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)
```