (foldl (lambda (x y acc)      21 2 liste
6   (+ x y acc))
7   0 '(1 2 3) '(4 5 6))
```

## Funcționala map

(map funcție listă)  
(map funcție lista1 lista2 ...)

Transformă independent elementele de pe poziții diferite ale uneia sau mai multor liste. Întoarce o listă cu același număr de elemente ca lista/ listele date ca parametru.

- Pentru o singură listă, aplică funcția, pe rând asupra fiecărui element: (map f (list e<sub>1</sub> ... e<sub>n</sub>)) → (list (f e<sub>1</sub>) ... (f e<sub>n</sub>))
- Pentru mai multe liste de aceeași lungime, funcția este aplicată la un moment dat asupra tuturor elementelor de pe aceeași poziție: (map f (list e<sub>11</sub> ... e<sub>1n</sub>) ... (list e<sub>m1</sub> ... e<sub>mn</sub>)) → (list (f e<sub>11</sub> ... e<sub>m1</sub>) ... (f e<sub>1n</sub> ... e<sub>mn</sub>))

Există și funcționalele andmap și ormap. Prima se asigură că, în urma aplicării lui map, toate rezultatele sunt diferite de false, iar a doua, că cel puțin un rezultat este diferit de false.

```
1 (map (lambda (x) (* x 10)) '(1 2 3)) '(10 20 30)
2 (map * '(1 2 3) '(10 20 30))          '(10 40 90)
3 (map list '(1 2 3))                    '((1) (2) (3))
4 (map list '(1 2) '(3 4))                '((1 3) (2 4))
5
6 (define (mult-by q)                      ; Curried
7   (lambda (x)
8     (* x q)))
9 (map (mult-by 5) '(1 2 3))              '(5 10 15)
```

## Funcționala apply

(apply funcție listă\_arg)  
(apply funcție arg\_1 ... arg\_n listă\_arg)

Aplică o funcție asupra parametrilor dați de elementele unei liste. Opțional, primii parametri ai funcției îi pot fi furnizați individual lui apply, înaintea listei cu restul parametrilor. (apply f x<sub>1</sub> ... x<sub>m</sub> (list e<sub>1</sub> ... e<sub>n</sub>)) → (f x<sub>1</sub> ... x<sub>m</sub> e<sub>1</sub> ... e<sub>n</sub>)

```
1 (apply + '(1 2 3))      6      suma
2 (apply + 1 '(2 3))      6      la fel
3 (apply list '(1 2 3))    '(1 2 3)
4 (apply list '(1 2 3) '(5 6 7)) '((1 2 3) 5 6 7)
```

# Racket CheatSheet

## Laborator5

### Promisiuni

#### delay force

```
1 (define p (delay (+ 1 2)))
2 p                                     #<promise:p>
3
4 ;; force forteaza evaluarea
5 (force p)                             3
6
7 ;; un force subsecvent ia rezultatul din cache
8 (force p)                             3
```

### Constructori fluxuri

#### empty-stream stream-cons

```
1 empty-stream                         #<stream>
2 (stream-cons 1 empty-stream)         #<stream>
3
4 (define ones (stream-cons 1 ones))    fluxul de 1
5
6 ;; fluxul numerelor naturale
7 (define naturals
8   (let loop ((n 0))
9     (stream-cons n (loop (add1 n)))))
```

### Operatori pe fluxuri

#### stream-first stream-rest stream-empty?

```
1 (stream-first naturals)               0
2 (stream-rest (stream-cons 2 ones))    fluxul de 1
3
4 (stream-empty? empty-stream)          #t
5 (stream-empty? ones)                  #f
```

### Funcționale pe fluxuri

#### stream-map stream-filter

```
1 ;; stream-map merge numai cu functii unare
2 (stream-map sqr naturals)              fluxul 0, 1, 4..
3
4 (stream-filter even? naturals)          fluxul nr pare
```

### Fluxuri definite explicit

#### Generator recursiv cu oricâți parametri definit în mod uzual cu named let

```
1 ;; fluxul puterilor lui 2
2 (define powers-of-2
3   (let loop ((n 1))
4     (stream-cons n (loop (* n 2)))))
5
6 ;; fluxul Fibonacci
7 (define fibonacci
8   (let loop ((n1 0) (n2 1))
9     (stream-cons n1 (loop n2 (+ n1 n2)))))
10
11 ;; fluxul 1/(n!)
12 ;; (cu care putem aproxima constanta lui Euler)
13 (define rev-factorials
14   (let loop ((term 1) (n 1))
15     (stream-cons term (loop (/ term n) (add1 n)))))
16
17 ;; testare: stream-take este definita de noi
18 ;; in laborator, nu exista in Racket
19
20 ;; rezultat '(1 2 4 8 16 32 64 128 256 512)
21 (stream-take powers-of-2 10)
22
23 ;; rezultat '(0 1 1 2 3 5 8 13 21 34)
24 (stream-take fibonacci 10)
25
26 ;; rezultat 2.7182815255731922
27 (apply + 0.0 (stream-take rev-factorials 10))
```

### AȘA DA / AȘA NU

#### Folosiți interfața Racket pentru fluxuri!

```
1 DA: (stream-cons x S) NU: (cons x (lambda () S))
2 NU: (cons x (delay S))
3 DA: (stream-rest S) NU: ((cdr S))
4 NU: (force (cdr S))
```

### Fluxuri definite implicit

#### Fără generator explicit

#### Dă explicit primii 1-2 termeni, apoi inițiază o prelucrare folosind (de obicei) funcționale pe fluxuri

```
1 ;; stream-zip-with este definita de voi
2 ;; in laborator, nu exista in Racket
3
4 ;; fluxul puterilor lui 2
5 (define powers-of-2-a
6   (stream-cons
7     1
8     (stream-zip-with +
9       powers-of-2-a
10      powers-of-2-a)))
11
12 (define powers-of-2-b
13   (stream-cons
14     1
15     (stream-map (lambda (x) (* x 2))
16       powers-of-2-b)))
17
18 ;; fluxul Fibonacci
19 (define fibonacci
20   (stream-cons
21     0
22     (stream-cons
23       1
24       (stream-zip-with +
25         fibonacci
26         (stream-rest fibonacci)))))
27
28 ;; fluxul 1/(n!)
29 (define rev-factorials
30   (stream-cons
31     1
32     (stream-zip-with /
33       rev-factorials
34       (stream-rest naturals))))
```

### Folositi cu incredere!

<http://docs.racket-lang.org/>

# Haskell CheatSheet

## Laborator 6

### Tipuri de bază

```
5      :: Int
'H'    :: Char
"Hello" :: String
True   :: Bool
False  :: Bool
```

### Determinarea tipului unei expresii

```
:t
> :t 42
42 :: Num a => a

a reprezintă o variabilă de tip, restrictionată la toate
tipurile numerice.

> :t 42.0
42 :: Fractional a => a
```

În acest exemplu, **a** este restrictionată la toate tipurile numerice fracționare (e.g. **Float**, **Double**).

### Constructorii liste

```
[]      (:) ..

[]      -- lista vida
(:)     -- operatorul de adaugare
-- la inceputul listei

1 : 3 : 5 : [] -- lista care contine 1, 3, 5
[1, 3, 5]     -- sintaxa echivalenta

[1:10]      -- lista care contine toate
-- numerele naturale de la 1 la 10
```

### Operatorii logici

```
not && ||

not True      False
not False     True
True || False True
True && False  False
```

### Operatorii pe liste

#### (++) head tail last init take drop

```
[1, 2] ++ [3, 4]      [1, 2, 3, 4]

head [1, 2, 3, 4]      1
tail [1, 2, 3, 4]      [2, 3, 4]

last [1, 2, 3, 4]      4
init [1, 2, 3, 4]      [1, 2, 3]

take 2 [1, 2, 3, 4]     [1, 2]
take 2 "HelloWorld"    "He"

drop 2 [1, 2, 3, 4]     [3, 4]

null []                True
null [1, 2, 3]         False
```

### Alte operații

#### length elem notElem reverse

```
length [1, 2, 3, 4]      4

elem 3 [1, 2, 3, 4]      True
elem 5 [1, 2, 3, 4]      False

notElem 3 [1, 2, 3, 4]   False
notElem 5 [1, 2, 3, 4]   True

reverse [1, 2, 3, 4]     [4, 3, 2, 1]
```

### Tupluri

Spre deosebire de liste, tuplurile au un număr fix de elemente, iar acestea pot avea tipuri diferite.

```
import Data.Tuple

("Hello", True) :: (String, Bool)
(1, 2, 3)        :: (Integer, Integer, Integer)

fst ("Hello", True)  "Hello"
snd ("Hello", True)  True
swap ("Hello", True) (True, "Hello")
```

### Funcții anonime (lambda)

#### \arg1 arg2 → corp

```
\x -> x      functia identitate
(\x y -> x + y) 1 2      3
let f = \x y -> x + y    legare la un nume
(f 1 2)          3
```

### Definire funcții

```
-- if .. then .. else
factorial x =
    if x < 1 then 1 else x * factorial (x - 1)

-- guards
factorial x
    | x < 1 = 1
    | otherwise = x * factorial (x - 1)

-- case .. of
factorial x = case x < 1 of
    True  -> 1
    _     -> x * factorial (x - 1)

-- pattern matching
factorial 0 = 1
factorial x = x * factorial (x - 1)
```

### Curry

În Haskell funcțiile sunt, by default, în forma curry.

```
:t (+)
(+) :: Num a => a -> a -> a

:t (+ 1)
(+ 1) :: Num a => a -> a
```

### Funcționale uzuale

#### map filter foldl foldr zip zipWith

```
map      :: (a -> b) -> [a] -> [b]
filter   :: (a -> Bool) -> [a] -> [a]
foldl    :: (a -> b -> a) -> a -> [b] -> a
zip       :: [a] -> [b] -> [(a, b)]
zipWith  :: (a -> b -> c) -> [a] -> [b] -> [c]

map (+ 2) [1, 2, 3]      [3, 4, 5]

filter odd [1, 2, 3, 4]  [1, 3]

foldl (+) 0 [1, 2, 3, 4] 10
foldl (-) 0 [1, 2]       -3    (0 - 1) - 2
foldr (-) 0 [1, 2]       -1    1 - (2 - 0)

zip [1, 2] [3, 4]        [(1, 3), (2, 4)]
zipWith (+) [1, 2] [3, 4] [4, 6]
```

## Sintaxa Let

```
let id1 = expr1
    id2 = expr2
    ...
    idn = expr3
in expr
```

Exemplu:

```
g = let x = y + 1
     y = 2
     (z, t) = (2, 5)
     f n = n * y
in (x + y, f 3, z + t)
```

Observație: Let este o **expresie**, o putem folosi în orice context în care putem folosi expresii.

Domeniul de vizibilitate al definițiilor locale este întreaga clauza let. (e.g. putem să li includem pe 'y' în definiția lui 'x', deși 'y' este definit ulterior. Cele două definiții nu sunt vizibile în afara clauzei let).

## Sintaxa Where

```
def = expr
where
  id1 = val1
  id2 = val2
  ...
  idn = valn
```

Exemple:

```
inRange :: Double -> Double -> String
inRange x max
  | f < low           = "Too_low!"
  | f >= low && f <= high = "In_range"
  | otherwise         = "Too_high!"
where
  f = x / max
  (low, high) = (0.5, 1.0)
```

```
-- with case
listType l = case l of
  [] -> msg "empty"
  [x] -> msg "singleton"
  _ -> msg "a_longer"
where
  msg ltype = ltype ++ "_list"
```

## Liste infinite

Putem exploata evaluarea leneșă a expresiilor în Haskell pentru a genera liste infinite. (un element nu este construit până când nu îl folosim efectiv).

Exemplu: definirea lazy a mulțimii tuturor numerelor naturale

```
naturals = iter 0
  where iter x = x : iter (x + 1)

-- Pentru a accesa elementele multimii putem
  folosi operatorii obisnuiti de la liste

> head naturals
0
> take 10 naturals
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

## Funcționale utile

**iterate, repeat, intersperse, zip, zipWith**

**iterate** generează o listă infinită prin aplicarea repetată a lui f: `iterate f x == [x, f x, f (f x), ...]`

Exemplu:

```
naturals = iterate (+ 1) 0
powsOfTwo = iterate (* 2) 1 -- [1, 2, 4, 8, ..]
```

```
repeat :: a -> [a]
> ones = repeat 1 -- [1, 1, 1, ..]
```

```
intersperse :: a -> [a] -> [a]
> intersperse ',' "abcde" -- "a,b,c,d,e"
```

```
zip :: [a] -> [b] -> [(a, b)]
zip naturals ["w", "o", "r", "d"]
-- [(0, "w"), (1, "o"), (2, "r"), (3, "d")]
```

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
```

```
evens = zipWith (+) naturals naturals
-- [2, 4, 6, ..]
```

```
fibonacci = 1 : 1 : zipWith (+) fibonacci (tail fibonacci)
-- sirul lui Fibonacci
```

```
concat :: [[a]] -> [a]
> concat ["Hello", "World", "!"]
"HelloWorld!"
```

## Operatorul '\$'

În anumite situații, putem omite parantezele folosind '\$'.

```
> length (tail (zip [1,2,3,4] ("abc" ++ "d")))
-- este echivalent cu
> length $ tail $ zip [1,2,3,4] $ "abc" ++ "d"
```

## Operatorul de compunere a funcțiilor '.'

$(f \circ g)(x)$  – echivalenta cu  $f(g(x))$

```
> let f = (+ 1) . (* 2)
> map f [1, 2, 3]
[3, 5, 7]
```

```
> length . tail . zip [1,2,3,4] $ "abc" ++ "d"
3
```

## List comprehensions

Cu ajutorul list comprehensions, putem genera liste pe baza altor liste, construite pe baza unor modele și a unor condiții.

```
[x | x <- [1..5]] -- [1, 2, 3, 4, 5]
```

```
[x + 2 | x <- [1..5]] -- [3, 4, 5, 6, 7]
```

```
[x | x <- [1..10], x `mod` 2 == 0] -- [2,4,6,8,10]
```

```
[(x, y) | x <- [1..4], y <- [10..12]]
-- aici se va construi o lista de perechi -
  [(1,10), (1,11), (1,12), (2,10), (2,11), (2,12),
   (3,10), (3,11), (3,12), (4,10), (4,11), (4,12)]
```

# Haskell CheatSheet

## Laborator 8

### type

```
1 -- type ne permite definirea unui sinonim de tip,
2 -- similar cu typedef din C
3 type Point = (Int, Int)
4
5 p :: Point
6 p = (2, 3)
```

### newtype

```
1 -- newtype este similar cu data, cu diferenta ca
2 -- ne permite crearea unui tip de date cu
3 -- un singur constructor, pe baza altor
4 -- tipuri de date existente
5 newtype Celsius = MakeCelsius Float deriving Show
6
7 newtype Fahrenheit = MakeFahrenheit Float
8 deriving Show
9
10 celsiusToFahrenheit :: Celsius -> Fahrenheit
11 celsiusToFahrenheit (MakeCelsius c) =
12     MakeFahrenheit $ c * 9/5 + 32
```

### data

```
1 -- data permite definirea de noi
2 -- tipuri de date algebrice
3 data PointT = PointC Double Double deriving Show
4
5 -- tipuri enumerate
6 data Colour = Red | Green | Blue | Black deriving
7 Show
8 nonColour :: Colour -> Bool
9 nonColour Black = True
10 nonColour _ = False
11
12 -- tipuri inregistrare
13 data PointT = PointC
14 { px :: Double
15   , py :: Double
16 } deriving Show
17 px (PointC x _) = x
18 py (PointC _ y) = y
19
20 --tipuri parametrizate
21 data Maybe a = Just a | Nothing deriving (Show,
22 Eq, Ord)
23 maybeHead :: [a] -> Maybe a
24 maybeHead (x : _) = Just x
25 maybeHead _ = Nothing
26
27 -- tipuri recursive
28 data List a = Void | Cons a (List a) deriving Show
29
30 data Natural = Zero | Succ Natural deriving Show
```



# Haskell CheatSheet

Laborator 8

## Polimorfism

1. **Parametric**: manifestarea aceluiași comportament pentru parametri de tipuri diferite.

Exemplu:

```
id :: a -> a
```

2. **Ad-hoc**: manifestarea unor comportamente diferite pentru parametri de tipuri diferite.

Exemplu:

```
elem :: (Eq a) => a -> [a] -> Bool
```

### Observație

O funcție poate conține ambele tipuri de polimorfism.

Exemplu:

```
lookup :: (Eq a) => a -> [(a,b)] -> Maybe
```

parametric pentru b și ad-hoc pentru a

## Clase

Clasele din Haskell seamănă mai mult cu conceptul de interfață din Java. O clasă reprezintă un set de funcții care definesc o interfață sau un comportament unitar pentru un tip de date.

```
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
```

### Observație

**Eq** definește 2 funcții, `(==)` și `(/=)`. Pentru a înrola un tip în clasa **Eq**, ambele funcții trebuie implementate.

## Instanțiere

Considerăm următorul tip:

```
data Point = Point { x :: Integer
                    , y :: Integer
                    }
```

Includem Point în clasa Eq astfel:

```
instance Eq Point where
    Point x1 y1 == Point x2 y2 =
        x1 == x2 && y1 == y2
    p1 /= p2 = not (p1 == p2)
```

Putem adăuga și tipuri de date generice într-o clasă.

```
data TrafficLight = Red | Yellow | Green
```

```
instance Eq TrafficLight where
    Red == Red = True
    Green == Green = True
    Yellow == Yellow = True
    _ == _ = False
```

## Deriving

**Ord, Enum, Bounded, Show, Read**

Având în vedere că, uneori, implementările pentru unele clase sunt relativ simple, compilatorul de Haskell poate face automat aceste implementări, dacă este folosit cuvântul cheie **deriving**.

```
data Point = Point Float Float
    deriving (Show)
data Person = Person
    { firstName :: String
    , lastName :: String
    , age :: Int
    , height :: Float
    , phoneNumber :: String
    , flavor :: String
    } deriving (Show)
data MyList a = Empty | a :-: (MyList a)
    deriving (Show, Read, Ord, Eq)
```

## Functor

Putem captura conceptul de containere „mapabile” într-o clasă folosind clasa **Functor**. Această clasă are o singură metodă, numită `fmap`, care este generalizarea funcției `map`.

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

```
instance Functor [] where
    fmap = map
```

```
instance Functor Maybe where
    fmap f (Just x) = Just (f x)
    fmap f Nothing = Nothing
```

## Extindere de clase

Haskell permite ca o clasă să extindă o altă clasă. Acest lucru este necesar când dorim ca un tip inclus într-o clasă să fie inclus doar dacă face deja parte dintr-o altă clasă.

```
class Located a where
    getLocation :: a -> (Int, Int)
class (Located a) => Movable a where
    setLocation :: (Int, Int) -> a -> a
```

```
data NamedPoint = NamedPoint
    { pointName :: String
    , pointX :: Int
    , pointY :: Int
    } deriving (Show)
```

```
instance Located NamedPoint where
    getLocation p = (pointX p, pointY p)
```

```
instance Movable NamedPoint where
    setLocation (x, y) p = p{ pointX = x
                             , pointY = y }
```

# Prolog CheatSheet

Laborator 9

## Fapte și Structuri

```
papagal(coco).
iubeste(mihai, maria).
iubeste(mihai, ana).
deplaseaza(scaun, camera1, camera2).
are(ion, carte(aventuri, 2002)).
merge(ion, facultate(upb)).
```

Faptele sunt **predicate de ordinul întâi** de aritate n, considerate adevărate. Structurile sunt înălțuiri de fapte.

Obținerea primei soluții este de obicei numită satisfacerea scopului iar obținerea altor soluții, resatisfacerea scopului.

Dacă în satisfacerea scopului s-au găsit alternative încă neexplorate pentru satisfacerea unor predicate, se poate cere resatisfacerea scopului tastând **;**. Dacă utilizatorul nu este interesat de resatisfacere poate apăsa tasta **.** sau **Enter**.

```
?- iubeste(mihai, X).
X = maria;    % mai există soluții, utilizatorul poate tasta ;
X = ana.      % nu mai există alte soluții, Prolog nu va mai cere
              input de la utilizator.
```

## Variabile

```
?- papagal(coco).
true.
?- papagal(CineEste).
CineEste = coco.
?- deplaseaza(_, DeUnde, Unde).
DeUnde = camera1, Unde = camera2
```

Numele argumentelor variabile începe cu literă mare iar numele constantelor simbolice începe cu literă mică.

O variabilă poate fi instanțiată (**legată**) dacă există un obiect asociat acestei variabile, sau neinstanțiată (**liberă**) dacă nu se știe încă ce obiect va desemna variabila.

Wildcard **'\_'** poate fi orice.

## Reguli

```
frumoasa(ana).           %1
bun(vlad).               %2
cunoaste(vlad, maria).   %3
cunoaste(vlad, ana).     %4
iubeste(mihai, maria).   %5
iubeste(X, Y):- bun(X),  %6
                    cunoaste(X, Y),
                    frumoasa(Y).
```

Regula 'iubeste(X,Y)' se poate traduce: "X iubeste pe Y dacă X e bun și dacă X cunoaste pe Y și dacă Y este frumoasă".

O **regulă** Prolog exprimă un fapt care depinde de alte **fapte**.

Faptele se pot înălțui folosind virgula(, - **și** logic) și punct și virgula(; - **sau** logic)

## Evaluare în Prolog

**Prolog este optimist:** atunci când Prolog primește o interogare, Prolog încearcă să demonstreze că interogarea este adevărată.

**Prolog este perseverent:** dacă interogarea conține variabile sau wild-cards, încercă să găsească valori pentru elementele neinstanțiate (în domeniul care pot fi desprinse din program) în așa fel încât să poată demonstra că interogarea este adevărată.

**Prolog face backtracking:** în timp ce încearcă demonstrarea unui scop, anumite predicate pot avea mai multe variante de demonstrare (e.g. mai multe reguli, sau folosesc operatori sau). Astfel se creează alternative, care vor fi explorate de Prolog prin backtracking pentru satisfacerea / resatisfacerea scopului.

## Operatori

Aritmetici: + - \* /  
Relationali: = \= < > =< >= := == \=  
Logici: \+ sau **not**

```
?- 1 + 2 := 2 + 1.
true.
?- 1 + 2 = 2 + 1.
false. % nu au aceeași formă (nu se evaluează)
?- 2 + 1 = 2 + 1.
true. % au aceeași formă (nu se evaluează)
?- X = 2 + 1.
X = 2+1.
?- X is 2 + 1.
X = 3.
?- X := 2 + 1.
ERROR: :=/2: Arguments are not sufficiently instantiated
```

- **\+** echivalent cu **not**, și întoarce true dacă scopul care urmează după nu poate fi satisfăcut în niciun fel (nu se poate demonstra că scopul este adevărat).

- **=** unifică partea din stânga cu partea din dreapta (dacă este posibil), realizând toate legările necesare. De exemplu: X-Y:Z = 5-[a, b, c]:y unifică astfel X = 5, Y = [a, b, c], Z = y (observați că nu realizează nicio evaluare).

- **\=** este opusul lui **=**, și este adevărat dacă partea din stânga nu unifică cu partea din dreapta. Ex: A \= B este echivalent cu (\+ (A = B))

- **==** verifică dacă partea din stânga și partea din dreapta sunt legate la aceeași valoare.

- **:=** evaluează **numeric** expresiile din stânga și dreapta și verifică dacă rezultatele au aceeași valoare numerică. Ambele părți trebuie să fie complet instanțiate.

- **=\=** evaluează **numeric** expresiile din stânga și dreapta și returnează true dacă valorile sunt diferite. Ambele părți trebuie să fie complet instanțiate.

- **is** evaluează numeric partea dreaptă (care trebuie să fie complet instanțiată, și:

- dacă partea stângă este instanțiată și este o valoare, verifică egalitatea valorilor.
- dacă partea stângă este o variabilă, leagă variabila la valoarea din dreapta.
- altfel, false.

## Liste

```
[]                - lista vida
[a,b,c]           - elementele a, b, c
[Prim|Rest]       - element concatenat la rest
[X1,X2,...,XN|Rest] - n elemente concatenate la restul listei
```

## Pattern matching

```
Haskell:
sum []      = 0           %1
sum (x:xs) = x + sum xs  %2
```

Prolog: (pentru predicatul sum(+L, -Sum)  
sum([], 0). % dacă primul argument este lista vidă, al doilea argument trebuie să fie / este legat la 0  
?- sum([H|T], S) :- sum(T, ST), S **is** ST + H. % dacă primul argument este o listă construită prin adăugarea unui element H la începutul unei liste T, și dacă suma listei T este ST, S este ST plus H.

În **Haskell** se realizează pattern matching într-o singură direcție: odata legate variabilele, se produce acțiunea descrisă în dreapta.

În **Prolog** pattern matching se efectuează în ambele direcții: folosirea variabilelor pentru a executa regulile și produce un raspuns **true** sau **false**, sau se pot căuta atât variabile care satisfac regulile scrise.

## Errors/Warnings

```
my_length1([],0).
my_length1([H|L],N) :- my_length1(L,N1), N1 is N - 1.
```

```
?- my_length([a, b, c])
Error: Arguments are not sufficiently instantiated
```

N nu are o valoare instanțiată pentru a putea scădea 1 și a da valoarea lui N1. N1 va primi valoarea 0 din apelul my\_length2([], 0), iar N trebuie calculat în funcție de N1 existent.

```
my_length2([],0).
my_length2([H|L],N) :- my_length2(L,N1), N is N1 + 1.
```

Warning: Singleton Variables: [H]

O variabilă singleton este o variabilă care apare o singură dată într-o regulă. Dacă variabila nu apare de mai multe ori în regulă, înseamnă că numele respectiv nu este folosit pentru a transmite o legare dintr-o parte în alta, deci este inutil.

```
Correct:
my_length3([],0).
my_length3([_|L],N) :- my_length3(L,N1), N is N1 + 1.
```

## Documentarea predicatelor și a argumentelor

```
predicat(nrArgumente
predicat(+Arg1, -Arg2, ?Arg3, ..., +ArgN)
```

Pentru a diferenția intrările (+) de ieșiri (-), se prefixează argumentele cu indicatori. Acele argumente care pot fi fie intrări, fie ieșiri se prefixează cu (?)

Unele predicate nu au parametrii

Ex: Predicatul **fail/0** care se va evalua mereu la **false**.

# Prolog CheatSheet

Laborator 11

## Liste și funcții utile pe liste

```
[ ] - lista vida
[a,b,c] - elementele a, b, c
[Prim|Rest] - element concatenat la rest
[X1,X2,...,XN|Rest] - n elemente concatenate la restul listei

append(?List1, ?List2, ?List1AndList2) % List1AndList2 este
concatenarea între List1 și List2
?- append([1,2,3], [4,5], X).
X = [1, 2, 3, 4, 5].
?- append([1,2,3], X, [1,2,3,4]).
X = [4].
?- append([1,2,3], X, [2,3,4]).
false.

member(?Elem, ?List) % Adevărat dacă Elem esre prezent în List
?- member(1, [1,2,3]).
true ;
false.
?- member(X, [1,2,3]).
X = 1 ;
X = 2 ;
X = 3.

length(?List, ?Int) % Adevărat dacă Int reprezintă numărul de
elemente din List
?- length([1,2,3], X).
X = 3.

reverse(?List1, ?List2) % Adevărat atunci când elementele din
List2 sunt în ordine inversă față de List1
?- reverse([1, 2, 3], X).
X = [3, 2, 1].

sort(+List, -Sorted) % Adevărat atunci când Sorted conține toate
elementele din List (fără duplicate), sortate în ordinea
standard
?- sort([2, 3, 1, 2], X).
X = [1, 2, 3].
```

## Backtracking când se cunoaște lungimea căii către soluție

Regulă: Atunci când calea către soluție respectă un anumit template (avem de instanțiat un număr finit, predeterminat, de variabile), este eficient să definim un astfel de template în program.

De exemplu, pentru problema celor opt regine putem scrie astfel:

```
template([1/_ , 2/_ , 3/_ , 4/_ , 5/_ , 6/_ , 7/_ , 8/_]).
```

```
correct([ ]):-!.
correct([X/Y|Others]):-
    correct(Others),
    member(Y, [1, 2, 3, 4, 5, 6, 7, 8]),
    safe(X/Y, Others). % predicat care verifică faptul că
                        reginele nu se atacă între ele
```

```
solve_queens(S):-template(S), correct(S).
```

## Backtracking când calea are un număr nedeterminat de stări

În această situație nu este posibil să definim un template care descrie forma soluției problemei. Vom defini o căutare mai generală, după modelul următor:

```
solve(Solution):-
    initial_state(State),
    search([State], Solution).
```

search(+StăriVizitate,-Soluție) definește mecanismul general de căutare astfel:

- căutarea începe de la o stare inițială dată (predicatul initial\_state/1)
- dintr-o stare curentă se generează stările următoare posibile (predicatul next\_state/2)
- se testează că starea în care s-a trecut este nevizitată anterior (evitând astfel traseele ciclice)
- căutarea continuă din noua stare, până se întâlnește o stare finală (predicatul final\_state/1)

```
search([CurrentState|Other], Solution):-
    final_state(CurrentState), !,
    reverse([CurrentState|Other], Solution).
```

```
search([CurrentState|Other], Solution):-
    next_state(CurrentState, NextState),
    \+ member(NextState, Other),
    search([NextState,CurrentState|Other], Solution).
```

## Mecanism BFS

bfs(+CoadăStărilorNevizitate,+StăriVizitate,-Soluție) va defini mecanismul general de căutare în lățime, astfel:

- căutarea începe de la o stare inițială dată care n-are predecesor în spațiul stărilor (StartNode cu părintele nil)
- se generează toate stările următoare posibile
- se adaugă toate aceste stări la coada de stări încă nevizitate
- căutarea continuă din starea aflată la începutul cozii, până se întâlnește o stare finală

```
bfs([(StartNode,nil)], [], Discovered).
```

## Mecanism A\*

astar(+End, +Frontier, +Discovered, +Grid, -Result) va defini mecanismul general de căutare A\*, astfel:

- căutarea începe de la o stare inițială dată care n-are predecesor în spațiul stărilor (StartNode cu părintele nil) și distanța estimată de la acesta până la nodul de final printr-o euristică (exemplu: distanța Manhattan)
- se generează toate stările următoare posibile și se calculează costul acestora adăugând costul acțiunii din părinte până în starea aleasă cu costul real calculat pentru a ajunge în părinte (costul părintelui în Discovered)
- dacă starea aleasă nu este în Discovered sau dacă noul cost calculat al acesteia este mai mic decât cel din Discovered, se adaugă în acesta, apoi va fi introdusă în coada de priorități (Frontier) cu prioritatea fiind costul cu care a fost adăugată în Discovered + valoarea dată de euristică din starea curentă până în cea finală
- căutarea continuă din starea aflată la începutul cozii, până se întâlnește o stare finală

```
astar_search(Start, End, Grid, Path) :-
    manhattan(Start, End, H),
    astar(End, [H:Start], [Start:(\"None\", 0)], Grid, Discovered),
    get_path(Start, End, Discovered, [End], Path).
```

# Prolog CheatSheet

Laborator 12

## Aflarea tuturor soluțiilor pentru satisfacerea unui scop

### findall/3

```
findall(+Template, +Goal, -Bag)
```

Predicatul findall creează o listă de instanțieri ale lui Template care satisfac Goal și apoi unifică rezultatul cu Bag

```
higherThan(Numbers, Element, Result):-
    findall(X, (member(X, Numbers), X > Element), Result).
?- higherThan([1, 2, 7, 9, 11], 5, X).
X = [7, 9, 11]
```

```
?- findall([X, SqX], (member(X, [1,2,7,9,15]), X > 5, SqX is X **
    2), Result). % în argumentul Template putem construi
    structuri mai complexe
Result = [[7, 49], [9, 81], [15, 225]].
```

## Aflarea tuturor soluțiilor pentru satisfacerea unui scop

### bagof/3

```
bagof(+Template, +Goal, -Bag)
```

Predicatul bagof este asemănător cu predicatul findall, cu excepția faptului că predicatul bagof construiește câte o listă separată pentru fiecare instanțiere diferită a variabilelor din Goal (fie că ele sunt numite sau sunt înlocuite cu underscore.

```
are(andrei,laptop,1). are(andrei,pix,5). are(andrei,ghiozdan,2).
are(radu,papagal,1). are(radu,ghiozdan,1). are(radu,laptop,2).
are(ana, telefon, 3). are(ana, masina, 1).
```

```
?- findall(X, are(_, X, _), Bag).
Bag = [laptop, pix, ghiozdan, papagal, ghiozdan, laptop, telefon,
    masina]. % laptop și ghiozdan apar de două ori pentru că
    sunt două posibile legări pentru persoană și pentru cantitate
```

```
?- bagof(X, are(andrei, X, _), Bag).
Bag = [laptop] ;
Bag = [ghiozdan] ;
Bag = [pix].
% bagof creează câte o soluție pentru fiecare posibilă legare
    pentru cantitate. Putem aici folosi operatorul existențial ^
?- bagof(X, C^are(andrei, X, C), Bag).
Bag = [laptop, pix, ghiozdan]. % am cerut lui bagof să pună toate
    soluțiile indiferent de legarea lui C în același grup
```

```
?- bagof(X, C^are(P, X, C), Bag).
P = ana, Bag = [telefon, masina] ;
P = andrei, Bag = [laptop, pix, ghiozdan] ;
P = radu, Bag = [papagal, ghiozdan, laptop].
```

Dacă aplicăm operatorul existențial pe toate variabilele libere din scop, rezultatul este identic cu cel al lui findall.

```
?- bagof(X, X^P^C^are(P, X, C), Bag).
Bag = [laptop, pix, ghiozdan, papagal, ghiozdan, laptop, telefon,
    masina].
```

## Aflarea tuturor soluțiilor pentru satisfacerea unui scop

### setof/3

```
setof(+Template, +Goal, -Bag)
```

Predicatul setof este asemănător cu bagof, dar sortează rezultatul (și elimină duplicatele) folosind sort/2.

```
?- setof(X, C^are(P, X, C), Bag).
P = ana, Bag = [masina, telefon] ; %se observă sortarea
P = andrei, Bag = [ghiozdan, laptop, pix] ;
P = radu, Bag = [ghiozdan, laptop, papagal].
```

```
?- setof(X, P^C^are(P, X, C), Bag).% setof elimină duplicatele
Bag = [ghiozdan, laptop, masina, papagal, pix, telefon].
```

## Aflarea tuturor soluțiilor pentru satisfacerea unui scop

### forall/2

```
forall(+Cond, +Action)
```

Predicatul forall verifică dacă pentru orice legare din Cond, care reprezintă un domeniu ce conține legări de variabile, se pot îndeplini condițiile din Action.

```
?- forall(member(X, [2, 4, 6]), X mod 2 == 0).
true.
```

```
?- forall(member(X, [2, 4, 3, 6]), X mod 2 == 0).
false.
```

```
?- forall(member(X, [6, 12, 18]), (X mod 2 == 0, X mod 3 == 0))
    .
true.
```