Haskell. Programowanie czysto funkcyjne

Popularność języków funkcyjnych stale rośnie. Paradygmat programowania funkcyjnego jest z powodzeniem adoptowany w językach obiektowych. Jeżeli na co dzień piszemy w C#, korzystamy z rozszerzeń Linq. Tworzone potoki wywołań i przekazywanie do nich funkcji anonimowych to przykład programowania funkcyjnego. JavaScript filtruje i mapuje kolekcje, pozwala przekazywać funkcje jako argumenty do innych funkcji. To jest programowanie funkcyjne. Kod jest krótki, czytelny i mniej podatny na błędy.

Następnie napotykamy na grupę języków, gdzie paradygmat programowania funkcyjnego jest wysunięty na pierwszy plan – OCaml, Scala, F#. Jest to grupa języków mieszanych, gdzie paradygmat funkcyjny możemy łączyć z programowaniem obiektowym czy imperatywnym. Haskell jest językiem czysto funkcyjnym. W porównaniu do wymienionych wcześniej bezkompromisowo związany z paradygmatem funkcyjnym, w którym nie ma możliwości stosowania innych konwencji programowania.

POWSTANIE JĘZYKA

Początki Haskella sięgają 1987 roku. Wtedy odbyła się konferencja (FPCA '87), podczas której uznano, że należy zunifikować wiele istniejących czysto funkcyjnych języków w jeden standard. Konsekwencją było powstanie języka Haskell, nazwanego na cześć amerykańskiego logika i matematyka Haskella Curry'ego. W roku 1990 światło dzienne ujrzała wersja 1.0 języka. Kolejne wersje 98 i 2010 pojawiły się odpowiednio w 1997 i 2009 roku. Kompilator języka został stworzony przez zespół związany z Uniwersytetem w Glasgow. Jego pełna, dosyć patetyczna nazwa, to *The Glorious Glasgow Haskell Compilation System* lub krócej *Glasgow Haskell Compiler* – albo po prostu GHC. Zostańmy przy akronimie GHC, z wiadomych względów. GHC jest dystrybuowane jako część pakietu *Haskell Platform*, która oprócz samego kompilatora zawiera zbiór standardowych bibliotek języka oraz innych narzędzi. *Haskell Platform* jest dostępne na wszystkie najpopularniejsze systemy operacyjne (Windows, Linux, Mac OS).

CECHY JĘZYKA

Jeżeli C, C++ to języki systemów operacyjnych, Java oraz C# to języki biznesu, to Haskellowi najbliżej do matematyki. Bez obaw, nie chodzi tu o matematyczne teorie. Haskell z matematyki czerpie pewne koncepty – ścisłość, prostotę wyrazu, elegancję oraz zwięzłość.

Cechy języka Haskell:

- funkcje czyste
- typowanie statyczne
- wnioskowanie typów
- polimorfizm
- wartości są niezmienne
- funkcje są wartościami
- funkcje mogą być wywoływane częściowo
- leniwe wartościowanie
- *pattern matching*
- *list comprehensions*
- *typeclasses*
- rozbudowany system typów (listy, krotki, typy algebraiczne, monady)

INSTALACJA HASKELL PLATFORM I URUCHOMIENIE GHCI

Pakiet instalacyjny pobieramy z https://www.haskell.org/platform/. Po zainstalowaniu platformy możemy uruchomić sesję interaktywną Haskella. W systemie Windows w wierszu poleceń wpisujemy komendę `ghci` lub wybieramy z listy zainstalowanych programów skrót GHCi. Osobiście preferuję tę druga opcję – okienko ma fajną ikonkę przedstawiającą literę lambda, która jest logiem Haskella. Po uruchomieniu konsoli sprawdźmy, czy Haskell potrafi dodawać:

Listing 1. Pierwsze wyrażenie w GHCi

```
Prelude> let x = 7 + 35
Prelude> x
42
```

Ok, potrafi. W tej chwili możemy wyrzucić wszystkie swoje domowe kalkulatory i liczydła.

Jeżeli nie odpowiada nam konsola, polecam edytor Sublime Text, który obsługuje Haskella *out of the box*. Tworzymy nowy plik z rozszerzeniem `.hs`, na przykład `hello.hs`, w pliku umieszczamy kod:

Listing 2. Pierwszy program

```
x = show (7 + 35)
main = do
  putStrLn x
```

Wybieramy polecenie „Build" (Ctrl + B)

Jak widać, aby uzyskać ten sam efekt, musieliśmy napisać odrobinę inny kod. W sesji interaktywnej należy jawnie definiować wartości nazwane słowem kluczowym `let`, natomiast w edytorze uruchomiliśmy pełnoprawny program w Haskellu (potrzebowaliśmy do tego metody `main`, a także zamienić wynik naszego obliczenia na reprezentację tekstową funkcją `show`).

TYPY I PROSTE WYRAŻENIA

Otwórzmy ponownie konsolę GHCi. Znak zachęty `Prelude>` oznacza, że w tej chwili w naszej sesji interaktywnej załadowany jest moduł `Prelude` (moduły zostaną opisane w dalszej części artykułu). Jest on domyślnie importowany i zawiera podstawowe typy danych (m.in. liczby całkowite, liczby zmiennoprzecinkowe, typ `Bool`, typ znakowy, listy i krotki), funkcje operujące na tych typach oraz operacje wejścia/wyjścia.

Wykonamy teraz kilka operacji w sesji GHCi na podstawowych typach danych.

Listing 3. Kilka operacji w sesji GHCi na podstawowych typach danych

```
Prelude> 7 + 35 -- komentarz w sesji interaktywnej
42
Prelude> (50 + 2 - 1) * 2 - 60 -- użycie nawiasów do grupowania
wyrażenia
42
Prelude> True
True
Prelude> True && False -- operacja na typach Bool
False
Prelude> True || False -- operacja na typach Bool
True
Prelude> 1 == 1 -- test na równość
True
Prelude> not (1 == 1) -- negacja logiczna
False
Prelude> max 1 5 -- wywołanie funkcji, parametry oddzielone
spacjami
5
Prelude> sqrt 4 -- sqrt zwraca typ Floating
2.0
Prelude> sqrt 2 > min 1 2
True
Prelude> "Hello" -- łańcuch znaków
"Hello"
Prelude> "Hello" ++ " " ++ "World" -- konkatenacja
"Hello World"
Prelude> [1,2,3] -- lista
[1,2,3]
Prelude> length [1,2,3] -- długość listy
3
Prelude> head [1,2,3] -- pierwszy element listy
1
Prelude> last [1,2,3] -- ostatni element listy
3
Prelude> (4,2) -- para (krotka dwuelementowa)
(4,2)
Prelude> fst (4,2) -- pierwszy element pary
4
Prelude> snd (4,2) -- drugi element pary
2
Prelude> ['H','i','!'] -- napisy są listami znaków
Hi!
Prelude> :quit
Leaving GHCi.
```

FUNKCJE – PODSTAWY

Funkcja jest centralnym i najważniejszym obiektem w świecie Haskella. W programowaniu funkcyjnym rozwiązanie konkretnego problemu sprowadza się do napisania zbioru niewielkich, atomowych funkcji, a następnie połączenie tych klocków w odpowiedni sposób. Specyfika języka wymusza tworzenie funkcji czystych (ang. *pure functions*), czyli takich, których wywołania nie powodują efektów ubocznych. Funkcje tego typu są deterministyczne – dla tych samych argumentów zwracają zawsze ten sam wynik. Nie zależą od żadnego zewnętrznego stanu ani nie modyfikują go. Na przykład funkcja, która inkrementuje zmienną globalną i następnie zwraca jej wartość, nie jest funkcją czystą – jej wynik zależy od wartości tej zmiennej globalnej, która z kolei może być modyfikowana przez inny fragment programu. Haskell nie dopuszcza do takiej sytuacji, ponieważ wartości są niezmienne. Na początku może się to wydawać bardzo ograniczające, zwłaszcza jeśli przyzwyczailiśmy się do programowania w stylu imperatywnym. Z czasem okazuje się, że niezmienność ma swoje zalety. Przede wszystkim zapewnia nam, wspomniany wcześniej, determinizm oraz brak efektów ubocznych, a w konsekwencji prowadzi do powstania bardziej przewidywalnego i wolnego od błędów kodu.

Zanim zdefiniujemy nasze pierwsze proste funkcje, dostosujemy nieco konsolę GHCi do naszych potrzeb. Po pierwsze, umożliwimy wprowadzenie wielowierszowego kodu. Druga opcja to wyświetlanie typu wyrażenia zaraz po jego ewaluacji. Jeżeli będziemy chcieli sprawdzić typ funkcji wbudowanej lub funkcji, którą wcześniej zdefiniowaliśmy, przydatna będzie komenda: `:t <nazwa>`.

Listing 4. Ustawienia sesji interaktywnej

```
Prelude> :set +m
Prelude> :set +t
```

I gotowe. Spróbujmy na początek sprawdzić typ funkcji `sqrt`:

Listing 5. Typ funkcji sqrt

```
Prelude> :t sqrt
sqrt :: Floating a => a -> a
```

Ok, sprawdziliśmy typ funkcji `sqrt`, `a -> a` oznacza funkcję, która przyjmuje jeden argument i zwraca wynik tego samego typu co ten argument. `Floating a =>` to *class constraint*. Jest to informacja, że funkcja przyjmuje parametr `a`, który musi być pewnego typu (konkretnie być instancją interfejsu (*typeclass*)) `Floating`. Najprościej rzecz ujmując, argument musi być liczbą zmiennoprzecinkową. W Haskellu typy są instancjami klas. Klasy (konkretnie *typeclasses*) nie mają nic wspólnego z klasami z języków obiektowych. Są bardziej podobne do interfejsów, jednak mogą dostarczać pewne zachowanie. Do tego tematu powrócę, prezentując tworzenie własnych typów. Informację o typie interesującego nas obiektu możemy uzyskać z Haskell'owej wyszukiwarki Hoogle (https://www.haskell.org/hoogle/). Oprócz typu Hoogle pokaże nam, z jakiego modułu pochodzi funkcja, a także (po kilku kliknięciach) możemy dotrzeć do jej kodu źródłowego. Warto skorzystać z Hoogle, zanim zabierzemy się za rozwiązywanie jakiegoś problemu. Być może istnieje gotowe rozwiązanie w bibliotece standardowej.

OK, teraz nasza własna funkcja, nazwijmy ją `greet`. Uruchamiamy GHCi w trybie multiline i z wyświetlaniem typów wyrażeń.

Listing 6. Funkcja greet

```
Prelude> let greet name = -- jeżeli definiujemy funkcję w wielu
wierszach, używamy "let"
Prelude|       "Hello " ++ name  -- wyrażenie musi być wcięte
poza blok główny funkcji
greet :: [Char] -> [Char]
Prelude> greet name = "Hello " ++ name -- efekt ten sam co
wyżej
greet :: [Char] -> [Char]
Prelude> greet "Alice" -- wywołanie funkcji greet z argumentem
"Alice"
"Hello Alice"
it :: [Char]
Prelude> greet ['B','o','b']
"Hello Bob"
it :: [Char]
```

Przeanalizujemy teraz ten fragment. Zdefiniowaliśmy funkcję `greet` i wywołaliśmy ją dwa razy. Funkcję zdefiniowaliśmy dwukrotnie, żeby pokazać różnicę w standardowym przetwarzaniu REPL w GHCi a trybem wielowierszowym. Przy budowie bardziej złożonych funkcji wyrażenia wewnątrz ciała funkcji muszą być odpowiednio wyrównane. Identacja jest częścią składni Haskella (podobnie jak w Pythonie), zastępuje nawiasy klamrowe czy inne elementy tworzące bloki kodu. Generalna zasada indentacji w Haskellu jest następująca: kod będący częścią wyrażenia musi być wysunięty bardziej niż początek tego wyrażenia.

Zaraz po wprowadzeniu kodu zobaczyliśmy, jaki funkcja ma typ: `greet :: [Char] -> [Char]`, czyli jest to funkcja o nazwie `greet`, która przyjmuje jeden argument – listę znaków i zwraca listę znaków. W drugim wywołaniu zaprezentowałem jawnie przekazaną listę znaków, aby pokazać, że `['B','o','b']` to to samo co `"Bob"`, co bardzo ułatwia pracę z ciągami znaków (na marginesie typ `[Char]` ma synonim `String`). Zdefiniujmy kolejną funkcję:

Listing 7. Funkcja square

```
Prelude> square x = x * x
square :: Num a => a -> a
Prelude> square 5
25
it :: Num a => a
Prelude> square 5.5
30.25
it :: Fractional a => a
```

Nasza funkcja oblicza kwadrat dla podanej liczby – to oczywiste. Ale jakiej liczby? Typ funkcji `square :: Num a => a -> a` mówi, że `square` jest funkcją, która pobiera argument `a` i zwraca wartość typu `a`, z kolei `a` jest instancją klasy `Num`. Hierarchia typów liczbowych w Haskellu jest dosyć rozbudowana, a `Num` to liczby, które między innymi obsługują operację mnożenia (czyli wszystkie liczby). Podstawowe typy liczbowe to `Int`, `Integer`, `Float`, `Double`, `Ratio` oraz `Complex`. Wszystkie one są instancjami `Num`. `Num` ma jeszcze swoje podtypy i tak `Int`, `Integer` są instancjami `Integral`, a `Float` i `Double` są instancjami `Fractional`. Więcej informacji o typach zostanie przedstawione w dalszej części artykułu.

Przeanalizujmy te kilka linii kodu. Deklarując funkcję, nie podawaliśmy typu dla argumentu oraz wartości zwracanej przez funkcję. Dodatkowo okazało się, że nasza funkcja raz zwraca liczbę całkowitą, a w kolejnym wywołaniu liczbę zmiennoprzecinkową. Na pierwszy rzut oka wygląda, jakby Haskell był językiem dynamicznie typowanym. Oczywiście tak nie jest. Typy są znane na etapie kompilacji lub ewaluacji (w konsoli GHCi). Możliwe jest zdefiniowanie typów funkcji jawnie, ale o tym za chwilę. Jeżeli nie zrobimy tego jawnie, Haskell zrobi to za nas. Wywnioskuje typy na podstawie kodu, jaki napiszemy. Dodatkowo w przykładzie zaobserwowaliśmy różne zachowanie dla funkcji, gdy przekazywane były liczby różnych typów – polimorfizm. Te trzy właściwości sprawiają, że Haskell jest językiem bezpiecznym, elastycznym i nie nadmiarowym składniowo.

Teraz dla odmiany napiszemy jeszcze raz funkcję `square` z konkretnymi typami dla argumentu i wartości zwracanej.

Listing 8. Funkcja square jawnie typowana

```
Prelude> let square :: Int -> Int
Prelude|     square x = x * x
square :: Int -> Int
Prelude> square 5
Prelude> 25
it :: Int
Prelude> square 5.2 -- błąd
```

Tym razem sami nadaliśmy typ naszej funkcji. Wywołanie funkcji z argumentem o typie innym niż `Int` powoduje błąd.

Zanim przejdziemy dalej, przyjrzyjmy się kilku prostym funkcjom z biblioteki standardowej (tym razem pominę informację o typach).

Listing 9. Wywołania kilku funkcji z biblioteki standardowej

```
Prelude> max 3 2
3
Prelude> min 3 2
2
Prelude> negate 1
-1
Prelude> abs (negate 1)
1
Prelude> signum 123
1
Prelude> mod 8 3
2
Prelude> succ 10
11
Prelude> pred 10
9
Prelude> even 2
True
Prelude> odd 2
False
```

W Haskellu operatory to też funkcje. Wywołania:

Listing 10. Operatory jako funkcje

```
Prelude> 1 + 2 -- infix
3
Prelude> (+) 1 2 -- prefix
3
```

są równoważne. Domyślnie każda funkcja, której nazwa składa się ze znaków niealfanumerycznych, traktowana jest jako operator. W Haskellu są funkcje, których sposób wywołania jest taki jak operatorów w innych językach programowania. Funkcje prefixowe możemy wywołać w notacji „infix". Należy wtedy nazwę funkcji umieścić między znakami `(*backtick*). Na przykład dla funkcji obliczającej resztę z dzielenia całkowitego taki zapis jest bardziej czytelny.

Listing 11. Definicja funkcji mod jako operator oraz jej przykład użycia

```
Prelude> 8 `mod` 3
2
Prelude> (%) x y = mod x y
Prelude> 8 % 3
2
Prelude> isEven x = x % 2 == 0
Prelude> isEven 42
True
Prelude> isOdd x = not (isEven x)
Prelude> isOdd 42
False
```

FUNKCJE – SKŁADNIA

Jak dotąd budowaliśmy proste funkcje, składające się z pojedynczego wyrażenia. Zanim przejdziemy do najważniejszego i najpotężniejszego narzędzia – dopasowaniu do wzorca (ang. *pattern matching*) – przedstawię cztery elementy składni Haskella, które będą użyteczne podczas tworzenia funkcji:

- wyrażenie `if then else`
- wyrażenie `case .. of`
- wyrażenie `let .. in`
- słowo kluczowe `where`

Wyrażenie `if then else` to jedna z niewielu konstrukcji w Haskellu, która ma analogię w językach imperatywnych. Wyrażenie to, podkreślam: wyrażenie, a nie instrukcja, jest podobne do operatora ternarnego `?`, znanego chociażby z C++. Skoro jest to wyrażenie, to musi zawsze zwrócić wartość, w związku z tym część `else` jest obowiązkowa.

Listing 12. Prosta instrukcja warunkowa if

```
Prelude> let checkNumber x =
Prelude|       if x < 0 then
Prelude|         "Number is negative"
Prelude|       else
Prelude|         "Number is positive or equal zero"
Prelude> checkNumber 42
"Number is positive or equal zero"
Prelude> checkNumber (-42)
"Number is negative"
```

Aby stworzyć bardziej skomplikowany warunek, należy umieścić wyrażenie `if` jako podwyrażenie w części `if` lub `else` wyrażenia nadrzędnego z odpowiednią indentacją.

Listing 13. Złożona instrukcja warunkowa if

```
Prelude> let checkNumber x =
Prelude|       if x < 0 then
Prelude|         "Number is negative"
Prelude|       else
Prelude|         if x > 0 then
Prelude|           "Number is positive"
Prelude|         else
Prelude|           "Number is equal zero"
Prelude> checkNumber 42
"Number is positive"
Prelude> checkNumber (-42)
"Number is negative"
Prelude> checkNumber 0
"Number is equal zero
```

Kolejną konstrukcją sterującą jest wyrażenie `case .. of`. Jest to bardziej uogólniona wersja wyrażenia `if then else`, która dodatkowo pozwala na użycie dopasowania do wzorca. Składnia wyrażenia wygląda następująco:

Listing 14. Składnia wyrażenia case

```
case expression of pattern1 -> result1
                   pattern2 -> result2
                   pattern3 -> result3
                   ...
```

W miejscu `pattern` można użyć wartości lub wzorca (o wzorcach za chwilę). W miejscu `expression` możemy użyć wartości lub złożonego wyrażenia. Przepiszemy teraz funkcję `checkNumber` z poprzedniego przykładu z użyciem konstrukcji `case .. of`.

Listing 15. Użycie wyrażenia warunkowego case

```
Prelude> let checkNumber x =
  case signum x of 1  -> "Positive"
    -1 -> "Negative"
    0  -> "Zero"
Prelude> checkNumber 42
"Positive"
Prelude> checkNumber (-42)
"Negative"
Prelude> checkNumber 0
"Zero"
```

Kolejne konstrukcje wyrażenie `let .. in` i słowo kluczowe `where` umożliwiają przechowanie tymczasowego wyniku naszych operacji wewnątrz funkcji. Dzięki zastosowaniu `let .. in` i `where` unikamy powtarzania kodu. W konstrukcji `where` możliwe jest użycie dopasowania do wzorca.

Listing 16. Przykład użycia słowa kluczowego where w funkcji

```
Prelude> let sumOfSquares x y = a + b
  where a = x * x
    b = y * y
Prelude> sumOfSquares 2 3
13
```

Podobny efekt z użyciem `let .. in` (wyrażenie ma wartość taką jak wartość wyrażenia po słowie `in`.)

Listing 17. Przykład użycia wyrażenie let .. in w funkcji

```
Prelude> let sumOfSquares x y =
  let a = x * x
    b = y * y
  in  a + b
Prelude> sumOfSquares 4 5
41
```

FUNKCJE – PATTERN MATCHING

Pattern matching jest konstrukcją języka Haskell, która robi różnicę. W większości języków imperatywnych nie istnieje analogia dla konstrukcji *pattern matching* (wprowadzenie tej konstrukcji jest planowane w C# 7). Zrozumienie i sprytne stosowanie tej konstrukcji prowadzi do zwięzłego i eleganckiego kodu. Ogólnie *pattern matching* można opisać jako sekwencję dopasowań wartości do wzorców. Gdy Haskell znajdzie pasujące dopasowanie, następuje wiązanie wartości. *Pattern matching* jest jednocześnie konstrukcją sterującą, jak i czymś, co pozwala zawrzeć logikę w naszych funkcjach. Najlepiej pokażą to przykłady.

Na początek w sesji interaktywnej utworzymy trzy wartości (liczbę, listę i krotkę), które później będziemy przekazywać do stworzonych funkcji wykorzystujących *pattern matching*.

Listing 18. Utworzenie trzech wartości – liczby, listy oraz krotki

```
Prelude> x = 42            -- liczba
Prelude> xs = [1,2,3,4,5]  -- lista
Prelude> t = (42, 43, 44)  -- krotka 3 elementowa
```

Proste dopasowanie:

Listing 19. Przykład prostej funkcji wykorzystującej pattern matching

```
Prelude> let isThis42 1 = "1 is not 42"
Prelude|     isThis42 2 = "2 is not 42"
Prelude|     isThis42 3 = "3 is not 42"
Prelude|     isThis42 42 = "Yeah. 42!"
Prelude|     isThis42 n = "I don't know what it is"
Prelude> isThis42 1
"1 is not 42"
Prelude> isThis42 4
"I don't know what it is"
Prelude> isThis42 x -- x = 42
"Yeah. 42!"
```

Zdefiniowaliśmy funkcję (zdecydowanie bezużyteczną) wykorzystującą *Pattern matching*. Pierwsze, co rzuca się w oczy, to wielokrotne definicje funkcji `isThis42`. W rzeczywistości jest to jedna, wielowierszowa definicja funkcji. Zauważmy, że dla pierwszych przypadków na sztywno podaliśmy wartości argumentu funkcji

i odpowiadające im wartości funkcji. Funkcja do tego momentu została zdefiniowana poprzez wartościowanie. W języku imperatywnym ten kod składałby się z serii czterech testów `if` (dla 1, 2, 3 i 42), a po nich zwrócona byłaby wartość, która nie spełnia żadnego z tych testów. Ewentualnie można to porównać do instrukcji `switch` z C++ czy C#, z klauzulą `default` na końcu dla pozostałych przypadków. Zaprezentowany przykład nie jest zbyt wyszukany – liczby są prostymi typami danych. Ciekawiej będzie, gdy użyjemy rekurencji. Wtedy będziemy mogli naprawdę coś policzyć. Teraz klasyka – silnia.

Listing 20. Funkcja factorial

```
Prelude> let factorial :: Int -> Int
Prelude|     factorial 0 = 1
Prelude|     factorial n = n * factorial (n - 1)
Prelude> factorial 5
120
```

Definicja funkcji `factorial` jest bardzo podobna do jej matematycznej postaci. Funkcja jest zdefiniowana rekurencyjnie – w Haskellu występują pętle `for` czy `while` znane z innych języków. Ucząc się języka Haskell, musimy się przestawić na projektowanie swoich funkcji w sposób rekurencyjny – każdą pętlę można zastąpić wywołaniem rekurencyjnym. Większość funkcji z biblioteki standardowej jest zaimplementowana rekurencyjnie. Innym sposobem na cykliczne wykonywanie operacji jest mapowanie list, ale o tym później.

Kolejny klasyczny przykład rekurencyjnej definicji funkcji – `quicksort`. Wykorzystana tu zostanie składnia list składanych (ang. *list comprehension*), która będzie omówiona w sekcji poświęconej listom.

Listing 21. Funkcja quicksort

```
Prelude> let quicksort [] = []
Prelude|     quicksort (x:xs) =
Prelude|       let list1 = quicksort [a | a <- xs, a <= x]
Prelude|           list2 = quicksort [a | a <- xs, a > x]
Prelude|       in  list1 ++ [x] ++ list2
Prelude> quicksort [3,1,5,2,4]
[1,2,3,4,5]
```

W kolejnym przykładzie zastosujemy *pattern matching* w funkcji przyjmującej jako argument listę. Zobaczymy, że dopasowanie może być zbudowane w taki sposób, aby potrafiło zareagować na wartości o określonej strukturze, na przykład pustą listę. Przyda nam się podstawowa wiedza na temat budowy list. Listy budowane są za pomocą operatora `:` (*cons operator*). Dodaje on element na początku list, nie zmieniając oryginalnej listy, a tworząc nową listę o jeden element większą.

Listing 22. Dodawanie elementów do listy

```
Prelude> 1 : [2,3]
[1,2,3]
Prelude> 1 : 2 : 3 : 4 : 5 : []
[1,2,3,4,5]
```

Ok, po tym wstępie funkcja, która zwraca maksymalnie trzy elementy z listy.

Listing 23. Funkcja getAtMost3

```
Prelude> let getAtMost3 :: [Int] -> [Int]
Prelude|     getAtMost3 [] = []
Prelude|     getAtMost3 (x:y:z:xs) = [x,y,z]
Prelude|     getAtMost3 (x:y:xs) = [x,y]
Prelude|     getAtMost3 (x:xs) = [x]
Prelude> getAtMost3 xs
[1,2,3]
Prelude> getAtMost3 []
[]
Prelude> getAtMost3 [5]
[5]
Prelude> getAtMost3 [7,8]
[7,8]
```

Kilka słów komentarza do funkcji `getAtMost3`. W pierwszym wierszu jawnie podaliśmy typ funkcji (funkcja przyjmuje i zwraca listę wartości `Int`). Taki zabieg może być pożądany z dwóch względów. Po pierwsze, jawnie podany typ jest częścią definicji funkcji i służy jako udokumentowanie jej działania. Po drugie, Haskell nie będzie wnioskował typu dla niej. Na podstawie struktury przekazanej listy funkcja zwraca odpowiednią wartość. Struktura listy w tym przypadku oznacza to, czy jest ona pusta – `[]`, czy ma co najmniej 3 elementy – `(x:y:z:xs)`, co najmniej 2 `(x:y:xs)` lub co najmniej jeden element – `(x:xs)`. Jeżeli kolejność dopasowań byłaby inna, funkcja nie działałaby poprawnie. Na przykład, przypadek, gdy lista ma 3 lub więcej elementów, jest bardziej ogólny niż przypadek dla listy z co najmniej dwoma lub jednym elementem. Argument przekazany do funkcji jest sprawdzany z kolejnymi dopasowaniami od góry do dołu, aż dopasowanie będzie spełnione. *Pattern matching* jest po pierwsze składnią sterującą, a po drugie – dokonuje wiązania wartości z nazwami, które możemy wykorzystać w zwracanym wyniku.

Mając już pewną wiedzę o dopasowaniach, a także o funkcjach rekurencyjnych, zdefiniujemy własną funkcję `len`, która będzie zwracała długość listy (czyli dokładnie to samo co funkcja `length` z Prelude). Wiele funkcji rekurencyjnych tworzy się według podobnego schematu: po pierwsze, należy zdefiniować warunek bazowy – dla funkcji `len` będzie to fakt, że pusta lista ma długość zero. Po drugie, część rekurencyjna – w tym przypadku to długość listy jest równa długości listy bez pierwszego element plus 1. Teraz zapiszmy to w kodzie:

Listing 24. Rekurencyjna funkcja len

```
Prelude> let len [] = 0
Prelude|     len (x:xs) = 1 + len xs
Prelude> len [1,2,3]
3
Prelude> len "Hello"
5
```

Na przykładzie listy zauważamy, że *Pattern matching* to dekonstrukcja obiektów w celu wydobycia interesujących nas wartości. Ta dekonstrukcja jest przeciwieństwem tworzenia obiektu. Jeżeli potrafimy stworzyć obiekt, to będziemy również zdolni napisać dla niego odpowiednie dopasowanie. Kolejne funkcje będą operowały na krotce trzyelementowej. W pierwszej sekcji użyłem funkcji `fst` i `snd`, które pobierały pierwszy i drugi element krotki dwuelementowej. Teraz stworzymy funkcje `fst`, `snd` i `trd` (pierwszy, drugi i trzeci element) dla krotki trzyelementowej.

Listing 25. Funkcje fst, snd i trd dla krotki trzyelementowej

```
Prelude> let fst (x,_,_) = x
Prelude> let snd (_,x,_) = x
Prelude> let trd (_,_,x) = x
Prelude> fst t -- t = (42, 43, 44)
42
Prelude> snd t
43
Prelude> trd t
44
```

Zwróćmy uwagę na pewną prawidłowość (zarówno dla funkcji z listą z poprzedniego przykładu, jak i ostatnich funkcji dla krotki). Wzorzec dopasowania jest strukturalnie bardzo podobny do obiektu. Przerobienie kilku przykładów sprawia, że pisanie dopasowań staje się naturalne. Uwaga do funkcji `fst`, `snd` i `trd` – znak podkreślenia w definicji tych funkcji oznacza, że nie interesuje nas wartość znajdująca się w jego miejscu. Wartość ta nie jest wiązana z nazwą.

Haskell umożliwia łączenie dopasowań z warunkami logicznymi, nazywanymi strażnikami (ang. *guards*). Strażników używamy, gdy mamy potrzebę zareagowania na wartość wyodrębnioną z dopasowania. Składniowo strażnicy przypominają instrukcję `switch` z C++ cz C#. Jako przykład napiszemy funkcję, która sprawdza, czy pierwsza liczba na liście równa się zero lub czy jest dodatnią liczbą parzystą lub dodatnią nieparzystą. Przykład jest mocno naciągany, ale pozwoli zaprezentować wymienione wcześniej konstrukcje.

Listing 26. Przykład wykorzystania strażników

```
Prelude> let guardFun [] = "empty list"
Prelude|     guardFun (x:_)
Prelude|       | x == 0 = "zero"
Prelude|       | x > 0 && even x = "even"
Prelude|       | x > 0 && odd x = "odd"
Prelude|       | otherwise = "negative"
Prelude> guardFun []
"empty list"
Prelude> guardFun [0,4,2]
"zero"
Prelude> guardFun [1,4,2]
"odd"
Prelude> guardFun [2,4,2]
"even"
Prelude> guardFun [-1,4,2]
"negative"
```

Składnia strażników jest następująca: po dopasowaniu nie ma znaku „=". Jest on przeniesiony za każdy z warunków. Warunki rozpoczyna się w nowym wierszu od wcięcia i znaku „|", po którym jest test logiczny i wartość dla tego testu. Z reguły na końcu umieszcza się warunek `otherwise`, który odpowiada wartości `True` – jest to sekcja dla pozostałych przypadków, które nie spełniają żadnego z testów powyżej.

FUNKCJE WYŻSZEGO RZĘDU

Funkcje są takimi samymi wartościami jak liczby, krotki czy listy. Podobnie jak one mogą być przekazywane jako argumenty funkcji i być zwracane przez funkcję. Funkcją, która zwraca lub przyjmuje inną funkcję jako parametr, nazywamy funkcją wyższego rzędu (ang. *higher order function*). Jest to koncepcja znana z języków dynamicznych takich jak JavaScript, jak i języków statycznie typowanych, na przykład w języku C# – rozszerzenia Linq. Pokażemy teraz przykład funkcji wyższego rzędu. Będzie to prosta funkcja, która będzie przyjmować liczbę oraz funkcję, która wykona na niej jakąś operację. Funkcja zwraca `True`, jeżeli wynik tego obliczenia będzie większy lub równy zero, `False` – w przeciwnym przypadku.

Listing 27. Przykład funkcji wyższego rzędu

```
Prelude> let isComputationPositiveOrZero :: Int -> (Int  ->
Int) -> Bool
Prelude|     isComputationPositiveOrZero x f = (f x) >= 0
Prelude> let minus10 x = x - 10
Prelude> isComputationPositiveOrZero 11 minus10
True
Prelude> isComputationPositiveOrZero 10 minus10
True
Prelude> isComputationPositiveOrZero 9 minus10
False
```

Funkcja `isComputationPositiveOrZero` wykonuje funkcję `f` z parametrem x, a następnie sprawdza, czy wynik jest większy lub równy zero. W przykładzie zdefiniowaliśmy funkcję `minus10`, z której korzystaliśmy w wywołaniu `isComputationPositiveOrZero`. Aby nie tworzyć wielu niewielkich funkcji z nazwą, możemy skorzystać z funkcji anonimowej (ang. *lambda function*).

Listing 28. Wywołania funkcji isComputationPositiveOrZero z użyciem funkcji anonimowych

```
Prelude> isComputationPositiveOrZero 11 (\x -> x - 10)
True
Prelude> isComputationPositiveOrZero 11 (\x -> x - 11)
True
Prelude> isComputationPositiveOrZero 11 (\x -> x - 12)
False
```

Funkcje anonimowe są często wykorzystywane w funkcjach operujących na listach, na przykład: `map`, `filter`, `foldl`.

Co się dzieje podczas wywołania funkcji z wieloma argumentami? Sprawdźmy to na przykładzie trzyargumentowej funkcji add.

Listing 29. Funkcja add

```
Prelude> let add x y z = x + y + z
Prelude> add 1 2 3
6
Prelude> :t add
add :: Num a => a -> a -> a -> a
```

Jak na razie nie ma żadnego zaskoczenia, add jest funkcją pobierającą 3 liczby i zwracającą liczbę. Co się stanie, gdy nie przekażemy wszystkich parametrów funkcji? Haskell utworzy nową funkcję z parametrami, których nie przekazaliśmy w wywołaniu. Haskell wywołuje funkcje w następujący sposób:

Listing 30. Wywołanie funkcji add

```
Prelude> ((add 1) 2) 3
6
```

Należy tutaj wprowadzić dwa terminy – *curried function* oraz *partial application*. *Curried function* to funkcja, która powstała z niepełnego wywołania innej funkcji. Jeżeli do funkcji trzyargumentowej przekażemy tylko pierwszy argument, to powstanie funkcja dwuargumentowa, która niejako zapamięta wartość pierwszego parametru. To jest właśnie wywołanie częściowe, czyli *partial application*.

Wróćmy do naszej funkcji add z trzema parametrami – zastosujemy wywoływanie częściowe do utworzenia nowych funkcji.

Listing 31. Częściowe wywołanie funkcji add

```
Prelude> let add10 = add 10
Prelude> :t add10
add10 :: Num a => a -> a -> a
Prelude> add10 20 30
60
Prelude> let add10and20 = add10 20
Prelude> :t add10and20
add10and20 :: Num a => a -> a
Prelude> add10and20 70
100
```

Po zastosowaniu wywołania częściowego z parametrem 10 utworzyliśmy funkcję `add10`. Z typu funkcji wynika, że przyjmuje ona już tylko dwa parametry. Przy wywołaniu `add10` parametry te są

dodawane do wartości 10. Następnie utworzyliśmy funkcję `add-10and20`, wykorzystując częściowo wywołaną funkcję `add10`, która ma już tylko jeden parametr, a jego wartość jest powiększana kolejno o 10 i 20. Dla przypomnienia operatory to też funkcje, również dla nich możemy stosować *partial application*. Bardzo częstą praktyką jest wykorzystywanie częściowo aplikowanych operatorów przy filtrowaniu list.

Częściowo aplikowany operator pokażemy na przykładzie funkcji `twice`.

Listing 32. Częściowo wywoływany operator w funkcji twice

```
Prelude> let twice x = x * 2 -- standardowo
Prelude> twice 21
42
Prelude> let twice = (2*)    -- z operatorem '*'
Prelude> twice 21
42
```

Funkcje w Haskellu mają jeszcze jedną ciekawą cechę (wynikającą z tego, że są funkcjami wyższego rzędu) – można je łączyć w jedną całość (tworzyć nową funkcję), która będzie połączeniem tych funkcji. Mechanizm ten nosi nazwę *function composition*. Funkcje łączy się operatorem „`.`" (kropka). Ważne, by kolejne funkcje przyjmowały jeden argument i był to argument o typie takim jak typ wartości zwracanej z poprzedniej funkcji.

Listing 33. Przykład złożenia funkcji

```
Prelude> let add1 x = x + 1
Prelude> let add2 x = x + 2
Prelude> let add3 x = x + 3
Prelude> let add = add1 . add2 . add3
Prelude> add 4
10
```

Kolejny przykład *function composition* – odwrócenie kolejności słów w łańcuchu znaków:

Listing 34. Funkcja reverseWords

```
Prelude> let reverseWords = unwords . reverse . words
Prelude> :t reverseWords
reverseWords :: String -> String
Prelude> reverseWords "Hello from Haskell"
"Haskell from Hello"
```

Ostatnim elementem, który chciałbym przedstawić, związanym z funkcjami i często używanym w praktyce, jest operator „`$`" (znak dolara). Operator ten ma następującą definicję:

Listing 35. Operator $

```
Prelude> f $ x = f x
```

Czyli po prostu wywołanie funkcji. Gdzie jest haczyk? Operator pozwala na pominięcie nawiasów w momencie, gdy przekazujemy wynik wywołania funkcji jako argument do kolejnej funkcji.

Listing 36. Przykład użycia operatora $

```
Prelude> sqrt (max 2 4)
2.0
Prelude> sqrt $ max 2 4
2.0
```

TYPY

Haskell ma unikalny system typów. Język jest silnie typowany, a typy w tworzonych przez nas funkcjach (o ile nie podamy typów jawnie) są wnioskowane przez kompilator (algorytm wnioskowania typów Hindleya–Milnera). Wnioskowanie typów (ang. *type inference*) po pierwsze skraca kod – deklaracja typu funkcji jest opcjonalna. Po drugie, Haskell stara się dobrać możliwie najbardziej ogólny typ dla tworzonych przez nas funkcji. Przeciwieństwem tego jest podanie typu jawnie, co ma dwie zalety: jest swego rodzaju dokumentacją, a w niektórych przypadkach kod, w którym użyjemy typu mniej ogólnego, może być szybszy. Jednak zalecaną praktyką jest pozostawienie określenia typu kompilatorowi. Konsekwencją faktu dopasowywania najbardziej ogólnego typu jest polimorficzne zachowanie funkcji. Jeżeli napiszemy funkcję, która dodaje dwie liczby, informację, jaką kompilator otrzymuje, to, że ma dopasować typ do obiektów do takiego, który obsługuje operator dodawania. Generalnie można powiedzieć, że ten typ to liczby – liczby mogą być dodawane. Nie jest istotne, czy są to liczby całkowite, czy zmiennoprzecinkowe.

Listing 37. Funkcja add wraz z jej typem

```
Prelude> let add x y = x + y
Prelude> :t add
add :: Num a => a -> a -> a
```

Należy wspomnieć, czym jest a w typie funkcji add. Nie jest to konkretny typ jak `Int` czy `Bool`. Aby pokazać, czym jest owe a, spójrzmy na kolejny przykład: funkcja `head` z biblioteki standardowej, która zwraca pierwszy element z listy.

Listing 38. Funkcja head wraz z jej typem

```
Prelude> head [1,2,3]
1
Prelude> :t head
head :: [a] -> a
Prelude> head [1,2,3]
1
Prelude> head "hello"
'h'
Prelude> head [[1,2,3], [5,6], [7]]
[1,2,3]
Prelude> head [(1, "Bob"), (2, "John")]
(1,"Bob")
```

Funkcja `head` przyjmuje listę elementów o typie a i zwraca wartość o typie a. Oznacza to, że a (*type variable*) może być dowolnego typu, co ma sens, ponieważ pobranie pierwszego elementu z listy nie jest uzależnione od tego, jakie elementy ta lista przechowuje. Ta funkcja jest polimorficzna, ponieważ może przyjmować listy o elementach dowolnego typu. Typy kolejnych funkcji – `areEqual` oraz `IsGreater` będą podobne do typu funkcji add:

Listing 39. Funkcje areEqual i isGreater

```
Prelude> let areEqual x y = x == y
Prelude> :t areEqual
areEqual :: Eq a => a -> a -> Bool
Prelude> let isGreater x y = x > y
Prelude> :t isGreater
isGreater :: Ord a => a -> a -> Bool
```

Typ `areEqual` odczytujemy następująco: jest to funkcja, która przyjmuje dwa argumenty typu a, który jest instancją typu

Eq i zwraca wartość typu Bool. Fragment znajdujący się przed symbolem => mówi, jakiego typu jest a (jest to *type constraint*). W trzech powyższych funkcjach mamy do czynienia z pewnym typem uogólnionym (Num jako liczby, Eq jako obiekty, które możemy sprawdzić, czy są sobie równe, Ord jako obiekty, które, mając ich parę, możemy określić, który z tej dwójki jest większy, a który mniejszy). Num i Eq to przykłady klas typów (*type classes*). W Haskellu klasy typów i polimorfizm nie mają nic wspólnego ze znaczeniem tych pojęć w językach obiektowych. Klasy typów można porównać do interfejsów, a funkcje polimorficzne do typów generycznych. Klasy typów określają, jakie operacje ma mieć typ, który jest ich instancją, a także dostarczają tak zwaną minimalną kompletną definicję tych operacji (ang. *minimal complete definition*). Ma to znaczenie przy tworzeniu własnych typów, które mają być instancjami konkretnych klas typów. Wtedy mamy dwie opcje: przyjąć domyślną definicję operacji z klasy typu lub napisać własne wersje tych operacji. Jako podsumowanie tego fragmentu, i przed przejściem do tworzenia własnych typów, wymienię najważniejsze klasy typów w Haskellu oraz niektóre operacje, jakie definiują:

- Eq – (==), (/=)
- Ord – compare, (<=)
- Show – show
- Read – readPrec
- Enum – toEnum, fromEnum
- Bounded – minBound, maxBound
- Num – (+), (*), abs, signum, fromInteger
- Integral – toInteger (do tego typu należą: Int oraz Integer)
- Floating – pi, exp, log, sin, cos (do tego typu należą: Float oraz Double)
- Monad

Po wprowadzeniu, dotyczącym typów z biblioteki standardowej, nadszedł moment na zaprezentowanie sposobu tworzenia typów użytkownika. W Haskellu istnieją następujące rodzaje i odmiany takich typów:

- synonimy
- *algebraic data types* – typy złożone z innych typów
- rekordy – typy złożone, tworzone za pomocą innej składni

Haskell umożliwia również tworzenie własnych klas typów, które nadają innym typom określone zachowanie.

Synonimy to najprostsze typy użytkownika. Jest to po prostu nowa nazwa dla innego typu. Można je stosować zamiennie z typami, do jakich się odnoszą. Chyba najczęściej używanym synonimem jest String:

Listing 40. Synonim typu [Char]

```
type String = [Char]
```

Wszędzie, gdzie napotkamy się na typ String, możemy użyć typu [Char] i na odwrót. Synonim tworzy się za pomocą słowa kluczowego type, następnie rozpoczynająca się z dużej litery nazwa synonimu i po znaku równości typ, jaki synonim ma zastępować. Używanie synonimów ma przede wszystkim poprawić czytelność kodu, jednak dla kompilatora to, czy użyjemy synonimu, czy oryginalnego typu, nie ma znaczenia.

Czas na coś ciekawszego – typy złożone. Przykładem typu złożonego (ang. *algebraic data type*) jest wbudowany typ Maybe. Jest to bardzo ciekawy i często spotykany typ. Pozwala na budowanie funkcji, w których zakładamy scenariusz pozytywny – wtedy wartość zwracana jest opakowana w konstruktor Just, oraz scenariusz negatywny, gdzie zamiast generowania błędu, funkcja zwraca Nothing. Do omówienia tworzenia typów złożonych posłużę się innym przykładem, jednak wcześniej kilka słów o typie Maybe. Oto jego definicja:

Listing 41. Definicja typu Maybe

```
data Maybe a = Nothing | Just a
  deriving (Eq, Ord)
```

Wróćmy do funkcji head. Oto jej definicja:

Listing 42. Implementacja funkcji head

```
head :: [a] -> a
head (x:_) = x
head [] = error "empty list"
```

Oczywiście pusta lista nie ma pierwszego elementu. Funkcja head dla takiego przypadku zwraca błąd, który przerywa program. Moglibyśmy się pokusić o napisanie własnej funkcji head, która byłaby na to odporna, z wykorzystaniem typu Maybe.

Listing 43. Kolejna implementacja funkcji head z użyciem Maybe

```
Prelude> let head :: [a] -> Maybe a
Prelude|     head (x:_) = Just x
Prelude|     head [] = Nothing
Prelude> head [1,2,3]
Just 1
Prelude> head []
Nothing
```

Ta wersja funkcji head nie generuje błędu dla pustej listy. Zamiast tego zwraca Nothing, które odpowiada tej sytuacji i może zostać odpowiednio obsłużone w dalszym kodzie. Zapamiętajmy sobie typ Maybe – jest on bardzo przydatny i często stosowany. Teraz stwórzmy własny typ – Person.

Listing 44. Typ Person

```
Prelude> data Person = Person Int String String |
UnregisteredPerson String String
```

Stworzyliśmy nowy typ Person. Składnia jest dość dziwna (takie było moje pierwsze wrażenie), ale zwięzła. Nie podajemy nazwy pól i odpowiadających im typów, tylko same typy. Musimy pamiętać o naszych intencjach podczas tworzenia typu, czyli na której pozycji będziemy przechowywać konkretną informację. Nowy typ tworzymy za pomocą słowa kluczowego data, następnie nazwa typu (z dużej litery), potem (po znaku równości) nazwa konstruktora oraz lista typów pól, jakie miał będzie obiekt utworzony tym konstruktorem, oraz kolejne opcjonalne konstruktory oddzielone znakiem „|". Nazwa konstruktora może być taka sama jak nazwa typu, co jest częstą praktyką. Liczba konstruktorów jest dowolna, tak samo jak lista wartości przyjmowanych przez konstruktor. Typy algebraiczne można porównać do struktur z imperatywnych języków programowania jak C. Różnica jest taka, że typ mający wiele konstruktorów może przechowywać różne dane – jakie one będą, zależy od tego, którym z konstruktorów obiekt został stworzony. Konstruktory typów algebraicznych

nie mają nic wspólnego z konstruktorami w językach obiektowych. Nie mają kodu. Służą do stworzenia konkretnej wersji typu, łącząc swoją nazwę z listą typów. Tworząc typ `Person`, chciałem odwzorować dwie sytuacje: obiekt osoby zarejestrowanej, która posiada numer identyfikacyjny oraz imię i nazwisko, oraz osoby niezarejestrowanej, bez numeru identyfikacyjnego. Utworzymy obiekty tego typu:

Listing 45. Instancje typu Person

```
Prelude> let person1 = Person 123 "Bob" "Jones"
Prelude> let person2 = UnregisteredPerson "John" "Smith"
```

Jeżeli spróbujemy wyświetlić nasz obiekt w konsoli, niestety otrzymamy błąd. Nasz typ nie ma reprezentacji w postaci łańcucha znaków, zatem komenda:

Listing 46. Próba wyświetlenia wartości person1 w GHCi

```
Prelude> person1
błąd...
```

wygeneruje błąd, mówiący mniej więcej tyle, że `person1` nie jest instancją Show, przez co nie może być przekształcone na ciąg znaków. Możemy to zmienić i zadeklarować typ `Person` w taki sposób, żeby od razu był instancją Show.

Listing 47. Nowa wersja typu Person

```
Prelude> data Person = Person Int String String |
UnregisteredPerson String String deriving (Eq, Show)
```

Zwróćmy uwagę na słowo kluczowe `deriving`. Po nim, w nawiasach, wymieniamy listę klas typów. Od teraz obiekty typu `Person` będą jednocześnie zachowywały się jak obiekty `Eq` i `Show`. Co więcej, ta składnia powoduje, że `Person` odziedziczył domyślne implementacje metod z tych klas typów. Starałem się uniknąć słowa „dziedziczenie", ponieważ, znowu, nie ma to nic wspólnego z dziedziczeniem z języków obiektowych. Zadeklarowanie w ten sposób typu `Person` nadaje mu cechy `Eq` i `Show`. Od tej chwili możemy wykonać następujące operacje:

Listing 48. Operacje na wartościach nowego typu Person

```
Prelude> person1
Person 123 "Bob" "Jones"
Prelude> person2
UnregisteredPerson "John" "Smith"
Prelude> person1 == person2
False
Prelude> person1 == person1
True
```

Mamy nowy typ. Możemy tworzyć obiekty i przechowywać informacje o osobach. Spróbujmy zrobić coś więcej – napisać funkcję operującą na tym typie. Napiszemy trzy funkcje: `isPersonRegistered` – sprawdzimy, czy osoba ma numer identyfikacyjny, `getPersonId` – zwrócimy identyfikator osoby zarejestrowanej oraz `getPersonFullName` – zwrócimy imię i nazwisko osoby. W wymienionych funkcjach skorzystamy z mechanizmu *pattern matching*, który zastosujemy względem konstruktorów typu `Person`.

Listing 49. Funkcja isPersonRegistered z argumentem typu Person

```
Prelude> let isPersonRegistered (Person _ _ _) = True
Prelude|     isPersonRegistered _ = False
Prelude> isPersonRegistered person1
True
Prelude> isPersonRegistered person2
False
```

Słowo komentarza do funkcji `isPersonRegistered`. `True` zwracamy dla osoby, która posiada numer identyfikacyjny, czyli takiej, która była utworzona za pomocą pierwszego konstruktora. Logika funkcji sprowadza się do rozpoznania, jakim konstruktorem był utworzony przekazany do funkcji obiekt. Nie interesują nas wartości, jakie znajdują się wewnątrz obiektu, dlatego ignorujemy je, stąd znak „_". Podobnie zdefiniujemy pozostałe funkcje:

Listing 50. Kolejne funkcje dla typu Person

```
Prelude> let getPersonId = (Person id _ _) = Just id
Prelude|     getPersonId _ = Nothing
Prelude> getPersonId person1
Just 123
Prelude> getPersonId person2
Nothing
Prelude>
Prelude> let getPersonFullName (Person _ first last) = first ++
" " ++ last
Prelude|     getPersonFullName (UnregisteredPerson first last)
= first ++ " " ++ last
Prelude> getPersonFullName person1
"Bob Jones"
Prelude> getPersonFullName person2
"John Smith"
```

Przedstawione funkcje wykorzystują *pattern matching* do dekompozycji obiektu typu `Person`. Dzięki temu możemy wyodrębnić interesujące nas informacje, jakie zawiera dany obiekt. Aby efektywnie korzystać z tego mechanizmu, musimy jedynie znać strukturę tego obiektu. To pozwala na budowanie logiki funkcji w oparciu o wartości konkretnych pól lub jakim konstruktorem obiekt został utworzony. Składnia tworzenia nowych typów jest bardzo oszczędna. Wadą takiej formy zapisu jest to, że po jakimś czasie możemy nie pamiętać, które pole odpowiadało za imię, a które za nazwisko (na przykładzie typu `Person`). W definicji brakuje identyfikatorów, są tylko typy. Haskell oferuje jednak inną formę zapisu definicji typu. Typ `Person` (w wersji skróconej) możemy zdefiniować w następujący sposób:

Listing 51. Typ Person jako rekord

```
Prelude> data Person = Person {
Prelude|   firstName :: String,
Prelude|   lastName :: String }
Prelude> let person = Person { firstName = "Bob", lastName =
"Jones" }
Prelude> firstName person
"Bob"
Prelude> lastName person
"Jones"
```

Typ utworzony w ten sposób to rekord. Rekordy mają także swoją składnię tworzenia obiektów. Wartości są przypisywane do nazwanych pól. Utworzenie typu rekordu powoduje automatyczne utworzenie funkcji o nazwach takich jak pola rekordu (w przykładzie to: `firstName` oraz `lastName`). Mimo że definicja jest wygodna, czytelna i sama w sobie dokumentuje kod, to stosowanie rekordów nie jest zalecane. Największą wadą rekordów są two-

rzone automatycznie funkcje do pobierania wartości z pól – nie jest możliwe zdefiniowanie dwóch typów rekordów z polami o tej samej nazwie.

Kolejną kategorią typów, jaką oferuje Haskell, są klasy typów (ang. *type classes*). Jest to konstrukcja, która leży u podstaw polimorficznego zachowania funkcji, które zaprezentowałem wcześniej na przykładzie `Num`. O klasach typów wspominałem podczas omawiania funkcji, teraz pokażę to zagadnienie w kontekście tworzenia własnych typów. Przypomnijmy sobie końcowy fragment definicji typu `Person` – `deriving (Eq, Show)`. Ten fragment kodu sprawia, że `Person` staje się instancją `Eq` i `Show` oraz przyjmuje ich domyślne zachowanie (funkcje). Dla większości nowych typów taka konstrukcja jest wystarczająca, o ile nie chcemy w jakiś nietypowy sposób nadpisać zachowania, które dostarczają te klasy typów. Jeżeli chodzi o `Eq`, to sprawa wydaje się oczywista, domyślna minimalna implementacja wystarczy, aby porównać dwa obiekty nowego typu pod kątem równości (sprawdzenie, czy każde pole obiektu pierwszego jest równe odpowiedniemu polu z obiektu drugiego). Klasa typu `Eq` jest zdefiniowana następująco:

Listing 52. Definicja typu Eq

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
  x == y = not (x /= y)
  x /= y = not (x == y)
```

Drugą klasą typu, która dodaliśmy do definicji typu `Person`, jest `Show`. `Show` odpowiada za reprezentację obiektu w postaci łańcucha znaków. Tutaj moglibyśmy się pokusić o dodanie innego zachowania niż domyślne. Typ `Show` ma funkcję `show`, której domyślne działanie będziemy chcieli zmienić, wtedy definicja `Person` będzie wyglądała tak:

Listing 53. Jawna implementacja Show dla typu Person

```
Prelude> data Person = Person Int String String |
UnregisteredPerson String String deriving (Eq)
Prelude> instance Show Person where
Prelude|   show (Person _ firstName lastName) = firstName ++ "
" ++ lastName
Prelude|   show (UnregisteredPerson firstName lastName) =
firstName ++ " " ++ lastName
Prelude> person1
Bob Jones  -- wcześniej: Person 123 "Bob" "Jones"
Prelude> person2
John Smith -- wcześniej: UnregisteredPerson "John" "Smith"
```

Składnia `instance <type> <type class> where` rejestruje dany typ jako instancję klasy typu oraz nadaje nową implementację funkcji specyficznych dla tej klasy typu. W ten sposób możemy przeciążyć domyślne zachowanie, jakie otrzymujemy, korzystając z konstrukcji ze słowem kluczowym `deriving`.

LISTY

Listy to jeden z najważniejszych i najczęściej spotykanych typów w języku Haskell. W tej sekcji omówię budowę list, najważniejsze funkcje operujące na listach oraz listy składane. Sprawne posługiwanie się listami w Haskellu jest umiejętnością fundamentalną. Obiekty list pojawiały się w poprzednich przykładach, jednak tym razem przyjrzymy się im dokładniej. Na początek kilka prostych przykładów tworzenia list za pomocą literałów:

Listing 54. Tworzenie list różnych typów

```
Prelude> [] -- lista pusta
[]
Prelude> :t []
[] :: [t]
Prelude> x = [1,2,3] -- lista liczb (obiektów Num)
Prelude> :t x
x :: Num t => [t]
Prelude> s = "hello" -- list znaków
Prelude> :t s
s :: [Char]
Prelude> s = ["hello", "world"] -- lista list
Prelude> :t s
s :: [[Char]]
```

W odróżnieniu od tablic czy list w językach dynamicznych wartości listy muszą być tego samego typu. Typ ten jest oczywiście dowolny. Może to być liczba, znak, krotka, typ własny, listy list i dowolna ich kombinacja. Dwie najbardziej elementarne operacje na listach to: dodawanie elementu na początku (operator `cons`) oraz konkatenacja list. Wiemy już, że wartości w Haskellu są niezmienne, dlatego po każdej z tych operacji tworzona jest nowa lista, a oryginalna lista pozostaje niezmieniona. Kilka prostych operacji na listach:

Listing 55. Podstawowe operacje na listach

```
Prelude> xs = [1,2,3]
Prelude> ys = 0 : xs -- operator cons
Prelude> xs
[1,2,3]
Prelude> ys
[0,1,2,3]
Prelude> [4,2] == 4 : 2 : [] -- literał w nawiasach
kwadratowych to syntactic sugar dla operatora (:)
True
Prelude> [1,2,3] ++ [4,5] -- konkatenacja list
[1,2,3,4,5]
Prelude> [1,2,3] !! 0 -- indeksowanie
1
Prelude> [1,2,3] !! 1
2
Prelude> [1,2,3] == [1,2,2] -- porównywanie (porównywane są
elementy na odpowiednich indeksach)
True
Prelude> [1,2,3] > [1,2,2]
True
Prelude> [3] > [2,3,4]
True
```

Do utworzenia list można użyć zakresu. W ten sposób można tworzyć listy nieskończone. Jest to potężny mechanizm języka, który można wykorzystać w skomplikowanych algorytmach. Oczywiście listy nie są tak naprawdę nieskończone. Haskell jest leniwie wartościowany, więc dopóki nie wykorzystamy listy nieskończonej, nie zostanie ona wygenerowana. Co więcej, jeżeli będziemy pobierać z jej początku elementy, to również nie zostanie ona całkowicie wyznaczona. Dopiero, na przykład, wyświetlanie listy nieskończonej w konsoli spowoduje zapętlenie – lista nieskończona jest tworzona rekurencyjnie, dodając kolejne elementy do samej siebie. Posiadanie takiej konstrukcji w języku pozwala na tworzenie algorytmów na zasadzie „znajdź określoną ilość elementów, spełniających pewien warunek, iterując po nieskończonej liście tych elementów". Tworzenie list za pomocą zakresów można wykonać na kilka sposobów:

Listing 56. Tworzenie list za pomocą zakresów

```
Prelude> [1..5]
[1,2,3,4,5]
Prelude> [5..1]
```

```
[]
Prelude> ['A'..'E']
"ABCDE"
Prelude> [1,2..10]
[0,2,4,6,8,10]
Prelude> [5,4..1]
[5,4,3,2,1]
Prelude> [1..] -- lista nieskończona od 1 do nieskończoności
(należy przerwać Ctrl + C)
[1,2,3,4,5,6,7...
Interrupted.
Prelude> [1,1..] -- lista nieskończona złożona z samych jedynek
[1,1,1,1,1,1,1...
Interrupted.
```

Skoro potrafimy już tworzyć listy, poznajmy kilka podstawowych funkcji operujących na nich:

Listing 57. Podstawowe funkcje na listach

```
Prelude> length [1..10]
10
Prelude> head [1..]
1
Prelude> tail [1,2,3]
[2,3]
Prelude> last [1,2,3]
3
Prelude> init [1,2,3]
[1,2]
Prelude> reverse [1,2,3]
[3,2,3]
Prelude> reverse "olleh"
"hello"
Prelude> elem 1 [1,2,3]
True
Prelude> 5 `elem` [1,2,3]
False
Prelude> Data.List.elemIndex 2 [1,2,3]
Just 1
Prelude> Data.List.elemIndex 5 [1,2,3]
Nothing
Prelude> take 5 [1..]
[1,2,3,4,5]
Prelude> drop 5 [1..10]
[6,7,8,9,10]
Prelude> map (+1) [1,2,3]
[2,3,4]
Prelude> filter (>1) [1,2,3]
[2,3]
Prelude> foldl (+) 0 [1,2,3]
6
```

Znamy już podstawowe funkcje na listach, teraz spróbujemy zbudować funkcję `sum` (na marginesie, taka funkcja już istnieje w Prelude), liczącą sumę liczb z listy na dwa sposoby. W pierwszej implementacji wykorzystamy rekurencję, a w drugiej funkcję `foldl`:

Listing 58. Dwie implementacje funkcji sum: rekurencyjna oraz z wykorzystaniem foldl

```
Prelude> let sum1 [] = 0 -- wykorzystanie pattern matching i
rekurencji
Prelude|     sum1 (x:xs) = x + sum1 xs
Prelude> sum1 [5,7,10,20]
42
Prelude> let sum2 = foldl (+) 0 -- wykorzystanie foldl
Prelude> sum2 [5,7,10,20,1]
43
```

Bardzo ciekawym narzędziem, związanym z listami, jest składnia list składanych (ang. *list comprehension*). Konstrukcja ta umożliwia, w bardzo kompaktowej formie, zawarcie logiki tworzenia listy. Rozważmy następującą listę składaną:

Listing 59. Przykład listy składanej

```
Prelude> [2^x | x <- [1..10], x > 1, x /=5, x `mod` 2 == 1]
[8,128,512]
```

Rozpoczynając od środka, nowa lista jest budowana na podstawie listy `[1..10]` (tych list może być więcej niż jedna), z której kolejne wartości są przypisywane do x. Następnie wartości te są ograniczone warunkami po prawej stronie. Na koniec każda z wartości służy do wyliczenia wyrażenia po prawej stronie. Wyniki tych operacji tworzą nową listę. Konstrukcja ta jest bardzo elegancka i zwięzła (nawiązuje do matematycznego zapisu z teorii zbiorów), łączy w sobie możliwości, jakie dają funkcje `map` oraz `filter`. Na koniec sekcji dotyczącej list jeden z moich ulubionych przykładów z list składanych – szukanie trójkątów prostokątnych metodą naiwną:

Listing 60. Przykład wykorzystania listy składanej – szukanie trójkątów prostokątnych

```
Prelude> [(a,b,c) | a <- [1..10], b <- [1..10], c <- [1..10],
c^2 == a^2 + b^2]
[(3,4,5),(4,3,5),(6,8,10),(8,6,10)]
```

MODUŁY

Moduły w Haskellu są jednostką organizującą powiązane funkcje i typy. Tworząc bardziej skomplikowane programy, z pewnością będziemy mieli potrzebę skorzystania z jakiegoś modułu z biblioteki standardowej lub z repozytorium Hackage. Korzystanie z modułów umożliwia składnia importu (słowo kluczowe `import`). Tworzenie własnych modułów w tym miejscu pominę, wspomnę tylko, że domyślnie wszystkie funkcje i typy z modułu są publiczne, o ile nie podamy jawnie listy obiektów, jakie moduł ma uzewnętrzniać. Jak importować moduły, pokażę na przykładach modułów `Data.List` i `Data.Char`. Import modułów, wraz ze wszystkimi jego funkcjami oraz typami, odbywa się następująco:

W sesji interaktywnej:

Listing 61. Import modułu

```
Prelude> import Data.List
Prelude Data.List>
```

Można także użyć polecenia `:m + <nazwa_modulu>` i zaimportować jeden lub więcej modułów:

Listing 62. Import wielu modułów

```
Prelude> :m + Data.List Data.Char
Prelude Data.List Data.Char>
```

Natomiast w pliku źródłowym korzystamy ze słowa kluczowego `import`:

Listing 63. Import modułów w pliku źródłowym

```
module Main where

import Data.List
import Data.Char
```

Po zaimportowaniu modułu może się tak zdarzyć, że powstanie konflikt nazw funkcji lub typów. Zdarzy się to, jeżeli na przykład

zdefiniowaliśmy funkcję o takiej samej nazwie jak istniejąca funkcja w importowanym module lub w dwóch zaimportowanych modułach są funkcje o tych samych nazwach. Przykład konfliktu:

Listing 64. Przykład konfliktu nazw

```
Prelude> import Data.Char
Prelude Data.Char> isDigit 'a'
False
Prelude Data.Char> isDigit '1'
True
Prelude Data.Char> isDigit c = c == '0' -- nowa funkcja o takim
samym typie jak Data.Char.isDigit
Prelude Data.Char> isDigit '1'
False -- wywołana została nowa wersja isDigit
```

Istnieje kilka możliwości, aby takiej sytuacji uniknąć. Po pierwsze, możemy skorzystać z pełnej ścieżki do funkcji `isDigit`:

Listing 65. Wywołanie funkcji z modułu z podaniem pełnej ścieżki

```
Prelude Data.Char> isDigit '1'
False
Prelude Data.Char> Data.Char.isDigit '1'
True
```

Możemy także wymusić podawanie pełnych ścieżek do funkcji:

Listing 66. Import kwalifikowany. Wymuszenie podawania pełnych ścieżek do funkcji

```
Prelude> import qualified Data.Char
Prelude Data.Char> Data.Char.isDigit '1'
True
```

Aby skrócić zapis, możemy użyć aliasu dla modułu:

Listing 67. Import kwalifikowany z aliasem dla modułu

```
Prelude> import qualified Data.Char as Chr
Prelude Chr> Chr.isDigit '1'
True
```

Istnieje jeszcze kilka dodatkowych opcji importu modułów, które można ze sobą łączyć. Ponieważ gdy importujemy moduł, bierzemy jego wszystkie typy i funkcje do swojej przestrzeni nazw, użyteczne może być zaimportowanie tylko konkretnych funkcji z tego modułu:

Listing 68. Import wybranych funkcji z modułu

```
Prelude> import Data.Char (isSpace, isUpper)
Prelude Data.Char> isSpace ' '
True
Prelude Data.Char> isUpper 'A'
True
Prelude Data.Char> isDigit '1'
błąd...
```

Teraz ten sam import tylko z kwalifikowanym aliasem:

Listing 69. Import wybranych funkcji z modułu z aliasem

```
Prelude> import qualified Data.Char as Chr (isSpace, isUpper)
Prelude Chr> Chr.isUpper 'A'
True
```

HELLO WORLD

Do tej pory tworzony przez nas kod był uruchamiany w ramach sesji interaktywnej GHCi. Naukę nowego języka tradycyjnie rozpoczyna się od programu „Hello World", jednak w przypadku Haskella przed stworzeniem tego prostego programu należy wyjaśnić kilka spraw. Jak w języku czysto funkcyjnym stworzyć program, który będzie miał efekty uboczne (w przypadku „Hello World" będzie to pisanie do konsoli)? Okazuje się, że funkcje, które piszą lub odczytują dane z konsoli, także są funkcjami czystymi. Dopiero uruchomienie ich w specjalnym kontekście powoduje rzeczywiste efekty uboczne w postaci, na przykład, drukowania łańcuchów znaków na konsoli. Aby umożliwić interakcję z użytkownikiem, należy skorzystać z typu `IO`. Typ `IO` jest odpowiedzialny za komunikację ze światem zewnętrznym.

Listing 70. Przykładowe akcje IO

```
Prelude> :t putStrLn
putStrLn :: String -> IO ()
Prelude> :t print
print :: Show a => a -> IO ()
Prelude> :t getLine
getLine :: IO String
```

Funkcje zbudowane wokół tego typu są czyste, jednak umieszczenie ich w bloku do (*do notation*) powoduje na przykład pisanie do konsoli – czyli efekt uboczny. `IO` jest monadą. Omówienie monad zasługuje na osobny artykuł. To, co jest nam potrzebne na tę chwilę, to wiedza, że `IO` jest monadą, służy do komunikacji ze światem zewnętrznym i jest swego rodzaju pudełkiem. `IO` jest instancją `Monad`:

Listing 71. Typ Monad

```
class Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b
```

Pierwsza funkcja – `return` – umożliwia opakowanie wartości wewnątrz monady, operator `>>=`, zwany też jako funkcja *bind*, wykonuje operację na wartości wewnątrz monady i zwraca nową monadę ze zmienioną wartością wewnątrz. Posługując się analogią pudełka w opisie monad: pudełko może przechowywać wartości oraz przekształcać je w inne wartości i pakować z powrotem do pudełka. W kontekście akcji `IO` (tak nazywamy funkcje operujące na monadzie `IO`) rozpakowywanie pudełka następuje w bloku do, który stanowi syntactic sugar dla funkcji `(>>=)` klasy `Monad`. Wyjście z bloku do zamyka wartość ponownie w pudełku. No cóż, w Haskellu to, co jest łatwe, jest trudne, a to, co trudne, łatwe – teraz możemy napisać (z pełnym zrozumieniem) program „Hello World".

Listing 72. Program Hello World

```
module Main where

main :: IO ()
main = do
  putStrLn "Hello World"
```

Funkcja `main` ma szczególne znaczenie, jeżeli znajduje się w module o nazwie `Main`. Po pierwsze, jest punktem wejścia do programu (podobnie jak w C), po drugie – jest akcją `IO`. Oznacza to, że właśnie w tym miejscu powinna odbywać się interakcja z użytkownikiem z wykorzystaniem innych akcji, takich jak `print`, `putStrLn` czy `getLine`. Zapisujemy ten kod w pliku *HelloWorld.hs* w dowolnej lokalizacji na dysku (w moim przypadku jest to katalog C:\Haskell). Następnie uruchamiamy wiersz poleceń i kompilujemy program.

Listing 73. Kompilacja i uruchomienie programu Hello World

```
C:\Haskell> ghc HelloWorld.hs
[1 of 1] Compiling Main         ( HelloWorld.hs, HelloWorld.o )
Linking HelloWorld.exe

C:\Haskell>HelloWorld.exe
Hello World
```

Pierwszy program skompilowany i uruchomiony. Zmodyfikujemy teraz ten program, tak aby umożliwiał interakcję z użytkownikiem, który zostanie poproszony o podanie swojego imienia. Do pliku *Greeting.hs* dodajemy następujący kod:

Listing 74. Program Greeting

```
module Main where

main :: IO ()
main = do
  putStrLn "Enter your name please."
  name <- getLine
  putStrLn $ "Hello " ++ name
```

Kompilacja i uruchomienie:

Listing 75. Kompilacja i uruchomienie programu Greeting

```
C:\Haskell> ghc Greeting.hs
[1 of 1] Compiling Main          ( Greeting.hs, Greeting.o )
Linking Greeting.exe

C:\Haskell>Greeting.exe
Enter your name please
Bob
Hello Bob
```

Funkcja `getLine` zwraca akcję `IO String`. W bloku do wartość związana z akcją pobiera się z pomocą operatora `<-`, który binduje tekst wpisany w konsoli ze zmienną `name`. Zmienna ta przechowuje wartość wpisaną w konsoli, której używany w kolejnym wierszu programu.

Chciałbym przedstawić jeszcze jeden program. Program ten będzie wyznaczał N-tą liczbę pierwszą, gdzie n będzie wprowadzane przez użytkownika w konsoli. Pominę tutaj sprawdzanie poprawności wprowadzanej liczby, założymy, że podana liczba będzie poprawna (czyli będzie liczbą całkowitą, większą od zera). Napisanie tego programu jest pretekstem do przedstawienia narzędzia caball-install oraz repozytorium Hackage (http://hackage.

haskell.org/). Podczas tworzenia większych aplikacji, aby wygodniej zarządzać pakietami, zalecane jest skorzystanie z narzędzia cabal-sandbox, które tworzy wyizolowaną przestrzeń dla naszego projektu. W tej przestrzeni umieszczamy swoje pliki źródłowe oraz instalujemy zależności. Wracając do cabal-install, narzędzie to służy do instalacji w naszym środowisku developerskim pakietów z repozytorium Hackage. Cabal jest formatem pakietów, który definiuje między innymi, z jakich modułów składa się pakiet oraz jakie ma zależności. Narzędzie cabal-init pozwala na tworzenie pakietów z własnych modułów, które możemy potem opublikować w Hackage. Hackage zawiera ogromną liczbę pakietów, pogrupowanych kategoriami. Pakiety możemy wyszukiwać po słowach kluczach. Polecam odwiedzenie Hackage, zwłaszcza zanim rozpoczniemy implementować własne rozwiązania. Prawdopodobnie znajdziemy tam biblioteki, które ułatwią nam pracę. W naszym programie z liczbami pierwszymi skorzystamy z pakietu „primes", który znajdziemy w Hackage (http://hackage.haskell.org/package/primes). Korzystając z narzędzi cabal, aktualizujemy listę pakietów Hackage, a następnie pobieramy pakiet „primes".

Listing 76. Pobranie listy pakietów z Hackage oraz instalacja pakietu primes

```
C:\Haskell>cabal update
Downloading the latest package list from hackage.haskell.org

C:\Haskell>cabal install primes
...
```

Po zainstalowaniu pakietu przyjrzyjmy się dokumentacji pakietu (http://hackage.haskell.org/package/primes-0.2.1.0/docs/Data-Numbers-Primes.html). Interesuje nas pierwsza pozycja – nieskończona lista `primes`:

Listing 77. Lista primes

```
primes :: Integral int => [int]
```

Następnie tworzymy plik *GetPrimeNumber.hs* i umieszczamy w nim następujący kod:

Listing 78. Program GetPrimeNumber

```
module Main where

import Data.Numbers.Primes

main :: IO ()
main = do
  putStrLn "Enter number:"
  numberStr <- getLine
  let n = read $ numberStr :: Int
  let primeNumber = head . drop (n - 1) $ primes
putStrLn $ "Prime number for " ++ show n ++ " is " ++ show
primeNumber
```

Kompilacja i uruchomienie programu z różnymi parametrami:

Listing 79. Kompilacja i uruchomienie programu GetPrimeNumber

```
C:\Haskell>ghc GetPrimeNumber.hs
...
C:\Haskell>GetPrimeNumber.exe
Enter number:
1
Prime number for 1 is 2
C:\Haskell>GetPrimeNumber.exe
Enter number:
2
"Prime number for 2 is 3
C:\Haskell>GetPrimeNumber.exe
Enter number:
3
Prime number for 3 is 5
C:\Haskell>GetPrimeNumber.exe
Enter number:
1000
Prime number for 1000 is 7919
```

PODSUMOWANIE

Haskell, dla osób nie mających jak dotąd styczności z programowaniem funkcyjnym, może wydawać się językiem ezoterycznym. Fakt, ponieważ język jest czysto funkcyjny, nakłada on sporo obostrzeń i wymusza zmianę sposobu myślenia podczas tworzenia programu. Dla mnie najważniejsza i najbardziej wartościowa w nauce Haskella była właśnie ta zmiana sposobu myślenia. Podchodząc do nauki tego języka, a będąc mocno umysłowo osadzonym w świecie programowania imperatywnego, musimy pokonać pewną barierę, przestawić nieco sposób myślenia. Jak dla mnie był to niesamowity trening i polecam to każdemu, choćby wyłącznie jako sposób na rozwój intelektualny. Zdaję sobie sprawę, że Haskell nie należy do najpopularniejszych języków i być może nigdy nie będzie się do nich zaliczał. Z drugiej strony ma silne podstawy teoretyczne, jest dojrzały i konsekwentny. Haskell nie jest tylko językiem akademickim – wiele znanych firm (np. Google, Facebook) używa Haskella do tworzenia różnego rodzaju wewnętrznych narzędzi (https://wiki.haskell.org/Haskell_in_industry). Warto również wymienić jeden z najczęściej pobieranych pakietów z Hackage – framework webowy Yesod. Zachęcam do dalszej nauki tego ciekawego języka. Sam korzystałem z bardzo przyjemnie napisanego tutoriala „Learn You a Haskell for Great Good!" (http://learnyouahaskell.com). Natomiast do poćwiczenia języka polecam zadania znajdujące się na stronie https://www.codewars.com/, gdzie, na tę chwilę, znajduje się prawie 700 zadań z Haskella, o różnym stopniu trudności.

Warto dać Haskellowi szansę – nauka każdego innego języka funkcyjnego, czy paradygmatu funkcyjnego w ogóle, będzie dla nas prostsza, a podejście funkcyjne w programowaniu może się przydać w codziennej pracy z językami z głównego nurtu, jak C# czy Java.

PAWEŁ WICHER

Programista .NET. Od kilku lat zajmuje się komercyjnym tworzeniem aplikacji w oparciu o stos technologiczny Microsoft. Aktywny na codewars. W wolnych chwilach gra w piłkę nożną, uczy się nowych języków programowania (ostatnio Haskell) oraz gra na basie.