Haskell – wprowadzenie

Inspiracja

Materiały do ćwiczeń będą oparte w dużej mierze na materiałach dr. Sławomira Bakalarskiego (do tego samego kursu).

Interpretery Haskella

- Będziemy korzystać z ghc – The Glasgow Haskell Compiler.
- Można zainstalować ze strony internetowej www.haskell.org lub np. przez Package Manager jeśli używają Państwo systemu Linux.
- Będziemy korzystać z interaktywnej wersji ghc, czyli ghci.
- W ostateczności można się uciec do wersji online, np. https://replit.com/languages/haskell.

Podstawowe idee

Języki funkcyjne nie przypominają C, C++, Pythona czy Javy.

- Nie używamy zmiennych (ale możemy definiować stałe).
- Nie używamy pętli (ale mamy do dyspozycji rekurencję).
- Będziemy się zajmować definiowaniem **funkcji**, które będą wykonywać to, co chcemy.
- Bardzo ważne będzie składanie funkcji (wbudowanych oraz samodzielnie zdefiniowanych).
- Funkcje mogą mieć wiele argumentów, którymi mogą być liczby, ale też ciągi znaków, listy czy **inne funkcje**.
- Funkcję n zmiennych z ustalonymi wartościami na $n - k$ argumentach możemy traktować jak funkcję k zmiennych.

Przykład funkcji

Poniższa funkcja dodaje do liczby (typu Int) jedynkę:

```
dodajJeden :: Int -> Int
dodajJeden x = x + 1
```

- Funkcje nazywamy z małej litery.
- Zaczynamy od podania **sygnatury funkcji** (jakiego typu są przyjmowane argumenty oraz zwracana wartość; Haskell jest silnie typowany).
- Po podaniu sygnatury podajemy definicję funkcji.

Korzystanie ze zdefiniowanych funkcji

- Tworzymy plik z rozszerzeniem *.hs, który chcemy skompilować (np. zawierający definicję funkcji dodajJeden ze slajdu wyżej).
- Uruchamiamy ghci.
- Komendą `:l <nazwaPliku>` (wystarczy bez rozszerzenia .hs) ładujemy nasz plik.
- Jeśli plik się kompiluje dostajemy dostęp do zdefiniowanych funkcji, w przykładzie z funkcją dodajJeden możemy teraz np. wywołać `dodajJeden 2` i otrzymać 3.

Podstawowe typy

Przykłady typów występujących w Haskellu:

- Int – "standardowy" int znany z innych języków.
- Integer – liczba całkowita (bez ograniczeń z dokładnością do skończonej pamięci).
- Float, Double – typy zmiennoprzecinkowe.
- Char – pojedynczy znak, używamy pojedynczych '.
- String – ciąg znaków, realizowany jako lista elementów typu Char, używamy ".
- Bool – True lub False.

Typy c.d.

Nazwy typów pisane są z dużej litery, jeśli nie określimy typu stałej wprost, Haskell spróbuje "zgadnąć" o co nam chodziło, np.

```
cztery = 4
```

spowoduje powstanie stałej cztery typu **Integer** (nie **Int**!). Typ możemy sprawdzić w ghci używając `:t <obiekt>`, w szczególności możemy sprawdzać typy nie tylko stałych, ale też funkcji. Jeśli chcemy wymusić typ stałej możemy to zrobić podobnie, jak przy definicji funkcji:

```
cztery :: Int
cztery = 4
```

Po każdej zmianie pliku nie musimy wpisywać `:l <nazwaPliku>`, wystarczy `:r`, który załaduje ponownie plik ładowany wcześniej. Ponieważ w ghci Ctrl-C przerywa wykonywanie aktualnie wykonywanej komendy wewnątrz ghci, aby wyjść używamy `:q` (lub Ctrl-D).

Funkcje w Haskellu

Spójrzmy na minimalnie bardziej złożoną definicję funkcji:

```
dodajTrzy :: Int -> Int -> Int -> Int
dodajTrzy x y z = x + y + z
```

Jest to oczywiście funkcja przyjmująca trzy argumenty typu Int, która zwraca wartość typu Int.

Funkcje w Haskellu

Spójrzmy na minimalnie bardziej złożoną definicję funkcji:

```
dodajTrzy :: Int -> Int -> Int -> Int
dodajTrzy x y z = x + y + z
```

Jest to oczywiście funkcja przyjmująca trzy argumenty typu Int, która zwraca wartość typu Int.
Równie dobrą interpretacją jest jednak również:
Jest to funkcja przyjmująca jako argument Int oraz zwracająca funkcję, która przyjmuje dwie wartości typu Int oraz zwraca wartość typu Int. Innymi słowy, powyższa sygnatura oraz sygnatura

```
dodajTrzy :: Int -> (Int -> Int -> Int)
```

czy nawet

```
dodajTrzy :: Int -> (Int -> (Int -> Int))
```

to w Haskellu dokładnie to samo (o czym można samemu się przekonać używając `:t`).

Funkcje w Haskellu c.d

Z powyższym przykładem można pójść jeszcze dalej:

- Funkcja `(dodajTrzy 7)` jest funkcją dwuargumentową, która przyjmuje dwie wartości typu Int i zwraca jako wynik ich sumę powiększoną o 7 (typu Int).
- Funkcja `(dodajTrzy 64 36)` jest funkcją jednoargumentową, która przyjmuje wartość typu Int i zwraca jako wynik tę wartość powiększoną o 100 (typu Int).

Odpowiada to wprost traktowaniu funkcji wielu zmiennych z ustalonymi niektórymi argumentami jako funkcji mniejszej liczby zmiennych. Jeśli mamy funkcję $f : \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ daną przez $f(x, y, z) = x + y + z$, to przez $f(7, \cdot, \cdot)$ rozumiemy funkcję dwuargumentową, z $\mathbb{Z} \times \mathbb{Z}$ w $\mathbb{Z}$, która dana jest "takim samym wzorem" co f, z dokładnością do ustalenia pierwszego argumentu na 7. Podobnie $f(64, 36, \cdot)$ jest funkcją z $\mathbb{Z}$ w $\mathbb{Z}$.
Ponownie, możemy się o tym przekonać w ghci za pomocą `:t`.

Operatory

Operatory, które mamy do dyspozycji:

- Matematyczne: +, -, /, *;
- Porównań: >, >=, <, <=, ==, /=;
- Logiczne: &&, ||;

Stosowane infiksowo, np. 2+3, 3.0/4.0, ale również:

- `mod` – reszta modulo;
- `div` – dzielenie całkowite;
- `not` – zaprzeczenie logiczne

Stosowane prefiksowo, np. `mod 13 10`.
Operatory infiksowe można stosować prefiksowo biorąc je w nawias, np. `(*) 2 3`, a operatory prefiksowe infiksowo używając ‘ (klawisz z tyldą, nie cudzysłów!), np. `13 ‘mod‘ 10`.

Funkcje przyjmujące funkcje

Jak już zostało wspomniane, funkcje poza liczbami mogą też przyjmować inne funkcje np.

```
sumaWartosci :: (Int -> Int) -> (Int -> Int) -> Int ->
 Int -> Int
sumaWartosci f g x y = (f x) + (g y)
```

jest funkcją, która przyjmuje dwie jednoargumentowe funkcje oraz dwie wartości, wykonuje podane funkcje na podanych wartościach (pierwszą na pierwszej, drugą na drugiej) i dodaje wyniki.

Instrukcje warunkowe

Przykładową funkcję

$$f = \begin{cases} -x^2, & \text{dla } x < 0 \\ x^2, & \text{w przeciwnym przypadku} \end{cases}$$

możemy zrealizować na parę sposobów (pomijam sygnaturę):

```
-- dozory (guards)
f x | x < 0 = - x * x
    | otherwise = x * x

-- dozory (guards), w jednej linijce
f x | x < 0 = - x * x | otherwise = x * x

-- "standardowy" if-then-else
f x = if x < 0 then - x * x else x * x
```

Instrukcje warunkowe c.d

Innym przykładem może być dopasowywanie do wzorca:

```
ocena :: Double -> String
ocena 2.0 = "niezaliczone"
ocena 5.0 = "brawo!"
ocena x = "wpisane masz " ++ show x
```

Jeśli nie potrzebujemy wartości argumentu, możemy użyć znaku "_":

```
ocena :: Double -> String
ocena 2.0 = "niezaliczone"
ocena 5.0 = "brawo!"
ocena _ = "inne"
```

Instrukcje warunkowe – uwagi

- Jeśli używamy "if", **zawsze** musimy użyć "else".
- Stosując dozory (guards) lub dopasowywanie do wzorca możemy wypisać dowolnie dużo przypadków.
- Jeśli używamy dozorów, to "otherwise" jest opcjonalne, podobnie jak stosowanie w dopasowywaniu do wzorca na końcu takiego, który "pasuje do wszystkiego". Jeśli jednak podczas obliczania funkcji trafimy na nieobsłużoną sytuację, wykonywanie zakończy się błędem.
- Przypadki czytane są "od góry", po natrafieniu na pierwszy pasujący to właśnie on jest wybierany.

Rekurencja

Używając dopasowania do wzorca możemy wykorzystywać funkcję, którą właśnie definiujemy (to my musimy zadbać, żeby miało to sens), co bezpośrednio umożliwia nam stosowanie definicji rekurencyjnych:

```
silnia :: Integer -> Integer
silnia 0 = 1
silnia n = n * silnia (n-1)
```

Jeśli jednak w ostatniej linijce zamiast $n-1$ użyjemy $n+1$ lub też zamienimy warunki miejscami (znajdywany jest pierwszy pasujący od góry!), to wykonywanie np. `silnia 5` nigdy się nie zakończy.

Listy

Jeśli T jest typem, to $[T]$ oznacza listę elementów typu T.

- `[]` oznacza listę pustą.
- Listy mogą zawierać tylko elementy jednego rodzaju (są *jednorodne*).
- Listy można konkatenować operatorem $++$ (w szczególności String jest listą elementów typu Char!).
- Do listy możemy dodać element na początek operatorem :, np. `1:[2,3]` oraz `1:2:3:[]` dadzą nam jako wynik listę `[1,2,3]`.
- Możemy się odwołać do konkretnego elementu listy po indeksie operatorem !!, np. `[1,2,3] !! 1` da nam wynik 2 (listy indeksowane są od zera!).

Krotki

Jeśli T_1, T_2 są typami, to (T_1, T_2) oznacza parę o elementach odpowiednio T_1 i T_2. Analogicznie możemy tworzyć krotki większej długości. Przykładowo `(Int,Double)` czy `(Int,Double,Integer)`.

- Wywołanie `fst (a,b)` zwraca pierwszy element pary, czyli a.
- Wywołanie `snd (a,b)` zwraca drugi element pary, czyli b.

Listy c.d.

- `[0..1000]` utworzy listę zawierającą wszystkie liczby naturalne nie większe niż 1000.
- `[0,2..1000]` utworzy listę zawierającą wszystkie naturalne liczby parzyste nie większe niż 1000.
- `['A'..'Z']` utworzy listę zawierającą duże litery alfabetu, po kolei.
- `['A','C'..'Z']` utworzy listę zawierającą co drugą literę alfabetu, po kolei.
- `[1,2..]` utworzy listę wszystkich dodatnich liczb naturalnych.
- `[10,20...]` utworzy listę wszystkich dodatnich wielokrotności liczby 10.

Ważnym szczególnie w kontekście dwóch ostatnich list jest fakt, że Haskell jest leniwy, tzn. liczy dopiero wtedy, kiedy musi, więc na przykład `[0,10..] !! 5` poprawnie zwróci 50, nie zostanie policzona cała nieskończona lista.

Listy c.d.

- Funkcje `head` i `last` zwracają odpowiednio pierwszy i ostatni element listy.
- Funkcje `init` i `tail` zwracają odpowiednio wszystkie elementy poza ostatniem oraz wszystkie poza pierwszym.
- Wywołanie `take n list` zwraca *n* pierwszych elementów listy.
- Wywołanie `drop n list` opuszcza *n* pierwszych elementów listy.

`head`, `last`, `init` oraz `tail` wywołane na pustej liście zwrócą błąd.

Przykład połączenia list i dopasowania do wzorca

Przykład funkcji, która zareaguje inaczej dla listy pustej, jednoelementowej, dwuelementowej oraz pozostałych:

```
lista :: [Int] -> String
lista [] = "Lista jest pusta"
lista (x:[]) = "Lista zawiera tylko " ++ show x
lista (x:y:[]) = "Lista zawiera tylko " ++ show x ++
                 " oraz " ++ show y
lista (x:xs) = "Lista zaczyna sie od " ++ show x ++
               ", reszta to " ++ show xs
```

where oraz let

Klauzula `where` umożliwia stosowanie funkcji/stałych, których potrzebujemy tylko lokalnie. Przykład funkcji obliczającej Symbol Newtona $\binom{n}{k}$ i wykorzystującej `where`:

```
{- Symbol Newtona n po k -}
binomial :: Int -> Int -> Int
binomial n k = fact n `div`(fact k * fact (n - k))
               where
                 fact 0 = 1
                 fact n = n * fact (n - 1)
```

where oraz let c.d.

Analogiczny przykład, wykorzystujący `let`:

```haskell
{- Symbol Newtona n po k -}
binomial :: Int -> Int -> Int
binomial n k = let fact 0 = 1
                   fact n = n * fact (n - 1)
               in  fact n `div`(fact k * fact (n - k))
```

O różnicach między `where` oraz `let` można przeczytać dokładniej na stronie:
http://learnyouahaskell.com/syntax-in-functions#let-it-be

Polecane strony (dr S. Bakalarski)

- http://learnyouahaskell.com/
- http://book.realworldhaskell.org/
- https://hoogle.haskell.org/