

# 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
```

### 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
```

### Operatorii pe liste

`(++) head tail last init take drop takeWhile dropWhile`

```
[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

takeWhile (<5) [1..8]   [1, 2, 3, 4]
dropWhile (<5) [1..8]   [5, 6, 7, 8]
```

### 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")
```

### 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 formă 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    :: (b -> a -> b) -> b -> [a] -> b
foldr    :: (a -> b -> b) -> b -> [a] -> b
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)
foldl (flip (:)) [] [1, 2, 3] [3, 2, 1]
foldr (:) [] [1, 2, 3]    [1, 2, 3]
```

```
zip [1, 2] [3, 4]        [(1, 3), (2, 4)]
zipWith (+) [1, 2] [3, 4] [4, 6]
zip = zipWith (,)
-- zip nu este propriu-zis o funcțională, dar se
-- poate obține pe baza funcționalei zipWith
```

## 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 și alte funcții utile

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

**iterate** generează o listă infinită prin aplicarea repetată a lui f:  $\text{iterate } f \ x == [x, f \ x, f \ (f \ x), \dots]$

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!"
```

## 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)]
```

```
[(x, y) | x <- [1..8], y <- [10..16], mod x 2 == 0,
  mod y 3 == 0]
-- [(2,12), (2,15), (4,12), (4,15), (6,12), (6,15),
   (8,12), (8,15)]
```

```
take 10 [x | x <- [1..], mod x 2 == 0]
-- flux de numere pare -
  [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

## 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 '.'

$(f \cdot g)(x)$  – echivalența 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
```