Języki i Paradygmaty Programowania

Odkrywamy Haskell

Marcin Benke

24 lutego 2014

Odkrywamy Haskell

Haskell zasadniczo kompilowany, ale też interpreter: **ghci**

```
$ ghci
GHCi, version 7.6.2: http://www.haskell.org/ghc/
                                          :? for help

Prelude> 2 ^ 100
1267650600228229401496703205376

Prelude> words "Ala ma kota"
["Ala","ma","kota"]
Prelude> :type words
words :: String -> [String]
```

Listy

- [] — lista pusta
- x : xs — lista o głowie x i ogonie xs

```
Prelude> [1,2,3]
[1,2,3]
Prelude> [4] ++ [5,6]
[4,5,6]
Prelude> 1:[]
[1]
Prelude> [0..9]
[0,1,2,3,4,5,6,7,8,9]
Prelude> [1,3..10]
[1,3,5,7,9]
Prelude> take 4 [0..9]
[0,1,2,3]
```

Napisy

Napisy są listami znaków

```
Prelude> "napis"
"napis"
Prelude> ['H','a','s','k','e','l','l']
"Haskell"
Prelude> unwords ["hello","world"]
"hello world"
```

NB w razie potrzeby, istnieją bardziej efektywne implementacje, np. **Data.Text, Data.ByteString**.

Wycinanki listowe

W matematyce często tworzymy zbiory przy pomocy aksjomatu wycinania:

$$\{3x \mid x \in \{1,\ldots,10\},\ x \bmod 2 = 1\}$$

Podobnie możemy tworzyć listy w Haskellu:

```
> [3*x | x <- [1..10], mod x 2 == 1]
[3,9,15,21,27]
```

`x<-[1..10]` jest *generatorem*
`mod x 2 == 1` jest *filtrem*.

Możemy tworzyć wycinanki o większej liczbie generatorów i filtrów, np.

```
> [(x,y) | x<-[1..5],y<-[1,2,3],x+y == 5,x*y == 6]
[(2,3),(3,2)]
```

Definiowanie funkcji

Stwórzmy plik `e01.hs`

```
mul x y = x * y
square x =  mul x x
area r = pi * square r
```

teraz

```
$ ghci e01.hs
...
Ok, modules loaded: Main.
*Main> area 1
3.141592653589793
*Main> area 2
12.566370614359172
```

Program obowiązkowy: silnia

```
fact1 n = if(n <= 0) then
            1
          else
            n*fact1(n-1)
```

W Haskellu mozemy to zapisać także tak:

```
fact2 n | n <= 1 = 1
        | otherwise = n*fact2(n-1)
```

Silnia na N sposobów
Silnia na N sposobów: `tinyurl.com/silniaN`

Definicje przez przypadki (dopasowanie)
Jak obliczyć długość listy?

1º lista pusta ([ ]) $\rightsquigarrow$ 0

2º lista niepusta (h:t) $\rightsquigarrow$ 1 + długość t

```
len [] = 0
len (h:t) = 1 + len t
```

Takie definicje rekurencyjne spotykamy często zarówno w matematyce i w Haskellu.

Także silnię możemy zdefiniować w ten sposób:

```
fact3 0 = 1
fact3 n = n * fact3 (n-1)
```

Typy

Każda wartość ma swój typ (intuicyjnie możemy mysleć o typach jako o zbiorach wartości), np:

```
     True :: Bool
        5 :: Int
      'a' :: Char
[1, 4, 9] :: [Int]              -- lista
  "hello" :: String             -- (String = [Char])
("ala",4) :: (String, Int)      -- para
```

Funkcje oczywiście też mają typy:

```
isdivby :: Int -> Int -> Bool
isdivby x y = mod x y == 0
```

W większości wypadków Haskell potrafi sprawdzić poprawność typów bez żadnych deklaracji (rekonstruuje typy)

Podstawowe typy

- `Int` — maszynowe liczby całkowite
- `Integer` — liczby całkowite dowolnej precyzji
- `Char` — znaki (Unicode)
- `String` — napisy (Unicode)
- `Maybe`:
  - jeśli a jest typem, to `Maybe a` jest typem
  - `Nothing::Maybe a`
  - jeśli `x::a` to `(Just x)::Maybe a`
- Listy: `[a]` jest listą elementów typu `a`
  - `[]::[a]`
  - jeśli `x::a` oraz `xs::[a]`, to `(x:xs)::[a]`
- Krotki: $(\alpha_1, \ldots, \alpha_n)$ jest produktem kartezjańskim typów $\alpha_1, \ldots, \alpha_n$
- W szczególności `()` jest typem zawierającym jeden element: `()::()`

Wyrażenia

Podstawowe rodzaje wyrażeń w Haskellu to:

- literały, np. `42`, `"Foo"`;
- zmienne, np. `x` (muszą być z małej litery);
- zastosowanie funkcji do argumentu, np. `succ 9`;
- zastosowanie operatora infiksowego, np. `x + 1`;
- wyrażenie `if e then e' else e"`;
- wyrażenie `case` (analiza przypadków);
- wyrażenie `let` (lokalna deklaracja);
- wyrażenie λ (funkcja anonimowa).
- konstruktory (na kolejnym wykładzie)

Wyrażenie case

Wyrażenie **if** jest lukrem syntaktycznym dla **case**:

```
if e then e' else e''
```

jest tłumaczone do

```
case e of { True -> e' ;  False -> e'' }
```

W ogólności **case** ma postać

```
case e of { wzorzec1 -> e1;...; wzorzec_n -> e_n }
```

...i oznacza: *"przeanalizuj wyrażenie e i w zalezności od tego, do którego z wzorców pasuje, daj odpowiedni wynik"*

Układ czyli wcięcia mają znaczenie

Nawiasy klamrowe i średniki możemy pominąć, jeśli użyjemy odpowiednich wcięć w programie:

```
case e of
  True -> wyrazenie
     ew. dalszy ciąg wyrażenia bardziej wcięty
  False -> nastepny przypadek tak samo wcięty
```

Używanie układu nie jest obowiązkowe — jeśli użyjemy nawiasów i średników, możemy wcinać dowolnie.

Układ jest dopuszczalny także w innych konstrukcjach, takich jak **let**, **do**, **where**, itp.

Definicje

- Program (a ściślej moduł) w Haskellu jest zbiorem definicji funkcji, typów i klas.
- Kolejność tych definicji nie ma znaczenia.
- Definicja funkcji składa się z (opcjonalnej) sygnatury typu i ciągu równań (dla różnych przypadków — tu kolejność ma znaczenie).
- Równania mogą być rekurencyjne, mogą też używać innych funkcji, także tych zdefiniowanych niżej.

```
f :: Int -> Int
f 0 = 1
f n = n * g(n-1)

g n = n * f(n-1)
g 0 = 1
```

Kolejność równań

```
f :: Int -> Int
f 0 = 1
f n = n * f(n-1)

g n = n * g(n-1)
g 0 = 1

kolejnoscRownan.hs:5:0:
    Warning: Pattern match(es) are overlapped
             In the definition of `g': g 0 = ...
Ok, modules loaded: Main.
*Main> f 5
120
*Main> g 5
*** Exception: stack overflow
*Main>
```

Dopasowywanie wzorców

Wzorce moga być bardziej złożone, np

```
third :: [a] -> a
third (x:y:z:_) = z
```

(podkreślenie oznacza "cokolwiek")

Równanie moze też wymagać dopasowania więcej niż jednego argumentu:

```
myzip :: [a] -> [b] -> [(a,b)]
myzip (x:xs) (y:ys) = (x,y):myzip xs ys
myzip _ _ = []
```

Dopasowanie wzorców działa w równaniach podobnie jak w **case** (i innych miejscach), poza tym, że w **case** oczywiście dopasowujemy tylko jedno wyrażenie.

Wyrażenie let

Wyrażenie **let** pozwala nam na użycie lokalnych definicji pomocniczych dla obliczenia jakiegoś wyrazenia:

```
let { definicja1; ...; definicja_n } in e
```

Tak jak przy **case** możemy uzyć wcięć zamiast nawiasów i średników, np.

```
let
  x = 1
  y = 2
in x + y
```

a nawet

```
let answer = 42 in answer
```

Definicje w **let** mogą być wzajemnie zależne (tak jak na najwyższym poziomie).

Wyrażenie let

W wyrażeniu **let** możemy definiować funkcje, używać rekurencji i dopasowań:

```
f xs = let
    len [] = 0
    len (x:xs) = 1 + len xs
  in len xs
```

Uwaga: reguły wcinania wymagają aby w tym wypadku:

- len było bardziej wcięte niż linia, w której zaczyna się **let** (czy f)
- oba równania dla len były tak samo wcięte
- **in** było wcięte mniej niż len, ale bardziej niż f

(oczywiście jeśli uzywamy `{ ; }` mozemy wcinać dowolnie).

Wyrażenie λ

- Wyrażenie λ pozwala na skonstruowanie i użycie w wyrażeniu funkcji anonimowej (jak w rachunku lambda).
- Tekstowo lambdę zapisujemy jako backslash

  ```
  (\x -> x + 1) 9
  10
  ```

- Możemy używać dopasowania wzorca (ale tylko jeden przypadek):

  ```
  pierwszy = \(x,y) -> x
  ```

- Może być więcej niż jeden argument:

  ```
  (\x y -> x) 1 2
  ```

  oznacza to samo co

  ```
  (\x -> (\y -> x)) 1 2
  ```

Podprzypadki

Zadanie: podzielić listę na dwie: elementy $\leqslant n$ oraz $> n$

```
splitBy :: Int -> [Int] -> ([Int],[Int])
splitBy n [] = ([],[])
splitBy n (x:xs) = let (ys,zs) = splitBy n xs in
  if x<= n then (x:ys,zs) else  (ys,x:zs)
```

Drugi przypadek naturalnie dzieli sie na dwa podprzypadki; nie możemy tego zapisać przez wzorce, ale możemy tak:

```
splitBy' n (x:xs)
  | x<=n = let (ys,zs)=splitBy' n xs in (x:ys,zs)
  | x>n  = let (ys,zs)=splitBy' n xs in (ys,x:zs)
```

Klauzula where

```
splitBy' n (x:xs)
  | x<=n = let (ys,zs)=splitBy' n xs in (x:ys,zs)
  | x>n  = let (ys,zs)=splitBy' n xs in (ys,x:zs)
```

W obu przypadkach powtarza się ta sama definicja, możemy to krócej zapisać:

```
splitBy'' n (x:xs)
  | x<=n = (x:ys,zs)
  | otherwise = (ys,x:zs)
  where (ys,zs) = splitBy'' n xs
```

where jest poniekąd podobne do **let**, ale

- **let** jest wyrażeniem, **where** jest częścią definicji
- Zasięgiem definicji w **let** jest wyrażenie po **in**; zasięgiem definicji w **where** — całe równanie

Operatory infiksowe

- Nazwy złożone z symboli domyślnie są używane w składni infiksowej: `xs ++ ys` to to samo co `(++) xs ys`.
- Ujęcie operatora w nawiasy "odbiera mu infiksowość".
- Podobnie ujęcie nazwy prefiksowej (z liter i cyfr) w odwrócone apostrofy "nadaje jej infiksowość": `x `mod` 2`
- Operatory nie są magiczne, możemy definiować własne:

```
infixl 6 +++
(+++) :: Int -> Int -> Int
x +++ y = (x+y)*(x+y+1) `div` 2
```

Deklaracja `infixl...` oznacza, że `+++` wiąże w lewo i ma priorytet 6 (taki sam jak +)

- Niektóre ciągi symboli są zastrzeżone: `..  ::  =  \  |  <-  ->  @  ~  =>`
- `--` (lub więcej −) rozpoczyna komentarz, ale np `+--` ani `--+` nie

Jeszcze o priorytetach

- Minusa unarnego nie należy mieszać z operatorami infiksowymi:

```
Prelude> 2 * -3
 Precedence parsing error
   cannot mix `*' [infixl 7] and prefix `-' ...
Prelude> 2 * (-3)
-6
```

- Prefiksowa aplikacja funkcji ma wyższy priorytet niż operatory infiksowe; wiąże w lewo, czyli `f x y` oznacza `(f x) y`
- Z kolei `($)` jest operatorem aplikacji funkcji o priorytecie 0, wiążącym w prawo, co pozwala oszczędzić nawiasów i napisać np.

```
length $ show $ foldl (*) 1 [1..1000]
```

zamiast

```
length ( show ( foldl (*) 1 [1..1000] ))
```

...chociaż prawdopoodobnie ta druga notacja jest dle wielu osób czytelniejsza; Haskell kładzie jednak nacisk na zwięzłość.

Przekroje

Operatory infiksowe są z natury swej dwuargumentowe. Podając operatorowi jeden z argumentów mozemy uzyskać funkcję jednoargumentową.

Konstrukcja taka nazywa się *przekrojem* (section) operatora.

```
Prelude> (+1) 2
3
Prelude> (1+) 3
4
Prelude> (0-) 4
-4
```

Przekrojów używamy przeważnie, gdy chcemy taką funkcję przekazać do innej funkcji.