Езикът Haskell

# Неща за дописване и научаване

* where – ключова дума
* curried functions
* ‘data’

# Увод

Това са записки по функционално програмиране с Haskell от курса „Функционално програмиране“ във ФМИ и „Практикум по функционално програмиране“ във ФМИ. Книгата, от която основно съм чел е „Graham Hutton - Programming in Haskell-Cambridge University Press (2016)“.

Компилаторът, който ползваме е „Glasgow Haskell Compiler“.

# Основни понятия и конструкции в Haskell

## Функции

### Дефиниция на функция

Както в останалите програмни езици, дефинирането на функции в Haskell става като зададем име, аргументи и тяло на функцията. В Haskell това става със следното равенство:



При прилагането на функция върху аргументи, аргументите се заместват в тялото на функцията. Това може да доведе до директна сметка над числа (на пример 2 + 3), която не може да се опрости повече и просто се смята, или може да е израз, в който има прилагане на други функции над аргументи. Във втория случай отново се заместват аргументите в тялото на извиканите функции и така нататък докато се стигне до израз, който не може да се опрости повече и просто се смята.



Можеше първо да заместим външното викане на double. В общия случай редът на заместването не променя крайния резултат, но може да промени броя на стъпките нужни за достигането му.



Засега това ще е достатъчно добър модел за прилагането на функции върху аргументи и няма да се задълбочавам повече[[1]](#footnote-1).

### Прилагане на функции

Прилагаме функция така:

<име-на-функцията> <аргумент-1> <аргумент-2> . . . <аргумент-n>

Името на функцията последвано от аргументите разделени с бяло поле.

В Haskell прилагането на функция е с най-висок приоритет от всички операции. Ето няколко примера:

* f(a, b) + a\*g(b), в Haskell ще се запише така: f a b + a \* g b
* f(a, g(b)), ще се запише така: f a (g b)
* f(g(a)), ще се запише така: f (g a)

## Няколко основни неща, които Haskell поддържа

* **Concise programs** – Haskell е специално направен да е много изразителен, тоест малко текст да има много значение.
* **Powerful type system** – строго типизиран, поддържа полиморфизъм и overloading.
* **List comprehensions** – поддържа списъци на базово ниво, може лесно да се манипулират списъци без да пишем рекурсия експлицитно.
* **Recursive functions** – основният механизъм за циклене е рекурсията (рекурсивни функции).
* **Higher-order functions** – (функции от по-висок ред) това са функции, които приемат като аргумент или връщат функция.
* **Generic functions** – подобно на шаблоните в C++ тук може да правим функции, които работят с множество от типове.
* **Lazy evaluation** – това е идеята, че изчислението не се прави докато не е нужно[[2]](#footnote-2).
* **Equational reasoning** – хиляди пъти по-лесни доказателства за коректност на програмите, отколкото в императивните езици.

## Standard prelude, скриптове, GHC

Стандартната библиотека в Haskell се казва “Standard prelude”.

Файловете на Haskell имат разширението .hs (не е задължително, но е препоръчително да е това разширение). Както при всеки друг програмен език, това са просто текстови файлове. В тези файлове пишем дефинициите на функции и на разни други неща.

**GHC** (Glasgow Haskell Compiler) е opensource компилатор, който ще ползваме. Той също има и интерпретатор към него (пишем директно изрази в терминала и той ги оценява). **GHC** може да се интегрира с може би почти всеки текстов редактор, като асистентът препоръча да ползваме **VS Code** или **Vim**. Общо взето писането на код ще има следните стъпки:

1. Отваряме .hs файловете (или project файл) в текстовия редактор.
2. Зареждаме същите файлове в **GHC** през терминала с командата :load <име>
3. Пишем в .hs файловете и ги запазваме.
4. В терминала пишем :reload, за да компилираме. Ако не се компилира, оправяме си грешките.
5. Повтаряме стъпки 3. и 4.

## Синтаксис

Ключови думи

case class data default deriving do else foreign if import in infix infixl infixr instance let module newtype of then type where

Имена

Имената на функции и аргументите ѝ трябва да започват с малки букви. Следващите символи може да са букви (малки, главни), долни черти, цифри и единични кавички. Примери за валидни имена: addTwice f\_2 x’ fun1 xs.

Конвенция: имената на списъци да завършват на “s”. Логиката е, че в английския език множественото число завършва на “s”, и в списъците имаме множество от елементи (ns – numberS, cs – characterS, bs – booleanS).

Индентация

Вместо скобите {} се ползва основно индентация за лесно четене на кода[[3]](#footnote-3). Пример:



Отбелязвам, че **не може** да се пропуска else.

Коментари

Коментарите на един ред започват с „—-“. Коментарите на няколко реда се обграждат с „{- -}”.

## Типове

Haskell е строго типизиран и в него всяко „нещо“ си има тип. Ползваме означението V :: T, за да кажем, че V е стойност от тип T. На пример:



Добре е да се отбележи, че двата клона на if конструкцията трябва да връщат един и същи тип. На пример това ще даде грешка, защото 1 е от тип Num, a False от тип Bool: if True then 1 else False.

Вградени типове

* Bool
* Char
* String
* Int – fixed-precision integers
* Integer – arbitrary-precision integers
* Float – 32 bit
* Double – 64 bit
* Списъци – пример: [’a’,’b’,’c’,’d’] :: [Char]
* Наредени двойки (tuple) – пример: ("Yes",True,’a’) :: (String,Bool,Char)
* Функции – пример: not :: Bool -> Bool

Полиморфни типове

Подобно на шаблоните в C++ в Haskell може да имаме функции, които работят на произволни типове. На пример вградената функция length, която връща дължината на подадения списък, не се интересува от типа на елементите на списъка и нейният тип е length :: [a] -> Int. “a” е просто някакъв име за произволния тип и се нарича type variable. Обикновено типовите променливи ги кръщаваме с малки букви (за тях важат същите правила за именуването). В случая length е от тип функция, която приема списък с елементи от произволен тип и връща число. Тип с една или повече типови променливи наричаме полиморфен тип (polymorphic type).

Overloaded types, class constraints

Подобно на полиморфизма може да дефинираме функция да работи само над определено множество от типове. На пример функцията (+) (стандартно събирана на числа има следния тип: (+) :: Num a => a -> a -> a, където Num a се нарича class constraint. Тоест това казва, че (+) може да се прилага над всички типове, които са числа. Class constraints са във вида C a, където C e клас и a е типова променлива. Тип, който съдържа class constraints, се нарича overloaded type.

1. В действителност не знам дали това прави компилаторът, но ако не, то ще е нещо еквивалентно като резултат. [↑](#footnote-ref-1)
2. Най-вероятно ще съм написал по-добро обяснение накъде… [↑](#footnote-ref-2)
3. Могат да се ползват и скоби и „;“, но се счита за по-добре да се разчита на индентацията. [↑](#footnote-ref-3)