

Documentatie laborator 6 – Introducere in Haskell

Haskell

Aceasta documentatie este o varianta (mult) prescurtata a documentatiei de la <http://www.haskell.org/tutorial/>. Haskell este un limbaj:

- pur functional
- tare tipat
- cu functii nestricte
- cu evaluare lenesa
- care permite definirea functiilor folosind pattern matching

Pentru a lucra in Haskell downloadati Hugs de la <http://cvs.haskell.org/Hugs/pages/downloading.htm> sau GHC de la http://www.haskell.org/ghc/download_ghc_7_4_1.

Tipuri de date

Haskell face singur inferenta de tip, insa este posibila si declararea tipurilor de catre utilizator. Cateva exemple de valori impreuna cu tipul asociat:

`5 :: Integer`

`'a' :: Char`

`inc :: Int -> Int`

`[1,2,3] :: [Int]`

`('b',4) :: (Char,Int) -- tip tuplu`

Semnul `::` poate fi citit ca "are tipul". Notatia `a -> b` refera o functie care primeste un argument de tip `a` si intoarce un rezultat de tip `b`. Alte exemple de tipuri predefinite: `Bool`, `String` etc.

De asemenea, Haskell permite tipuri definite de catre utilizator. De exemplu:

```
type Pair = (Int, Int)
```

```
addTwo :: Pair -> Int
```

```
addTwo (first,second) = first+second
```

Sintaxa

- un script e o serie de definitii de functii
- ele sunt separate prin asezarea in pagina, nu prin `;`, `.` sau alti separatori de acest fel
- o definitie se termina cand apare prima bucata de text aliniata la acelasi nivel cu ea sau mai la stanga

Cateva exemple:

```
square x = x*x      -- ok

square x =
    x*x              -- ok

square x =
x*x                  -- NOT ok
```

Definirea functiilor

```
fun v1 v2 ... vn = corp
```

SAU

```
fun v1 v2 ... vn =
    corp
```

SAU

```
fun v1 v2 ... vn
    | clauza1      = expresie1
    | clauza2      = expresie2
    ...
    | otherwise    = expresiep
```

atunci cand folosim mecanismul de pattern match.

Un exemplu de definire de functie cu pattern match:

```
maxi n m
    | n>m          = n
    | otherwise    = m
```

Se pot defini functii si ca in exemplul urmator:

```
factorial 0 = 1
factorial n = n * factorial (n-1)
```

Atentie! Nu este permisa re folosirea unei variabile intr-un pattern (in ideea ca programul se prinde ca este vorba de aceeasi valoare). De exemplu, pentru definitia:

```
sameInTuple (x,x) = True
```

veti primi urmatorul mesaj de eroare: `Repeated variable "x" in pattern`

Comentarii

- semnul -- face ca restul randului sa fie un comentariu
- pentru a comenta bucati mai mari de text, acestea se incadreaza intre {- si -}

Operatori pe numere intregi

`+, *` -- $2+3$, $5*6$

`^` -- $2^3 \Rightarrow 8$

`-` -- $5-2 \Rightarrow 3$

`div` -- $\text{div } 15 \ 4 \Rightarrow 3$, se poate scrie si $15 \text{ `div` } 4$

`mod` -- la fel se foloseste `mod a b` sau `a `mod` b`

`abs` -- valoare absoluta

`negate` -- schimba semnul

Se recomanda scrierea numerelor negative in paranteze: (-5) .

`>, >=, ==, <=, <`

`/=` -- pentru not equal

Operatorii pot fi convertiti in functii care isi preceda argumentele, daca ii punem intre paranteze. De exemplu:

`(+) 3 4 = 3+4`

Operatori pe numere reale

`**` -- $x**y$ inseamna x la puterea y unde ambele numere sunt reale

`cos, sin, tan, exp, log`

`fromInt :: Int -> Float`

`sqrt :: Float -> Float`

Operatori logici

`&&`

`||`

Liste

`[1,2,3] :: [Int]`

`['a','b','c'] :: [Char]`

`[True] :: [Bool]`

`[[1,2] , [3,4,5]] :: [[Int]]`

Elementele listei au un anumit tip `t` (nu sunt permise liste eterogene, dar sunt permise liste omogene de orice tip). Tipul listei va fi `[t]`.

Constructori si operatori pe liste

`[]`

:

O lista nevada se exprima in forma `x:xs`, unde `x` e capul (head) si `xs` e restul listei (tail).

Atentie! Operatorul `:` nu este prioritar, in general trebuie sa folositi scrierea `(x:xs)` atunci cand aplicati functii pe liste nevide.

`length :: [t] -> Int`

S-ar putea defini asa:

`length [] = 0`

`length (x:xs) = 1 + length xs`

`head :: [t] -> t`

`tail :: [t] -> [t]`

`head (x:_) = x`

`tail (_:xs) = xs`

In script se poate utiliza underscore ca mai sus pentru a defini o valoare oarecare, care nu intereseaza.

`sum` -- face suma elementelor din lista

`product` -- face produsul

`++` -- face append

Mai multe functii pentru liste gasiti la <http://www.haskell.org/ghc/docs/7.0-latest/html/libraries/base-4.3.1.0/Prelude.html#g:11>.

List comprehensions

List comprehensions reprezinta o modalitate de reprezentare a listelor apropiata de cea din matematica.

Exemple:

```
[3*x | x <- [1,2,3]]      -- va fi [3,6,9]
[3*x | x <- [1,2,3], x>2]  -- va fi [9]
addPairs pairList = [ x+y | (x,y) <- pairList ]
```

Cititi explicatiile si exemplele de la: http://www.haskell.org/haskellwiki/List_comprehension.