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].
```

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].
```

Backtracking când se cunoaște lungimea căii către soluție

Regulă: Atunci când calea către soluţie respectă un anumit template (avem de instanţiat 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 situaţie nu este posibil să definim un template care descrie forma soluţiei problemei. Vom defini o căutare mai generală, după modelul următor:

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

search(+StăriVizitate,-Soluţie) 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 întâlneşte 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(+CoadaStărilorNevizitate,+StăriVizitate,-Soluţie) va defini mecanismul general de căutare în lăţime, astfel:

- căutarea începe de la o stare inițială dată care n-are predecesor în spaţiul 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 întâlneşte 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 spaţiul 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 adaugată în Discovered + valoarea dată de euristică din starea curentă până în cea finală
- căutarea continuă din starea aflată la începutul cozii, până se întâlneşte 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).
```

Haskell CheatSheet

Laborator 9

Polimorfism

1. **Parametric**: manifestarea aceluiaşi 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
   ```

Observaţie

O funcţie poate conţine ambele tipuri de polimorfism.
Exemplu:

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

parametric pentru `b` şi ad-hoc pentru `a`

Clase

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

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

Observaţie

Eq defineşte 2 funcţii, `(==)` şi `(/=)`. Pentru a înrola un tip în clasa **Eq**, ambele funcţii trebuie implementate.

Instanţiere

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 şi 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 funcţionalei `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}
```

Racket CheatSheet

Laborator 3

Funcții anonime (lambda) și funcții cu nume

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

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

((if true + -) (+ 1 2) 3)       6   if-ul se evaluează
                                    la o funcție

(define (comp f g)                  funcția de
  (lambda (x)                       compunere (o)
    (f (g x))))                     a altor 2 funcții

((comp car cdr) '(1 2 3))       2   car o cdr

((comp (lambda (x) (+ x 1)) 11 inc o dublare
       (lambda (x) (* x 2)))
 5)
```

Funcții *curried*/ *uncurried*

```
(define add-uncurried               parametri luați
  (lambda (x y)                     simultan
    (+ x y)))

(add-uncurried 1 2)             3

(define add-curried                 parametri luați
  (lambda (x)                       succesiv
    (lambda (y)
      (+ x y))))

((add-curried 1) 2)             3

(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_1 ... e_n`)`) → (`list` e_{i_1} ... e_{i_m}), cu (`f` e_{i_k}) ↛ `false`.

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

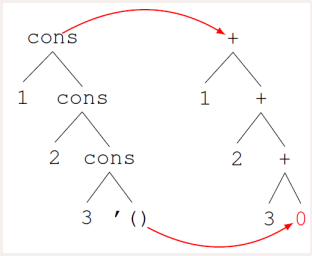
Funcționalele `foldl` și `foldr`

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

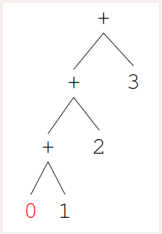
Îmbină toate elementele unei liste pentru a construi o valoare finală, pornind de la un acumulator inițial. Într-un pas, funcția dată ca parametru combină elementul curent din listă cu acumulatorul, întorcând un nou acumulator. Acumulatorul final este întors ca rezultat al funcționalelor `fold*`. Acesta 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_1 ... e_n`)`) → (`f` e_1 (`f` ... (`f` e_n `acc`)...))
- `foldl` (*left*) prelucrează elementele de la stânga la dreapta: (`foldl f acc (list` e_1 ... e_n`)`) → (`f` e_n (`f` ... (`f` e_1 `acc`)...))

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




```
(foldr + 0 '(1 2 3))  6
```




```
(foldl + 0 '(1 2 3))  6
(foldr cons '() '(1 2 3))    '(1 2 3)   identitate!
(foldl cons '() '(1 2 3))    '(3 2 1)   inversare
(foldl (lambda (x y acc)              21   2 liste
         (+ x y acc))
       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_1 ... e_n`)`) → (`list (f` e_1`)` ... (`f` e_n`)`)
- 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_{11} ... e_{1n}`)` ... (`list` e_{m1} ... e_{mn}`)`) → (`list (f` e_{11} ... e_{m1}`)` ... (`f` e_{1n} ... e_{mn}`)`)

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`.

```
(map (lambda (x) (* x 10)) '(1 2 3))    '(10 20 30)
(map * '(1 2 3) '(10 20 30))            '(10 40 90)
(map list '(1 2 3))                   '((1) (2) (3))
(map list '(1 2) '(3 4))              '((1 3) (2 4))

(define (mult-by q)                       ; Curried
  (lambda (x)
    (* x q)))
(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_1 ... x_m (`list` e_1 ... e_n`)`) → (`f` x_1 ... x_m e_1 ... e_n)

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

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
```

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

Constructori liste

[] (:)

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

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

Operatori logici

not && ||

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

Operatori 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 reverse

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

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

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 functii

```
-- 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

In Haskell funcțiile sunt, by default, in 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]
```

Haskell CheatSheet

Laborator 7

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 sa li includem pe 'y' în definiția lui 'x', deși 'y' este definit ulterior. Cele doua 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"
```

Structuri de date infinite

Putem exploata evaluarea leneșă a expresiilor în Haskell pentru a genera liste sau alte structuri de date 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, ..]

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

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

System.Random

mkStdGen primește un număr natural (seed) și întoarce un generator de numere aleatoare.
next primește un generator și întoarce un tuplu: (următorul număr generat, noua stare a generatorului)

```
mkStdGen :: Int -> StdGen
> mkStdGen 42
43 1
-- starea unui generator este reprezentata intern
de doua valori intregi (in acest caz, 43 si 1)

next :: g -> (Int, g)
> next (mkStdGen 42)
(1679910, 1720602 40692)
> next (snd (next (mkStdGen 42)))
(620339110,128694412 1655838864)
```

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"
3
```

Operatorul de compunere a funcțiilor '.'

(f . 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
```

map filter fold

map filter foldl foldr

```
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)
```

Racket CheatSheet

Laborator1

Sintaxa Racket

(nume_functie arg1 arg2 ...)

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

Operatori relationali

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

```
(< 3 2)                    #f
(>= 3 2)                   #t
(= 1 1)                    #t          (numere)
(= '(1 2) '(1 2))          eroare
(equal? '(1 2) '(1 2))     #t          (valori)
(eq? '(1 2 3) '(1 2 3))    #f          (referinte)

(define x '(1 2 3))
(eq? x x)                  #t

(zero? 0)                  #t (true)
(zero? 1)                  #f (false)
```

Operatori logici

not and or

```
(not true)                 #f
(not false)                #t
(or true false)            #t
(and #f #t)                #f
```

Constructori liste

null '() cons list

```
'()                                ()
null                               ()

(cons 1 null)                      (1)
(cons 'a '(b c))                   (a b c)

(list 1)                           (1)
(list 1 2 3 4)                     (1 2 3 4)
```

Operatori pe liste

car cdr first rest null? length member reverse append

```
(car '(1 2 3 4))                   1
(first '(1 2 3 4))                 1
(cdr '(1 2 3 4))                   (2 3 4)
(rest '(1 2 3 4))                  (2 3 4)
(cadr '(1 2 3 4 5))                2
(cdar '(1 2 3 4 5))                eroare
(cddr '(1 2 3 4 5))                (3 4 5)
(caddr '(1 2 3 4 5))               3

(null? null)                       #t (true)
(null? '(1 2))                     #f (false)

(length '(1 2 3 4))                4
(length '(1 (2 3) 4))              3

(member 'a '(b c a d a e))         '(a d a e)
(member 'f '(b c a d a e))         #f

(reverse '(1 (2 3) 4))             (4 (2 3) 1)

(list? '())                        #t
(list? 2)                          #f

(append '(1 2 3) '(4) '(5))        (1 2 3 4 5)
(append 1 '(2 3 4))                eroare
```

take și drop

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

(take-right '(1 2 3 4) 2)          '(3 4)
(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)

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

Sintaxa if

```
(if test exp1 exp2)

(if (< a 0)
    a
    (if (> a 10) (* a a) 0))
```

Sintaxa cond

```
(cond  (test1 exp1) (test2 exp2) ... (else exp))

(cond
   ((< a 0) a)
   ((> a 10) (* a a))
   (else 0))
```

AȘA DA / AȘA NU

```
DA: (cons x L)          NU: (append (list x) L)
                        NU: (append (cons x '()) L)
DA: (if c vt vf)        NU: (if (equal? c #t) vt vf)
DA: (null? L)           NU: (= (length L) 0)
DA: (zero? x)           NU: (equal? x 0)
DA: test                NU: (if test #t #f)
DA: (or ceva1 ceva2)    NU: (if ceva1 #t ceva2)
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)

Folositi cu incredere!

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

Prolog CheatSheet

Laborator 10

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 înlănțuiri 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 'iubește(X,Y)' se poate traduce: "X iubește pe Y dacă X e bun și dacă X cunoaște pe Y și dacă Y este frumoasă".
O **regulă** Prolog exprimă un fapt care depinde de alte **fapte**.
Faptele se pot înlănțui 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 wildcards, încercă să găsească valori pentru elementele neinstanțiate (în domenii 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 intr-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**.

Racket CheatSheet

Laborator4

let

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

```
(let ((id1 val1)
      (id2 val2)
      ...
      (idn valn))
   expr1
   expr2
   ...
   exprn)

(define a 10)

(let ((a 1) (b (+ a 1)))
   (cons a b))                              (1 . 11)
```

let*

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

```
(let* ((id1 val1)
       (id2 val2)
       ...
       (idn valn))
   expr1
   expr2
   ...
   exprn)

(define a 10)

(let* ((a 1) (b (+ a 1)))
   (cons a b))                              (1 . 2)
```

named let

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

```
(let nume ((id1 val1)
           (id2 val2)
           ...
           (idn valn))
   corp)

(let loop ((n 5)
           (fact 1))
   (if (zero? n)
       fact
       (loop (sub1 n) (* n fact))))          120
```

letrec

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

```
(letrec ((id1 val1)
         (id2 val2)
         ...
         (idn valn))
   expr1
   expr2
   ...
   exprn)

;; cand evaluez b, b trebuie sa fi fost definit
(letrec ((a b) (b 1))
   (cons a b))                                eroare

;; corpul unei inchideri functionale
;; nu se evalueaza la momentul definirii
(letrec
     ((even-length?
       (lambda (L)
         (if (null? L)
             #t
             (odd-length? (cdr L)))))
      (odd-length?
       (lambda (L)
         (if (null? L)
             #f
             (even-length? (cdr L))))))
   (even-length? '(1 2 3 4 5 6)))               #t
```

let-values

Ca let, pentru expresii care întorc valori multiple

```
;; val-expr este o expresie care intoarce n valori
(let-values ( ((id1 id2 .. idn) val-expr)
              ... )
   corp)

(let-values (((x y) (quotient/remainder 10 3)))
   (cons x y))                                (3 . 1)
```

Alte funcții

sort remove assoc andmap findf splitf-at

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

Folositi cu incredere!

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

Racket CheatSheet

Laborator 2

Recursivitate pe stivă

```
; suma elementelor unei liste
(define (sum-list L)


; aici nu avem nevoie de funcție auxiliară


  (if (null? L)
      0    ; la sfârșit creăm valoarea inițială
      (+ (car L) (sum-list (cdr L)))
     ; ^ construim rezultatul pe revenire
     ;  (după întoarcerea din recursivitate)
      ))
; fiecare apel recursiv întoarce rezultatul
    corespunzător argumentelor



; concatenarea a două liste
(define (app L1 L2)



  (if (null? L1)
      L2  ; când L1 este vidă, întoarcem L2
      (cons (car L1) (app (cdr L1) L2))
      ; ^ construim rezultatul pe revenire))
```

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

Recursivitate pe coadă

```
; suma elementelor unei liste
(define (sum-list L)
  (sum-list-tail 0 L))  ; <-- funcție ajutătoare
                ; ^ valoarea inițială pentru sumă

                ; în sum construim rezultatul
(define (sum-list-tail sum L)
  (if (null? L)
      sum        ; la sfârșit avem rezultatul gata
      (sum-list-tail
        (+ sum (car L))
       ; ^ construim rezultatul pe avans
       ;  (pe măsură ce intrăm în recursivitate)
        (cdr L))))
  ; funcția întoarce direct rezultatul apelului
      recursiv -- toate apelurile recursive întorc
      același rezultat, pe cel final


; concatenarea a două liste
(define (app A B)
  (app-iter B (reverse A)))
  ; nevoie de funcție ajutătoare
  ; rezultatul este construit în ordine inversă

(define (app-iter B Result)
  (if (null? B) ; la sfârșit rezultatul e complet
      (reverse Result) ; inversăm rezultatul
      (app-iter (cdr B) (cons (car B) Result))))
        ; 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 ...)

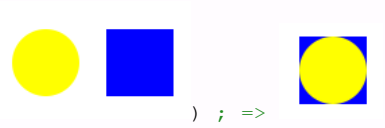
```
(max 2 3)                3
(+ 2 3)                  5
```

AȘA DA / AȘA NU

```
DA: (cons x L)           NU: (append (list x) L)
                         NU: (append (cons x '()) L)
DA: (if c vt vf)         NU: (if (equal? c #t) vt vf)
DA: (null? L)            NU: (= (length L) 0)
DA: (zero? x)            NU: (equal? x 0)
DA: test                 NU: (if test #t #f)
DA: (or ceva1 ceva2)     NU: (if ceva1 #t ceva2)
DA: (and ceva1 ceva2)    NU: (if ceva1 ceva2 #f)
```

Imagini în Racket

image-height, overlay, flip-vertical



```
(overlay                ) ; =>
```

```
(flip-vertical          ) ; =>
(image-height (circle 20 "solid" "red")) ; => 40
```

```
(image-height           ) ; => 60
```

Folosiți cu încredere!

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

Racket CheatSheet

Laborator5

Promisiuni

delay force

```
(define p (delay (+ 1 2)))
p                                       #<promise:p>

;; force forteaza evaluarea
(force p)                                       3

;; un force subsecvent ia rezultatul din cache
(force p)                                       3
```

Constructori fluxuri

empty-stream stream-cons

```
empty-stream                            #<stream>
(stream-cons 1 empty-stream)            #<stream>

(define ones (stream-cons 1 ones))    fluxul de 1

;; fluxul numerelor naturale
(define naturals
  (let loop ((n 0))
    (stream-cons n (loop (add1 n)))))
```

Operatori pe fluxuri

stream-first stream-rest stream-empty?

```
(stream-first naturals)                         0
(stream-rest (stream-cons 2 ones))    fluxul de 1

(stream-empty? empty-stream)                   #t
(stream-empty? ones)                           #f
```

Funcționale pe fluxuri

stream-map stream-filter

```
;; stream-map merge numai cu functii unare
(stream-map sqr naturals)        fluxul 0, 1, 4..

(stream-filter even? naturals)    fluxul nr pare
```

Fluxuri definite explicit

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

```
;; fluxul puterilor lui 2
(define powers-of-2
  (let loop ((n 1))
    (stream-cons n (loop (* n 2)))))

;; fluxul Fibonacci
(define fibonacci
  (let loop ((n1 0) (n2 1))
    (stream-cons n1 (loop n2 (+ n1 n2)))))

;; fluxul 1/(n!)
;; (cu care putem aproxima constanta lui Euler)
(define rev-factorials
  (let loop ((term 1) (n 1))
    (stream-cons term (loop (/ term n) (add1
        n)))))

;; testare: stream-take este definita de noi
;; in laborator, nu exista in Racket

;; rezultat '(1 2 4 8 16 32 64 128 256 512)
(stream-take powers-of-2 10)

;; rezultat '(0 1 1 2 3 5 8 13 21 34)
(stream-take fibonacci 10)

;; rezultat 2.7182815255731922
(apply + 0.0 (stream-take rev-factorials 10))
```

AȘA DA / AȘA NU

Folosiți interfața Racket pentru fluxuri!

```
DA: (stream-cons x S) NU: (cons x (lambda () S))
                      NU: (cons x (delay S))
DA: (stream-rest S)   NU: ((cdr S))
                      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

```
;; stream-zip-with este definita de voi
;; in laborator, nu exista in Racket

;; fluxul puterilor lui 2
(define powers-of-2-a
  (stream-cons
   1
   (stream-zip-with +
                    powers-of-2-a
                    powers-of-2-a)))

(define powers-of-2-b
  (stream-cons
   1
   (stream-map (lambda (x) (* x 2))
               powers-of-2-b)))

;; fluxul Fibonacci
(define fibonacci
  (stream-cons
   0
   (stream-cons
    1
    (stream-zip-with +
                     fibonacci
                     (stream-rest fibonacci)))))

;; fluxul 1/(n!)
(define rev-factorials
  (stream-cons
   1
   (stream-zip-with /
                    rev-factorials
                    (stream-rest naturals))))
```

Folositi cu incredere!

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

Haskell CheatSheet

Laborator 8

type

```haskell
-- type ne permite definirea unui sinonim de tip,
-- similar cu typedef din C
type Point = (Int, Int)

p :: Point
p = (2, 3)
```

newtype

```haskell
-- newtype este similar cu data, cu diferenta ca
-- ne permite crearea unui tip de date cu
-- un singur constructor, pe baza altor
-- tipuri de date existente
newtype Celsius = MakeCelsius Float deriving Show

newtype Fahrenheit = MakeFahrenheit Float
    deriving Show

celsiusToFahrenheit :: Celsius -> Fahrenheit
celsiusToFahrenheit (MakeCelsius c) =
    MakeFahrenheit $ c * 9/5 + 32
```

data

```haskell
-- data permite definirea de noi
-- tipuri de date algebrice
data PointT = PointC Double Double deriving Show

-- tipuri enumerate
data Colour = Red | Green | Blue | Black deriving
    Show
nonColour :: Colour -> Bool
nonColour Black = True
nonColour _     = False

-- tipuri inregistrare
data PointT = PointC
  { px :: Double
  , py :: Double
  } deriving Show
px (PointC x _) = x
py (PointC _ y) = y

--tipuri parametrizate
data Maybe a = Just a | Nothing deriving (Show,
    Eq, Ord)
maybeHead :: [a] -> Maybe a
maybeHead (x : _) = Just x
maybeHead _       = Nothing

-- tipuri recursive
data List a = Void | Cons a (List a) deriving Show

data Natural = Zero | Succ Natural deriving Show
```