Programowanie w języku Haskell

Podstawy

Marcin Szlenk (IAiIS PW)

1 marca 2018

Strona domowa: http://www.haskell.org

Najpopularniejszą implementacją jest GHC (The Glasgow Haskell Compiler).
Najwygodniej zainstalować *Haskell Platform*
(http://www.haskell.org/platform)

Materiały źródłowe:

- Graham Hutton, *Programming in Haskell (second edition)*, Cambridge University Press 2016 (http://www.cs.nott.ac.uk/~pszgmh/book.html)
- Fethi Rabhi, Guy Lapalme, *Algorithms: A Functional Programming Approach*, Addison-Wesley 1999

Przykładowy plik źródłowy (`prog.hs`):

```
{- komentarz -}
x = 5                    -- komentarz
increment n = n + 1
main = print (increment x)
```

Kompilacja z optymalizacją:

```
C:\> ghc prog.hs -O2 -o prog.exe
C:\> prog.exe
6
```

Uruchomienie interpretera GHCi:

```
C:\> ghci
ghci> :load prog.hs
ghci> x
5
ghci> increment 0
1
ghci> main
6
```

Polecenie `:reload` powoduje ponowne wczytanie pliku

Biblioteka standardowa (`Prelude.hs`) oferuje szereg funkcji i operatorów:

```
ghci> (2+3)*4
20
ghci> sqrt (3^2 + 4^2)
5.0
ghci> head [1,2,3,4,5]
1
ghci> sum [1,2,3,4,5]
15
```

Definiowanie wartości:

```
pi = 3.141592653589793
```

Definiowanie funkcji:

```
square x = x ^ 2
triangleArea a h = 0.5 * a * h
```

Nazwy wartości i funkcji muszą zaczynać się małą literą. Dalej mogą wystąpić litery, cyfry, znaki podkreślenia (_) lub apostrofu (')

Aplikowanie funkcji do argumentów:

```
ghci> square 2
4
ghci> triangleArea 4 2
4.0
```

Aplikacja funkcji ma wyższy priorytet niż pozostałe operatory:

```
ghci> triangleArea 4 2 * square 2 + 1
17.0
```

Czasem potrzebne są nawiasy:

```
ghci> triangleArea (square 2) 2
4.0
```

Tabela 1: Aplikowanie funkcji

Matematyka	Haskell
$f(x)$	`f x`
$f(x, y)$	`f x y`
$f(g(x))$	`f (g x)`
$f(x, g(y))$	`f x (g y)`
$f(x)g(y)$	`f x * g y`

Klauzula `where` pozwala definiować lokalne wartości i funkcje:

```
volume r = a * pi * cube r
    where a = 4 / 3
          cube x = x ^ 3
```

Podobnie wyrażenie `let-in`:

```
volume r = let a = 4 / 3
               cube = r ^ 3
           in a * pi * cube
```

Wartości warunkowe można zdefiniować przy użyciu wyrażenia `if-then-else`:

```
sgn x = if x < 0 then -1
        else if x == 0 then 0
             else 1
```

Gałąź `else` jest wymagana

Innym sposobem definiowania wartości warunkowych są tzw. strażnicy (*guards*):

```
sgn x | x < 0      = -1
      | x == 0     = 0
      | otherwise = 1
```

Strażnicy są sprawdzani w kolejności ich zdefiniowania

Kolejny sposób zdefiniowania wartości warunkowych to wykorzystanie tzw. dopasowania wzorca (*pattern matching*):

```
conjunction True b = b
conjunction a    b = False
```

Wzorce są rozpatrywane w kolejności ich zdefiniowania

W miejsce nazwy argumentu, który nie jest używany do obliczenia zwracanej wartości, można użyć podkreślenia (tzw. *wildcard*):

```
conjunction True b = b
conjunction _    _ = False
```

Dopasowanie wzorca występuje również w wyrażeniu `case-of`:

```
check n = case n < 0 of
              True  -> "liczba ujemna"
              _     -> "liczba dodatnia"
```

Rozpatrzmy matematyczną definicję silni:

$$n! = \begin{cases} 1, & n = 0 \\ n(n-1)!, & n \neq 0 \end{cases}$$

Alternatywne definicje w Haskellu:

```haskell
factorial n = if n == 0 then 1
              else n * factorial (n - 1)
```

```haskell
factorial n | n == 0    = 1
            | otherwise = n * factorial (n - 1)
```

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

```haskell
factorial n = case n of 0 -> 1
                        _ -> n * factorial (n - 1)
```

Prześledźmy proces obliczania wartości funkcji:

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

```
factorial 2
= 2 * factorial (2 - 1)
= 2 * factorial 1
= 2 * (1 * factorial (1 - 1))
= 2 * (1 * factorial 0)
= 2 * (1 * 1)
= 2
```

Operator dwuargumentowy jest funkcją, ale...

Definiując go, nazwę zapisujemy pomiędzy argumentami:

```
x & y = (x + y) / 2
```

Jego aplikacja ma niższy priorytet niż aplikacja funkcji:

```
ghci> factorial 1 & 2
1.5
```

Nazwa operatora może składać się z symboli: `!#$%&*+./<=>?@\^|-~:`

Można przekształcić operator dwuargumentowy w funkcję, zapisując jego nazwę pomiędzy nawiasami:

```
sum  a b = a + b
sum' a b = (+) a b
```

Można przekształcić funkcję dwuargumentową w operator, zapisując jej nazwę pomiędzy odwrotnymi apostrofami:

```
v  = triangleArea 4 2
v' = 4 `triangleArea` 2
```

Zasady formatowania kodu:

1. Wszystkie definicje muszą zaczynać się w tej samej kolumnie:

```
abs x = if x < 0 then -x else x
a = 5
```

2. Kolejne linie, wcięte w stosunku do początku definicji, są traktowane jako jej kontynuacja:

```
abs x =
    if x < 0
       then -x
          else x
```

3. Powyższe zasady dotyczą również definicji lokalnych po słowach `where` i `let` oraz wyrażeń po słowie `of`:

```
a = b + c
    where
        b = 1
        c = 2
d = a * 2
```

Uwaga!

Znak tabulatora jest traktowany jak 8 znaków spacji

W razie potrzeby definicje mogą być grupowane i oddzielane *explicite*:

```
a = b + c
    where { b = 1;
            c = 2 }
d = a * 2
```

co pozwala np. zapisać je w jednej linii:

```
a = b + c where { b = 1; c = 2 }; d = a * 2
```

Idea programowania piśmiennego (*literate programming*):

Traktuj program jako informację dla istot ludzkich,
a nie zbiór instrukcji dla komputera

Tekst w pliku z rozszerzeniem `.lhs` jest komentarzem, a linie kodu trzeba oznaczyć:

```
Poniższa funkcja zwraca wartość bezwzględną:

> abs n | n >= 0    = n
>       | otherwise = -n

Użyjemy jej do policzenia...
```

Ćwiczenia podsumowujące:

1. Napisz funkcję zwracającą największy wspólny dzielnik dwóch liczb. Skorzystaj z algorytmu Euklidesa:

$$\mathrm{NWD}(a, 0) = a,$$
$$\mathrm{NWD}(a, b) = \mathrm{NWD}(b, a \bmod b)$$

   oraz funkcji standardowej `mod`, która zwraca resztę z dzielenia liczb:

```
ghci> mod 3 2
1
```

2. Korzystając z tego samego algorytmu, zdefiniuj operator `//` zwracający największy wspólny dzielnik dwóch liczb
3. Znajdź trzy błędy w poniższej implementacji funkcji liczącej pole trójkąta ze wzoru Herona:

$$A(a,b,c) = \sqrt{s(s-a)(s-b)(s-c)}, s = \frac{(a+b+c)}{2}$$

```
area a b c = sqrt s * (s - a) * (s - b) * (s - c)
   where
      Abc = a + b + c
        s = Abc / 2
```