Вивчить собі Хаскела на велике щастя!

Автор: Міран Ліповача

Переклад здійснили: Ганна Лелів, Семен Тригубенко, Богдан Пеньковський, Марина Стрельчук і Тетяна Богдан

Мовні редактори: Тетяна Богдан і Ганна Лелів
Науковий редактор: Семен Тригубенко

Переклад виконано за підтримки
Словненя

2017-05-21T00:06:25Z
Версія v4.7-54-gda41cf2

Зміст

6 Функції вищого порядку **1**

6.1 Карійовані функції 1

6.2 Із вищим порядком усе в порядку 4

6.3 Відображення і фільтри 8

6.4 Лямбди 13

6.5 Згортком і батька легше бити 15

6.6 Доларове застосування функцій 21

6.7 Композиція функцій 22

Покажчик **26**

Розділ 6

Функції вищого порядку

Переклад українською Ганни Лелів

У Хаскелі функції можуть брати інші функції як параметри і повертати функції як **значення-результати** [return values]. Функція, що бере або повертає інші функції, називається функцією вищого порядку. Функції вищого порядку — це не просто інгредієнт хаскельного борщу, вони власне і є хаскельним борщем. Виявляється, що, якщо ви захочете щось порахувати за рахунок означування на кшталт «щось *дорівнює* чомусь», а не за рахунок *описання* кроків, що змінюють якийсь стан (можливо, навіть, у циклі), то саме функції вищого порядку стануть вашими незамінними супутниками. Це дуже потужний підхід для розв'язання задач і для міркування про програми.

6.1 Карійовані функції

У Хаскелі кожна функція формально бере тільки один параметр. То як же, раніше, ми змогли означити і використати кілька функцій, що приймають декілька параметрів? О, це спритний викрутас! Дотепер, усі функції, що приймали *кілька параметрів*, були **карійованими**. Що це таке? Найкраще навести приклад. Ось наша добра знайома — функція `max`. Виглядає так, наче вона бере два параметри і повертає більший із них. Але насправді, виконання `max 4 5` сперт

шу створює функцію, що бере один параметр, і повертає або `4`, або переданий параметр, залежно від того, який зі двох є більшим. Тоді до цієї функції передають `5`, і функція повертає очікуваний результат. Звучить не зовсім зрозуміло, але насправді це суперкрута концепція. Ці два виклики рівнозначні:

```
ghci> max 4 5
5
ghci> (max 4) 5
5
```

Пробіл між двома числами — це всього лиш **застосування функції**. Пробіл — це немовби оператор, який має найвищий пріоритет. Розгляньмо тип `max`. Він є `max :: Ord a => a -> a -> a`. Або ж, можна також подати його отак: `max :: Ord a => a -> (a -> a)`. Це, в свою чергу, можна прочитати ось як: `max` приймає `a` і повертає (повернення позначається тут `->`) функцію, що приймає `a` і повертає `a`. Ось чому немає особливої різниці між типом результату і типами параметрів функцій — вони всі просто відокремлені стрілочками.

Яка нам із цього користь? Просто кажучи, якщо ми викличемо функцію із недостатньою кількістю параметрів, то отримаємо **частково застосовану** функцію, тобто, функцію, що бере стільки параметрів, скільки ми їй не додали. Вдатися до часткового застосування (іншими словами — до виклику функції із недостатньою кількістю параметрів) — це вправний спосіб створення нових функції «на ходу», щоб можна було відразу ж передати їх іншим функціям. Це непоганий спосіб створення нових функції, коли треба «зарядити» вже існуючі функції якимись даними.

Погляньте на цю сміховинно просту функцію:

```
multThree :: Num a => a -> a -> a -> a
multThree x y z = x * y * z
```

Що насправді відбувається, коли ми виконуємо `multThree 3 5 9` або `((multThree 3) 5) 9`? Спершу, `3` застосовується до `multThree`, бо вони відокремлені пробілом. Отримуємо функцію, що приймає один параметр і повертає функцію. Тоді до неї застосовується `5`, що втворює функцію, яка бере параметр і множить його на 15. До тієї функції застосовується `9`, і в результаті отримуємо 135 (або щось дуже схоже на 135). Запам'ятайте, що тип цієї функції

можна також записати як `multThree :: Num a => a -> (a -> (a -> a))`. Перед `->` записаний параметр, що його приймає функція. А після нього — те, що функція повертає. Отож, наша функція приймає `a` і повертає функцію, що має тип `Num a => a -> (a -> a)`. Ця нова функція, в свою чергу, приймає `a` і повертає функцію, що має тип `Num a => a -> a`. І насамкінець ця функція приймає `a` і повертає `a`. Ось погляньте:

```
ghci> let multTwoWithNine = multThree 9
ghci> multTwoWithNine 2 3
54
ghci> let multWithEighteen = multTwoWithNine 2
ghci> multWithEighteen 10
180
```

Викликаючи функції із, так би мовити, недостатньою кількістю параметрів, ми на ходу створюємо нові функції. А якби ми захотіли створити функцію, що бере число і порівнює його із `100`? Можна було б зробити ось так:

```
compareWithHundred :: (Num a, Ord a) => a -> Ordering
compareWithHundred x = compare 100 x
```

Якщо викликати функцію із `99`, вона поверне `GT`. Усе просто. Зверніть увагу, що `x` стоїть праворуч по обидва боки рівняння. А тепер подумаймо, що повертає `compare 100`. Вона повертає функцію, що бере число і порівнює його із `100`. Оце так! А хіба це не функція, що ми хотіли отримати? Перепишімо попереднє означення ось як:

```
compareWithHundred :: (Num a, Ord a) => a -> Ordering
compareWithHundred = compare 100
```

Оголошення типу не змінилося, тому що `compare 100` повертає функцію. `compare` має тип `Ord a => a -> (a -> Ordering)`, і якщо її викликати із `100`, то отримаємо результат, що має тип `(Num a, Ord a) => a -> Ordering`. Сюди прокралася додаткова умова типокласу, адже `100` на додачу належить до типокласу `Num`.

Люди! Упевніться, що ви дійсно розумієте, як працюють карійовані функції і часткове застосування, адже вони дуже важливі!

Інфіксні функції також можна застосовувати частково за допомогою розтину. Щоб розітнути інфіксну функцію, просто візьміть її у дужки і вкажіть параметр тільки з одного боку. Це побудує функцію, що бере один параметр і подає його в ту частину, де бракувало операнда. Нахабно банальна функція:

```
divideByTen :: Floating a => a -> a
divideByTen = (/10)
```

Викликати, скажімо, `divideByTen 200` — те ж саме, що виконати `200 / 10` чи `(/10) 200`. Ось функція, яка перевіряє, чи даний їй символ є великою літерою:

```
isUppercaseLetter :: Char -> Bool
isUppercaseLetter = (`elem` ['A'..'Z'])
```

Єдина особливість розтину — це використання `-`. Згідно із означенням розтину, `(-4)` створить функцію, що бере число і віднімає від нього 4. Але, як очікує більшість із нас, `(-4)` таки означає мінус чотири. Отож, якщо вам потрібно створити функцію, що віднімає 4 від числа, яке вона отримує як параметр, частково застосуйте функцію `subtract` ось як: `(subtract 4)`.

Що трапиться, якщо ми спробуємо виконати `multThree 3 4` у GHCi замість того, щоб прив'язати її до імені за допомогою *let* або передати її іншій функції?

```
ghci> multThree 3 4
<interactive>:1:0:
    No instance for (Show (t -> t))
      arising from a use of `print' at <interactive>:1:0-12
    Possible fix: add an instance declaration for (Show (t -> t))
    In the expression: print it
    In a 'do' expression: print it
```

GHCi каже, що цей вираз створив функцію типу `a -> a`, але не знає, як вивести її на екран. Функції не втілюють типоклас `Show`, тому нам не вдалося отримати гарненьке рядкове представлення функції. Якщо на запрошення GHCi відповісти `1 + 1`, то компілятор спершу обчислить це як `2`, а тоді викличе `show` по `2`, щоб отримати це число у текстовому вигляді. А текстове представлення `2` — це всього лише рядок `"2"`, який і виводиться на екран.

6.2 Із вищим порядком усе в порядку

Функції можуть приймати функції як параметри і повертати функції. Щоб це продемонструвати, напишемо функцію, що бере функцію, а тоді двічі її до чогось застосовує!

```
applyTwice :: (a -> a) -> a -> a
applyTwice f x = f (f x)
```

Насамперед зверніть увагу на оголошення типу. Раніше нам не потрібні були дужки, тому що `->` сам по собі правоасоціативний. Але у цьому випадку дужки обов'язкові. Вони означають, що перший параметр — це функція, яка щось приймає і повертає щось інше, але такого ж типу. Другий параметр — це щось такого ж типу, а значення-результат — теж має такий самий тип. Це оголошення типу можна прочитати на карійований манер, але щоб даремно не перевантажувати мозок, я скажу тільки те, що ця функція бере два параметри і повертає одне значення. Перший параметр — це функція (що має тип `a -> a`), а другий — має тип `a`. Функція може виглядати як `Int -> Int` або `String -> String` або як завгодно. Але другий параметр мусить бути такого ж самого типу, як аргумент і результат функції, що її надано як перший аргумент.

Примітка: Надалі я казатиму, що функції приймають декілька параметрів попри те, що кожна функція насправді бере тільки один параметр і повертає частково застосовану функцію. Аж поки ми не дійдемо до функції, що повертає «тверде» значення. Щоб не ускладнювати, я казатиму, що `a -> a -> a` приймає два параметри, хоч ми і знаємо, *що* там відбувається насправді.

Тіло функції доволі просте. Ми всього лише використовуємо параметр `f` як функцію, застосовуємо її до `x`, відокремивши їх пробілом, а тоді знову застосовуємо `f` до результату. Годі балакати — трохи побавмося із цією функцією:

```
ghci> applyTwice (+3) 10
16
ghci> applyTwice (++ " HAHA") "HEY"
"HEY HAHA HAHA"
ghci> applyTwice ("HAHA " ++) "HEY"
"HAHA HAHA HEY"
ghci> applyTwice (multThree 2 2) 9
144
ghci> applyTwice (3:) [1]
[3,3,1]
```

Очевидно, що часткове застосування — це страшенно крута і корисна штука. Якщо функція вимагає, щоб ми передали їй функцію, що приймає тільки

один параметр, ми можемо частково застосувати таку функцію до «моменту», коли лишатиметься функція одного параметру, а тоді передати її.

А тепер використаймо програмування вищого порядку, щоб реалізувати надзвичайно корисну функцію зі стандартної бібліотеки. Вона називається `zipWith`, і бере вона функцію і два списки як параметри, а тоді сполучає ті два списки, застосувавши функцію між відповідними елементами. Ось як ми її реалізуємо:

```
zipWith' :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith' _ [] _ = []
zipWith' _ _ [] = []
zipWith' f (x:xs) (y:ys) = f x y : zipWith' f xs ys
```

Погляньте на оголошення типу. Перший параметр — це функція, що бере дві штуки і будує третю. Вони не мусять належати до одного типу, але це можливо. Другий і третій параметри — списки. Результат — також список. Другий параметр мусить бути списком з елементів типу `a`, оскільки сполучна функція приймає `a` як свій перший аргумент. Третій параметр мусить бути списком з елементів типу `b`, адже другий параметр сполучної функції належить до типу `b`. Результат — список з елементів типу `c`. Якщо оголошення типу функції каже, що функція приймає функцію `a -> b -> c` як параметр, то вона прийме і функцію `a -> a -> a`, але не навпаки! Запам'ятайте: коли ви пишете функції, особливо функції вищого порядку, і не впевнені, до якого типу вони належать, ви можете спробувати не оголошувати тип одразу, а після написання використати `:t`, щоб дізнатися, який тип вивів для тієї функції Хаскел.

Поведінка цієї функції схожа до звичайної `zip`. Крайові умови такі самі, з'являється тільки додатковий аргумент — сполучна функція, — але цей аргумент не грає ролі у крайових умовах, тому ми просто пишемо `_` для нього. Тіло функції в останньому взірці також подібне до `zip`, тільки воно виконує не `(x,y)`, а `f x y`. Одна функція вищого порядку може мати безліч застосувань, якщо вона достатньо загальна. Ось невеличка демонстрація того, що може наша функція `zipWith'`:

```
ghci> zipWith' (+) [4,2,5,6] [2,6,2,3]
[6,8,7,9]
ghci> zipWith' max [6,3,2,1] [7,3,1,5]
[7,3,2,5]
ghci> zipWith' (++) ["foo ", "bar ", "baz "] \
                    ["fighters", "hoppers", "aldrin"]
["foo fighters","bar hoppers","baz aldrin"]
ghci> zipWith' (*) (replicate 5 2) [1..]
[2,4,6,8,10]
```

```
ghci> zipWith' (zipWith' (*)) [[1,2,3],[3,5,6],[2,3,4]] \
                              [[3,2,2],[3,4,5],[5,4,3]]
[[3,4,6],[9,20,30],[10,12,12]]
```

Як бачите, одну функцію вищого порядку можна використати багатьма різноманітними способами. Імперативні мови програмування використовують речі на кшталт циклів for, циклів while, збереження чогось в змінну, перевірку її стану і ще багато іншого, щоб домогтися бажаної поведінки, а тоді загортають це діло в інтерфейс функції. Функційні мови програмування використовують функції вищого порядку, які абстрагують ідіоми програмування в собі, — ідіоми на кшталт паралельної обробки двох списків, на рівні пар, із виконанням якоїсь роботи із кожною парою, або ж, отримання множини розв'язків і вилучення із неї непотрібних.

Реалізуймо іншу функцію, що вже входить до стандартної бібліотеки і має назву `flip`. `flip` бере функцію і повертає функцію, яка виглядає, як першопочаткова функція, тільки от перші два аргументи помінялися місцями. Можемо реалізувати її ось так:

```
flip' :: (a -> b -> c) -> (b -> a -> c)
flip' f = g
    where g x y = f y x
```

Згідно із оголошенням типу, можна сказати, що ця функція бере функцію, що приймає `a` і `b` і повертає функцію, яка приймає `b` і `a`. Але оскільки функції за замовчуванням карійовані, друга пара дужок — зайва, адже `->` за замовчуванням є правоасоціативним. `(a -> b -> c) -> (b -> a -> c)` — те ж саме, що і `(a -> b -> c) -> (b -> (a -> c))`, що, своєю чергою, те ж саме, що `(a -> b -> c) -> b -> a -> c`. Ми написали, що `g x y = f y x`. Якщо це істинно, то `f y x = g x y` — істинно також, чи не так? Пам'ятаючи про це, означмо цю функцію ще простіше:

```
flip' :: (a -> b -> c) -> b -> a -> c
flip' f y x = f x y
```

У цьому випадку ми скористалися тим, що ці функції карійовані. Коли ми викликаємо `flip' f` без параметрів `y` і `x`, вона повертає `f`, що бере ті два параметри, але вони тепер помінялися місцями. Як бачимо, каріювання полегшує написання функцій вищого порядку, особливо якщо наперед подумати про те, як виглядатиме кінцевий результат мовою тих функцій (себто, функції-аргументів до функцій вищих порядків), але застосованих повністю.

```
ghci> flip' zip [1,2,3,4,5] "hello"
[('h',1),('e',2),('l',3),('l',4),('o',5)]
```

```
ghci> zipWith (flip' div) [2,2..] [10,8,6,4,2]
[5,4,3,2,1]
```

6.3 Відображення і фільтри

`map` бере функцію і список та застосовує цю функцію до кожного елемента списку, створюючи новий список. Погляньмо, яка в неї сиґнатура типу та як вона означена.

```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = f x : map f xs
```

Сиґнатура типу каже, що `map` як перший аргумент бере функцію, яка приймає `a` і повертає `b`, як другий аргумент — список з елементів типу `a` і повертає список з елементів типу `b`. Цікаво, що тут, як зазвичай і з іншими функціями в Хаскелі, всього лиш подивившись на сиґнатуру типу функції, можна сказати, *що* ця функція робить. Функція `map` є однією із тих універсальних функцій вищого порядку, що їх можна використати у мільйон різних способів. Ось вона в дії:

```
ghci> map (+3) [1,5,3,1,6]
[4,8,6,4,9]
ghci> map (++ "!") ["BIFF", "BANG", "POW"]
["BIFF!","BANG!","POW!"]
ghci> map (replicate 3) [3..6]
[[3,3,3],[4,4,4],[5,5,5],[6,6,6]]
ghci> map (map (^2)) [[1,2],[3,4,5,6],[7,8]]
[[1,4],[9,16,25,36],[49,64]]
ghci> map fst [(1,2),(3,5),(6,3),(2,6),(2,5)]
[1,3,6,2,2]
```

Мабуть, ви помітили, що ці результати можна було б також отримати за допомогою спискових характерів. `map (+3) [1,5,3,1,6]` — те ж саме, що `[x+3 | x <- [1,5,3,1,6]]`. Але реалізації із використанням `map` набагато легше читати у випадках, коли ви просто застосовуєте якусь функцію до елементів списку, зокрема коли маєте справу з відображеннями відображень. Відповідний код із характерами буде більш неохайним, адже там швидше за все будуть дужка на дужці, і дужка зверху.

`filter` — це функція, що бере предикат (до речі, предикат — це функція, що каже, чи є щось істинним чи ні; отож, у нашому випадку — це функція,

яка повертає булеве значення) і список, а тоді повертає список з елементів, які задовольняють предикат. Сиґнатура типу і реалізація виглядають ось так:

```
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter p (x:xs)
    | p x       = x : filter p xs
    | otherwise = filter p xs
```

Досить просто. Якщо `p x` після обчислення приймає значення `True`, відповідний елемент потрапляє до нового списку. Якщо ні, то ні — він відфільтровується. Кілька прикладів використання:

```
ghci> filter (>3) [1,5,3,2,1,6,4,3,2,1]
[5,6,4]
ghci> filter (==3) [1,2,3,4,5]
[3]
ghci> filter even [1..10]
[2,4,6,8,10]
ghci> let notNull x = not (null x) in \
    filter notNull [[1,2,3],[],[3,4,5],[2,2],[],[],[]]
[[1,2,3],[3,4,5],[2,2]]
ghci> filter (`elem` ['a'..'z']) "u LaUgH aT mE BeCaUsE I aM diFfeRent"
"uagameasadifeent"
ghci> filter (`elem` ['A'..'Z']) "i lauGh At You BecAuse u r aLL the Same"
"GAYBALLS"
```

Усього цього ми б досягли і зі списковими характерами, скориставшись предикатами. Не існує чіткого правила, коли використовувати `map` і `filter`, а коли спискові характери. Ви вирішуєте на власний розсуд, що буде більш читабельним у конкретному коді та контексті. Для функції `filter` еквівалентом застосування кількох предикатів у списковому характері є або фільтрування чогось декілька разів, або сполучення предикатів за допомогою логічної функції `&&`.

Пам'ятаєте функцію швидкого сортування з розділу ??? Ми скористалися списковими характерами, щоб відфільтрувати елементи списку менші (чи рівні) за значенням або більші за стрижень. Те ж саме можна реалізувати і за допомогою `filter`, і результат буде набагато читабельніший:

```
quicksort :: Ord a => [a] -> [a]
quicksort [] = []
quicksort (x:xs) =
    let smallerSorted = quicksort (filter (<=x) xs)
        biggerSorted = quicksort (filter (>x) xs)
```

```
    in  smallerSorted ++ [x] ++ biggerSorted
```

Відображення та фільтрування — це повсякденний інструмент програміста, який пише функційною мовою. Ага. Немає значення, чи ви користуєтеся функціями `map` і `filter`, чи списковими характерами. Пригадуєте, як ми розв'язали задачу, де потрібно було знайти правильні трикутники із певним периметром? Якби ми писали імперативною мовою, то мали б три вкладені цикли у розв'язку, а усередині тестували б, чи задовольняє поточна комбінація умови правильного трикутника і чи має він такий як треба периметр. Якщо так, ми б вивели трикутник на екран (або ще щось). У функційному програмуванні ідіомою для розв'язання такої задачі є відображення і фільтрування. Ви пишете функцію, яка приймає якесь значення і виводить якийсь результат. Ми відображаємо за допомогою цієї функції список значень, а тоді фільтруємо отриманий список, щоб залишилися тільки результати, що задовольняють критеріям пошуку. Завдяки тому, що Хаскел є лінивим, навіть якщо ви відобразите щось по списку кілька разів і профільтруєте результати теж кілька разів, він пройдеться по списку тільки один раз.

Знайдімо найбільше число менше ніж 100000, яке ділиться на 3829. Для цього профільтруймо множину можливих варіантів, серед яких знаходиться розв'язок.

```
largestDivisible :: Integral a => a
largestDivisible = head (filter p [100000,99999..])
    where p x = x `mod` 3829 == 0
```

Спершу ми складаємо список усіх чисел менших ніж 100000 у порядку спадання. Тоді фільтруємо цей список за допомогою предиката. Оскільки числа посортовані від більшого до меншого, найбільше число, що задовольняє наш предикат, буде першим елементом у профільтрованому списку. Нам навіть не треба було починати працювати із скінченним списком. Знову ж таки, бачимо лінивість Хаскела в роботі. Оскільки ми беремо лише голову профільтрованого списку, нам байдуже, чи цей список скінченний чи ні. Розрахунки буде припинено, як тільки буде знайдено перший розв'язок, що задовольняє критеріям пошуку.

Тепер ми **знайдемо суму усіх непарних квадратів, менших за 10000**. Але спершу потрібно ввести функцію `takeWhile`, яка нам буде потрібна для розв'язку. Ця функція бере предикат і список, а тоді йде від початку списку і повертає елементи, і повертатиме доки предикат лишати-

меться істинним. Щойно ця функція знаходить елемент, для якого предикат є хибним, вона зупиняється. Якщо нам потрібно отримати перше слово з рядка `"elephants know how to party"`, ми можемо написати `takeWhile (/=' ') "elephants know how to party"`, що поверне `"elephants"`. Гаразд. Сума усіх непарних квадратів, менших за 10000. Спершу відобразимо за допомогою функції `(^2)` по нескінченному списку `[1..]`. Тоді профільтруємо результати, щоб залишилися тільки непарні квадрати. А тепер відберемо з цього списку елементи, які менші за 10000. Врешті-решт, ми отримаємо суму списку. Навіть не потрібно означувати для цього функцію. Достатньо одного рядка коду в GHCi:

```
ghci> sum (takeWhile (<10000) (filter odd (map (^2) [1..])))
166650
```

Чудово! Ми починаємо з якихось даних на вході (нескінченний список усіх натуральних чисел), відображаємо по ньому за допомогою функції, фільтруємо список-результат та обрізаємо його тоді, коли треба, а тоді вираховуємо суму. Це можна було б написати і за допомогою спискових характерів:

```
ghci> sum (takeWhile (<10000) [n^2 | n <- [1..], odd (n^2)])
166650
```

Як виглядає гарніше — це вже питання смаку. Знову ж таки, усе завдяки лінощам Хаскела. Можемо відобразити по нескінченному списку та профільтрувати його, тому що Хаскел не буде відразу ж відображати і фільтрувати. Він відкладе ці дії на потім. І лише коли ми змусимо Хаскел показати нам суму, функція `sum` скаже `takeWhile`, що їй потрібні ті числа. `takeWhile` дає старт фільтруванню та відображенню, яке припиняється, щойно було знайдене число, яке більше ніж чи дорівнює 10000.

У наступному завданні ми матимемо справу із послідовностями Коллатца. Беремо натуральне число. Якщо воно парне, то ділимо його на 2. Якщо непарне, то множимо на 3 і додаємо 1. Те саме робимо із отриманим числом, дістаємо нове число, і так далі. По суті ми отримуємо ланцюжок чисел. Вчені вважають, що, яке б початкове число ми не брали, ланцюжок закінчиться числом 1. Отож, якщо початкове число — 13, то послідовність виглядатиме ось так: *13, 40, 20, 10, 5, 16, 8, 4, 2, 1.* Другий елемент — $13*3+1$ дорівнює 40. 40 поділити на 2 дорівнює 20, і так далі. Ланцюжок налічує 10 елементів.

Потрібно дізнатися ось що: **серед усіх початкових чисел між 1 і 100, скільки ланцюжків мають довжину більшу ніж 15?** Спочатку напишемо функцію, що створює ланцюжок:

```
chain :: Integral a => a -> [a]
chain 1 = [1]
```

```
chain n
    | even n =  n:chain (n `div` 2)
    | odd n  =  n:chain (n*3 + 1)
```

Оскільки ланцюжок закінчується на 1, це — крайовий випадок. Маємо доволі стандартну рекурсивну функцію.

```
ghci> chain 10
[10,5,16,8,4,2,1]
ghci> chain 1
[1]
ghci> chain 30
[30,15,46,23,70,35,106,53,160,80,40,20,10,5,16,8,4,2,1]
```

Ура! Здається, все працює. А тепер функція, що відповість на наше запитання:

```
numLongChains :: Int
numLongChains = length (filter isLong (map chain [1..100]))
    where isLong xs = length xs > 15
```

Ми відображаємо за допомогою `chain` по `[1..100]`, щоб отримати список ланцюжків, які самі по собі теж представлено як списки. Тоді фільтруємо ланцюжки предикатом, який тільки перевіряє чи довжина списку більша ніж 15. Коли фільтрування закінчилося, ми рахуємо скільки в кінцевому списку залишилося ланцюжків.

Примітка: Ця функція має тип `numLongChains :: Int`, тому що `length` повертає `Int` замість `Num a` (так склалося історично). Якби ми хотіли повернути загальніше `Num a`, можна було б застосувати `fromIntegral` до отриманої довжини.

За допомогою `map` можна робити речі на кшталт `map (*) [0..]`, хоча б для того, щоб продемонструвати, як працює каріювання, і як (частково застосовані) функції є реальними величинами, що їх можна передати іншим функціям чи вкласти до списку (єдине, що їх не можна перетворити на рядки). Наразі ми відображали за допомогою функцій одного аргументу по списках. Якот `map (*2) [0..]`, яка виводить список типу `Num a => [a]`. Але можна без жодних проблем виконати й `map (*) [0..]`. У цьому випадку число у списку застосовується до функції `*`, яка має тип `Num a => a -> a -> a`. Застосування лише одного параметра до функції, що приймає два параметри, повертає функцію, яка бере один параметр. Якщо ми відобразимо за допомогою `*` по

списку `[0..]`, то отримаємо список функцій, що приймають лише один параметр. Тобто `Num a => [a -> a]`. `map (*) [0..]` створює список, що його ми б отримали, якби написали `[(0*),(1*),(2*),(3*),(4*),(5*)...`.

```
ghci> let listOfFuns = map (*) [0..]
ghci> (listOfFuns !! 4) 5
20
```

Якщо ми візьмемо елемент з індексом `4` з нашого списку, то отримаємо функцію еквівалентну `(4*)`. А тоді просто застосуємо її до `5`. Це є те саме, що й написати `(4*) 5`, або ж просто `4 * 5`.

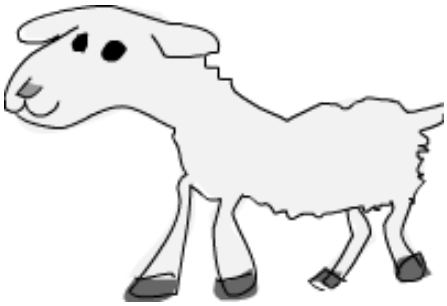
6.4 Лямбди

Лямбди — це по суті безіменні функції, що їх ми використовуємо, коли потребуємо якусь функцію тільки один раз. Зазвичай лямбду пишуть тільки для того, щоб передати її якійсь функції вищого порядку. Щоб створити лямбду, напишіть `\` (якщо дуже добре придивитися, то побачимо в цьому символі «спину» грецької літери лямбди), а тоді — параметри, відокремивши їх пробілами. Далі йде `->` і тіло функції. Зазвичай лямбди беруть у дужки, інакше вони поширяться на все, що стоїть праворуч.

Якщо ви піднімете погляд трохи догори, то побачите, що ми використали зв'язку *where* у функції `numLongChains`, щоб створити функцію `isLong`. Все, що ми із нею робимо, — це передаємо її функції `filter`. Ну а тепер, замість цього використаймо лямбду:

```
numLongChains :: Int
numLongChains = length (filter (\xs -> length xs > 15) (map chain [1..100]))
```

Лямбди — це вирази, тому їх можна ось так просто взяти і передати. Вираз `(\xs -> length xs > 15)` повертає функцію, що каже, чи довжина переданого їй списку більша за 15.



Люди, які не зовсім розуміють, як працює каріювання та часткове застосування, часто використовують лямбди недоречно.

До прикладу, вирази `map (+3) [1,6,3,2]` і `map (\x -> x + 3) [1,6,3,2]` рівнозначні, адже і `(+3)`, і `(\x -> x + 3)` — це функції, що беруть число і додають до нього 3. Без сумніву, тут нема сенсу використовувати лямбду, адже часткове застосування набагато читабельніше.

Як і звичайні функції, лямбди приймають яку завгодно кількість параметрів:

```
ghci> zipWith (\a b -> (a * 30 + 3) / b) [5,4,3,2,1] [1,2,3,4,5]
[153.0,61.5,31.0,15.75,6.6]
```

В лямбдах можна зіставляти із взірцем, як і звичайні функції. Єдина відмінність полягає у тому, що для одного параметра не можна означити кілька взірців, як-от `[]` і `(x:xs)` для одного параметру-списку, так, щоб в разі незіставлення із першим було здійснено перехід до наступного взірця. Якщо у лямбді не пройде зіставлення із взірцем, трапиться помилка періоду виконання. Тому будьте уважні, коли зіставляєте із взірцем у лямбдах!

```
ghci> map (\(a,b) -> a + b) [(1,2),(3,5),(6,3),(2,6),(2,5)]
[3,8,9,8,7]
```

Лямбди зазвичай беруть у дужки, хіба що ми хочемо поширити їхню дію на все, що знаходиться праворуч. Цікаво, що через спосіб, у який лямбди каріюються за замовчуванням, ось ці дві функції рівнозначні:

```
addThree :: Num a => a -> a -> a -> a
addThree x y z = x + y + z
```

```
addThree :: Num a => a -> a -> a -> a
addThree = \x -> \y -> \z -> x + y + z
```

Якщо ми ось так означимо функцію, то зрозуміло, чому оголошення типу виглядає саме так. В означенні типу та рівнянні є три `->`. Але звісно, що перший спосіб написання функцій набагато читабельніший, а другий — це просто такий собі викрутас, щоб показати, що ж таке каріювання.

Але часом такий запис стає у пригоді. На мій погляд, функція `flip` найчитабельніша, якщо означена таким чином:

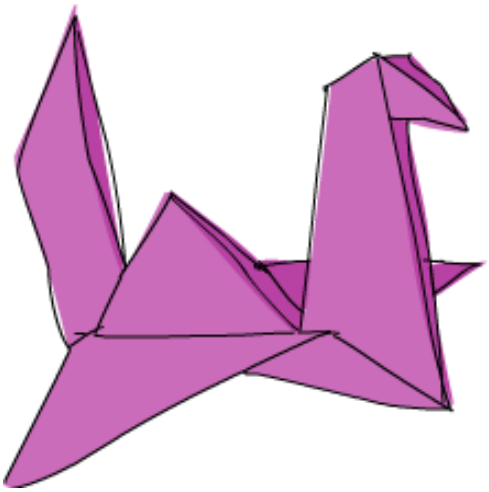
```
flip' :: (a -> b -> c) -> b -> a -> c
flip' f = \x y -> f y x
```

Навіть якщо ми написали те саме, що й `flip' f x y = f y x`, з версії із лямбдою легше зрозуміти, що ця функція створюватиме нову функцію (здебільшого). Найчастіше функцію `flip` використовують ось так: їй подають лише параметр-функцію, а тоді передають отриману функцію `map` або `filter`. Отож, використовуйте лямбди, коли хочете чітко вказати користувачеві вашого коду, що цю функцію написано для застосування саме в такий спосіб, себто, — частково застосовувати її, а тоді передати результат іншій функції як параметр.

6.5 Згортком і батька легше бити

Коли ми мали справу з рекурсією, то помітили одну особливість рекурсивних функцій, що працюють зі списками. Зазвичай, порожній список був крайовим випадком. Ми вводили взірець `x:xs`, а тоді здійснювали якусь дію, що опрацьовувала один елемент і решту списку. Виявляться, що це є добре відома ідіома, і тому було створено декілька вельми корисних функцій, які її інкапсулюють. Ці функції називаються згортками. Вони схожі на функцію `map`, тільки вони зводять список до одного значення.

Згорток бере бінарну функцію, початкове значення (я називаю його накопичувач) і список, і згортає його. Сама бінарна функція приймає два параметри. Бінарна функція викликається із накопичувачем і першим (або останнім) елементом списку, і в результаті обрахунку отримується новий накопичувач. Тоді той новий накопичувач і новий перший (або останній) елемент знову передаються як параметри бінарній функції, знову отримується новий накопичувач і так далі. Коли ми перейдемо цілий список, залишиться тільки накопичувач — він і буде результатом, на який перетворився увесь список в наслідок операції згортання.



Спершу погляньмо на функцію `foldl`, так званий лівий згорток. Вона згортає список із лівого боку. Бінарна функція застосовується до початкового значення і голови списку. В результаті отримується нове значення накопичувача, яке разом із наступним елементом передається бінарній функції, і так далі.

Спробуймо знову реалізувати `sum`, але цього разу використаймо згорток замість явної рекурсії.

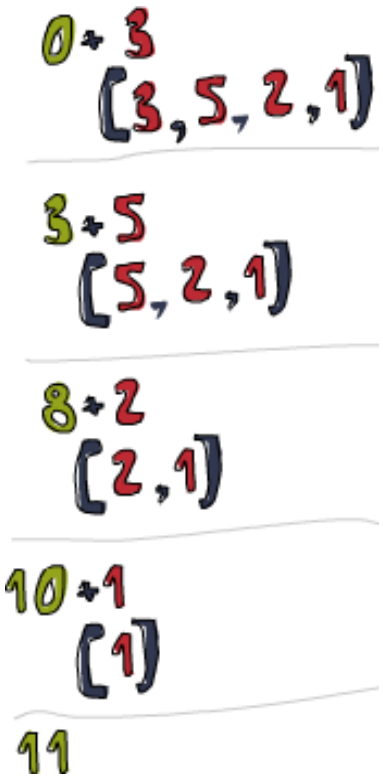
```
sum' :: Num a => [a] -> a
```

```
sum' xs = foldl (\acc x -> acc + x) 0 xs
```

Тестуємо, один-два-три:

```
ghci> sum' [3,5,2,1]
11
```

А тепер розгляньмо детально, як працює цей згорток. `\acc x -> acc + x` — це бінарна функція. `0` — це початкове значення, а `xs` — список, який треба згорнути. Спочатку `0` використовується як параметр `acc` бінарної функції, а `3` — як параметр `x` (так званий «поточний елемент»). `0 + 3` після обрахунку приймає значення `3` і стає новим значенням накопичувача, так би мовити. Далі це `3` використовується як значення накопичувача, а `5` — як поточний елемент, і `8` стає новим значенням накопичувача. Рухаємося далі: `8` — це значення накопичувача, `2` — поточний елемент, а нове значення накопичувача після обрахунку — `10`. Врешті-решт, `10` використовується як значення накопичувача, а `1` — як поточний елемент, що генерує `11`. Вітаємо, ви вперше згорнули список!



Професійно виконана діаграма ліворуч зображає, як відбувається згорток, крок за кроком (день за днем!). Зеленкувато-коричневе число — це значення накопичувача. Бачите, як з лівого боку накопичувач немовби поглинає список. Гам-гам-гам-гам! Якщо врахувати, що функції каріюються, то можна записати цю реалізацію ще стисліше. Ось так:

```
sum' :: Num a => [a] -> a
sum' = foldl (+) 0
```

Лямбда-функція `(\acc x -> acc + x)` — те саме що і `(+)`. Можемо вилучити параметр `xs`, адже виклик `foldl (+) 0` поверне функцію, що приймає список. Загалом, якщо ви маєте функцію на кшталт `foo a = bar b a`, то можете переписати її як `foo = bar b` завдяки каріюванню.

Добре. Перед тим як перейти до правих згортків, реалізуймо ще одну функцію із залученням лівого згортка. Я певен, що вам усім відомо, що `elem` перевіряє, чи є якесь значення в списку чи немає, тож я не буду вам ще раз це пояснюва-

ти (ой, не вийшло — пояснив-таки!). Реалізуймо `elem` із використанням лівого згортка.

```
elem' :: Eq a => a -> [a] -> Bool
elem' y ys = foldl (\acc x -> if x == y then True else acc) False ys
```

Так, так, так... І що ми тут маємо? Початкове значення і накопичувач — це булеві значення. Коли маєте справу із згортками, тип значення накопичувача і тип кінцевого результату завжди однакові. Запам'ятайте: якщо ви не знаєте, *що* використати як початкове значення, пригадайте собі це правило. Почнімо із `False`. Використати `False` як початкове значення — досить розумно. Ми припускаємо, що шуканого значення немає в списку. До того ж, якщо ми викличемо згорток по порожньому списку, то отримаємо в результаті просто початкове значення. Далі, ми перевіряємо, чи поточний елемент є елементом, що його ми шукаємо, чи ні. Якщо так, міняємо значення накопичувача на `True`. Якщо ні, то просто залишаємо накопичувач без змін — якщо перед тим він був `False`, то таким він і залишається, адже поточний елемент не є ним; якщо він був `True`, не чіпаємо — лишаємо як є також.

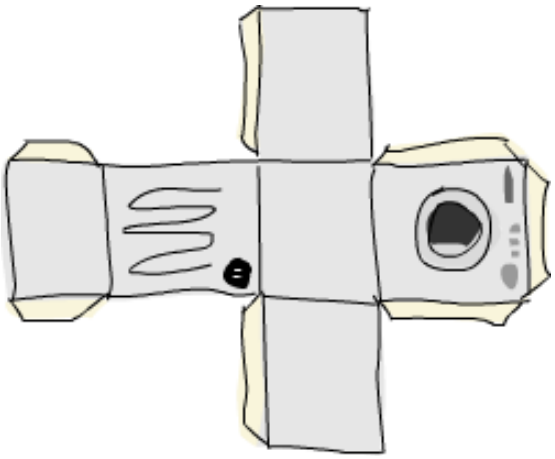
Правий згорток, `foldr`, працює подібно до лівого згортка, тільки накопичувач поїдає значення з правого боку. На додачу, бінарна функція лівого згортка бере накопичувач як перший параметр, а поточне значення — як другий параметр (отож `\acc x -> ...`), а бінарна функція правого згортка має поточне значення як перший параметр і накопичувач — як другий (отож `\x acc -> ...`). Логічно, що правий згорток має накопичувач справа, адже процес згортання іде з правого боку.

Значення накопичувача (а отож і результату) може належати до якого завгодно типу. Це може бути число, булеве значення чи навіть новий список. Ми реалізуємо функцію відображення `map` використовуючи правий згорток. Накопичувачем буде список. Ми накопичуватимемо перетворені елементи в списку, елемент за елементом. Очевидно, що початковим елементом буде порожній список.

```
map' :: (a -> b) -> [a] -> [b]
map' f xs = foldr (\x acc -> f x : acc) [] xs
```

Ми відображаємо за допомогою `(+3)` по `[1,2,3]`, і підходимо до списку з правого боку. Ми беремо останній елемент, тобто `3`, і застосовуємо до нього функцію. Отримуємо `6`. Тоді додаємо те `6` до початку накопичувача, тобто `[]`. `6:[]` дорівнює `[6]`, і це тепер є новим накопичувачем. Застосовуємо `(+3)` до `2`, отримуємо `5` і приєднуємо (за допомогою `:`) і цей результат до накопичувача. Накопичувач тепер — `[5,6]`. Застосовуємо `(+3)` до `1` і приєднуємо результат до накопичувача також. Отож кінцеве значення — `[4,5,6]`.

Звісно, що цю функцію можна було б реалізувати і за допомогою лівого згортка. Тоді б ми мали `map' f xs = foldl (\acc x -> acc ++ [f x]) [] xs`, але справа в тому, що функція `++` набагато затратніша, ніж `:`, тому коли ми створюємо нові списки зі списку, то зазвичай використовуємо праві згортки.



Лівий згорток це є те саме, що правий згорток по розвернутому списку, і навпаки. Інколи навіть і розвертати нічого не треба. Функцію `sum` можна успішно реалізувати як лівим, так і правим згортком. Єдина, значна, відмінність полягає у тому, що праві згортки працюють із нескінченними списками, тоді як ліві — ні![†] Кажучи по-простому, якщо ви візьмете нескінченний список і згорнете його справа, то врешті-решт дійдете до початку списку. А от якщо візьмете нескінченний список і спробуєте згорнути його зліва, то ніколи не дійдете кінця!

Згортки можна використовувати в реалізаціях функцій, які один раз проходять списком, елемент за елементом, і повертають щось на основі цієї прогулянки. Якщо ви захочете пройти список, аби щось вернути, швидше за все вам треба скористатися згортком. Ось чому згортки, разом із відображеннями та фільтрами, є страшенно корисними функціями у функційному програмуванні.

Функції `foldl1` і `foldr1` працюють подібно до `foldl` і `foldr`, от тільки їм не потрібно явно передавати початкове значення. Вони використовують перший (або останній) елемент списку як початкове значення, а тоді починають згорток із сусіднього елемента. Озброївшись цим, можемо реалізувати функцію `sum` ось так: `sum = foldl1 (+)`. Оскільки ці функції очікують на списки із принаймні одним елементом, `foldl1` і `foldr1` спричиняють помилки періоду виконання, якщо їх застосувати до порожніх списків. Водночас, `foldl` і `foldr` чудово працюють із порожніми списками. Створюючи згорток, подумайте про те, як він поводиться із порожнім списком. Якщо ситуація, коли функції пода-

[†] Під «працюють» мається на увазі оце: обчислення правого згортка із лінивою функцією по нескінченному списку (наприклад, `foldr (\x y -> x) 0 [1..]`) завершується, а лівого — ні. Завдяки цьому, правий згорток можна використовувати в побудові нескінченних структур даних (які потім споживатимуться ліниво, наприклад `take 6 $ foldr (\x y -> [x,x]++y) [] [1..]`). Із завзятими функціями, обчислення не завершуються як із правими, так із лівими згортками.

ється порожній список, не має сенсу (себто, вкрай безглузда, помилкова ситуація), тоді використовуйте `foldl1` чи `foldr1`, щоб цю функцію реалізувати.

Щоб ви переконалися, що згортки справді могутні, реалізуймо низку стандартних бібліотечних функцій за їх допомогою:

```
maximum' :: Ord a => [a] -> a
maximum' = foldr1 (\x acc -> if x > acc then x else acc)

reverse' :: [a] -> [a]
reverse' = foldl (\acc x -> x : acc) []

product' :: Num a => [a] -> a
product' = foldr1 (*)

filter' :: (a -> Bool) -> [a] -> [a]
filter' p = foldr (\x acc -> if p x then x : acc else acc) []

head' :: [a] -> a
head' = foldr1 (\x _ -> x)

last' :: [a] -> a
last' = foldl1 (\_ x -> x)
```

`head` краще реалізувати за допомогою зіставлення із взірцем, але те ж саме вдасться зробити і зі згортком. Я гадаю, що наше означення `reverse'` досить розумне. Ми беремо початкове значення порожнього списку, тоді підходимо до списку зліва, і приєднуємо до нього накопичувача. Врешті-решт, побудується розвернений список. `\acc x -> x : acc` дещо схожа на функцію `:`, тільки параметри обернені місцями. Також можна було б означити наше розвернення як `foldl (flip (:)) []`.

Праві та ліві згортки можна описати ще й так: маємо правий згорток, бінарну функцію `f` і початкове значення `z`. Якщо ми згортаємо список `[3,4,5,6]` справа, то по суті робимо ось що: `f 3 (f 4 (f 5 (f 6 z)))`. `f` викликається із останнім елементом у списку та накопичувачем. Отримане значення надається функції як накопичувач разом із передостаннім значенням, і так далі. Якщо ми покладемо `f` рівним `+`, а початкове значення накопичувача — `0`, то вираз перетвориться на `3 + (4 + (5 + (6 + 0)))`. А якщо напишемо `+` як префіксну функцію, то цей вираз виглядатиме отак: `(+) 3 ((+) 4 ((+) 5 ((+) 6 0)))`. Так само, згортання цього списку зліва з бінарною функцією `g` і накопичувачем `z` рівнозначне `g (g (g (g z 3) 4) 5) 6`. Якщо ми скористаємося `flip (:)` як бінарною фун-

кцією та `[]` як накопичувачем (тобто, будемо розвертати список), то отримаємо `flip (:) (flip (:) (flip (:) (flip (:) [] 3) 4) 5) 6`. І звісно, що, якщо ви вирахуєте цей вираз, то отримаєте `[6,5,4,3]`.

`scanl` і `scanr` схожі на `foldl` і `foldr`, тільки вони звітують про проміжні стани накопичувача у формі списку. Існують також `scanl1` і `scanr1`, аналоги `foldl1` і `foldr1`.

```
ghci> scanl (+) 0 [3,5,2,1]
[0,3,8,10,11]
ghci> scanr (+) 0 [3,5,2,1]
[11,8,3,1,0]
ghci> scanl1 (\acc x -> if x > acc then x else acc) [3,4,5,3,7,9,2,1]
[3,4,5,5,7,9,9,9]
ghci> scanl (flip (:)) [] [3,2,1]
[[],[3],[2,3],[1,2,3]]
```

Якщо використати `scanl`, то остаточний результат буде в останньому елементі отриманого списку, тоді як `scanr` помістить результат у голову.

Скани іноді використовують для нагляду за процесом роботи функцій в дебаг режимі, а в реліз версії тих функцій просто заміняють скан на згорток. Відповімо собі на оце запитаннячко: **Скільки елементів має увійти до суми коренів усіх натуральних чисел, щоб вона перевищила 1000?** Щоб отримати усі корені, ми просто пишемо `map sqrt [1..]`. А щоб отримати суму, можемо зробити згорток. Але оскільки нас цікавить саме перебіг процедури обчислення суми, ми використаємо замість згортка скан. Після завершення сканування побачимо, скільки сум мають значення менше 1000. Перша сума міститиме лише один доданок — 1. Друга дорівнюватиме $1 + \sqrt{2}$. Третя — значення попередньої суми плюс квадратний корінь від 3, тобто $1 + \sqrt{2} + \sqrt{3}$. Якщо існує X сум менших за 1000, то буде потрібно $X + 1$ елементів, щоб сума перевищила 1000.

```
sqrtSums :: Int
sqrtSums = length (takeWhile (<1000) (scanl1 (+) (map sqrt [1..]))) + 1
```

```
ghci> sqrtSums
131
ghci> sum (map sqrt [1..131])
1005.0942035344083
ghci> sum (map sqrt [1..130])
993.6486803921487
```

Тут ми використовуємо `takeWhile` замість `filter`, тому що `filter` не працює із нескінченними списками. Ми-то знаємо, що список на вході висхі-

дний, а `filter` — ні. Тому ми і обрізаємо результат сканування за допомогою `takeWhile`, як тільки знайдеться перша сума, більша за 1000.

6.6 Доларове застосування функцій

Гаразд, далі розглянемо функцію `$` або ж *застосування функції*. Спершу подивимось, як вона означена:

```
($) :: (a -> b) -> a -> b
f $ x = f x
```

Що це в біса таке? Що за непотрібний оператор? Це всього лиш застосування функції! Ну так, але не зовсім! Якщо звичайне застосування функції (коли два елементи відокремлені пробілом) має найвищий пріоритет, то оператор `$` має найнижчий пріоритет. До того ж, застосування функції пробілом — лівоасоціативне, (тобто, `f a b c` є те ж саме, що `((f a) b) c)`, а застосування функції за допомогою `$` — правоасоціативне.

Звучить чудово, але яка нам із цього користь? Здебільшого, ця функція використовується для зручності — вона допомагає нам зменшити кількість дужок. Ось, наприклад, вираз `sum (map sqrt [1..130])`. Оскільки `$` має низький пріоритет, ми можемо переписати цей вираз як `sum $ map sqrt [1..130]`, без дужок! Із `$` все просто — вираз праворуч від `$` застосовується як параметр до функції, що стоїть ліворуч. А як щодо `sqrt 3 + 4 + 9`? Він додає докупи 9, 4 і квадратний корінь від 3. Якщо нам потрібен квадратний корінь від *3 + 4 + 9*, мусимо написати `sqrt (3 + 4 + 9)`. А якщо скористаємося `$`, то зможемо написати `sqrt $ 3 + 4 + 9`, адже `$` має найменший пріоритет з усіх операторів. Можете уявити собі, що `$` — це те саме, що спочатку відкрити дужки, а тоді закрити їх справа, в кінці виразу.

А як щодо `sum (filter (>10) (map (*2) [2..10]))`? Ну, оскільки `$` правоасоціативний, то `f (g (z x))` дорівнює `f $ g $ z x`. Тому можемо переписати `sum (filter (>10) (map (*2) [2..10]))` як `sum $ filter (>10) $ map (*2) [2..10]`.

Але `$` не тільки знищує дужки в виразах, а ще й дозволяє поводитись із застосуванням функції як із звичайною функцією. Таким чином, ми можемо, наприклад, відобразити за допомогою застосування функції список функцій.

```
ghci> map ($ 3) [(4+), (10*), (^2), sqrt]
[7.0,30.0,9.0,1.7320508075688772]
```

6.7 Композиція функцій

У математиці композиція функцій означена ось так: $(f \circ g)(x) = f\,(g(x))$, що означає: композиція двох функцій створює нову функцію, яка — якщо її викликати із параметром x — є рівнозначною виклику g із параметром x, а тоді застосування f до отриманого результату.

У Хаскелі композиція функцій означає майже те саме. Композиція функцій виконується за допомогою функції `(.)`, яка означена ось так:

```
(.) :: (b -> c) -> (a -> b) -> a -> c
f . g = \x -> f (g x)
```

Зверніть увагу на оголошення типу. `f` мусить приймати як параметр значення, що має такий самий тип як тип значення-результату `g`. Отже, функція, що ми її отримуємо в результаті композиції, приймає параметр такого ж типу як параметр `g` і повертає значення такого ж типу, як результат `f`. Вираз `negate . (*3)` повертає функцію, що бере число, множить його на 3, а тоді міняє його знак.

Композиція функцій використовується для створення функції «на ходу» для передачі їх іншим функціям. Звісно, що для цього можна використати лямбди, але композицію функцій легше читати і розуміти. До прикладу, ми маємо список чисел, які хочемо перетворити на від'ємні числа. Один із шляхів: отримати абсолютне значення кожного числа, а тоді зробити його від'ємним, ось так:

```
ghci> map (\x -> negate (abs x)) [5,-3,-6,7,-3,2,-19,24]
[-5,-3,-6,-7,-3,-2,-19,-24]
```

Зверніть увагу на лямбду і на те, наскільки вона схожа на результат композиції функцій. За допомогою композиції функцій можна переписати це ось так:

```
ghci> map (negate . abs) [5,-3,-6,7,-3,2,-19,24]
[-5,-3,-6,-7,-3,-2,-19,-24]
```

Чудово! Композиція функцій правоасоціативна, тому можна компонувати багато функцій одночасно. Вираз `f (g (z x))` рівнозначний `(f . g . z) x`. Пам'ятаючи про це, можемо перетворити

```
ghci> map (\xs -> negate (sum (tail xs))) [[1..5],[3..6],[1..7]]
[-14,-15,-27]
```

на

```
ghci> map (negate . sum . tail) [[1..5],[3..6],[1..7]]
[-14,-15,-27]
```

Але як бути з функціями, що приймають кілька параметрів? Якщо ми хочемо використати їх у композиції функцій, то здебільшого мусимо частково застосувати їх так, щоб кожна функція приймала тільки один параметр. `sum (replicate 5 (max 6.7 8.9))` можна переписати як `(sum . replicate 5 . max 6.7) 8.9` або як `sum . replicate 5 . max 6.7 $ 8.9`. Ось що тут відбувається: створюється функція, що приймає те, що приймає `max 6.7`, і застосовує до результату обрахунку `max 6.7` функцію `replicate 5`. В свою чергу, результат, що його повертає ця композиція, передається до функції `sum`, яка вираховує суму. Цей великий потрійний композит викликається із `8.9`. Але зазвичай це все читається ось як: застосовуємо `max 6.7` до `8.9`, тоді застосовуємо до результату `replicate 5`, а тоді застосовуємо до того `sum`. Якщо ви хочете переписати вираз із купою дужок за допомогою композиції функцій, то спершу поставте останній параметр найглибше вкладеної функції після `$`, а тоді скомпонуйте виклики усіх інших функцій, написавши їх без відповідних останніх параметрів (в кожної свій) та поставивши між ними (функціями) крапки.

```
replicate 100 (product (map (*3) (zipWith max [1,2,3,4,5] [4,5,6,7,8])))
```

можна записати як

```
replicate 100 . product . map (*3) . zipWith max [1,2,3,4,5] $ [4,5,6,7,8]
```

Найімовірніше, що якщо вираз закінчується трьома дужками, то він після перетворення на композицію функцій матиме три оператори композиції.

Композицію функцій використовують і для того, щоб означити функції у так званий безточковий спосіб.[†] Ось, до прикладу, функція, яку ми написали раніше:

[†] Термін «безточкове означення» походить з топології, галузі математики, яка працює із

```
sum' :: Num a => [a] -> a
sum' xs = foldl (+) 0 xs
```

`xs` стирчіть праворуч з обох сторін рівняння. Дякуючи каріюванню, можемо вилучити `xs` з обох боків, оскільки виклик `foldl (+) 0` створює функцію, що приймає список. Запис функції як `sum' = foldl (+) 0` називається записом на безточковий лад. А тепер — нова задача: як записати у безкрапковому стилі оце:

```
fn x = ceiling (negate (tan (cos (max 50 x))))
```

Не можна просто позбутися `x` праворуч з обох боків. Після `x` у тілі функції стоять дужки. `cos (max 50)` не має сенсу. Неможливо отримати косинус функції. Проте `fn` можна переписати із використанням композиції функцій, а тоді перевести у безкрапкову форму.

```
fn = ceiling . negate . tan . cos . max 50
```

Прекрасно! Зазвичай безточковий стиль лаконічніший і читабельніший, оскільки він змушує нас думати про функції, які у нас є, і про те, які функції з цих функцій можна збудувати за допомогою композиції, а не про дані і те, як вони течуть по програмі. Беремо прості функції та склеюємо їх докупи композицією, щоб утворити складніші функції. Але часом функція, записана у безточковий спосіб, буде менш читабельною, якщо вона занадто складна. Ось чому небажано створювати довгі ланцюжки композицій функцій, хоча — каюсь! — я частенько цим грішу. Найкраще використовувати зв'язки *let*, щоб поіменувати проміжні результати чи розділити проблему на задачі і підзадачі, а тоді складати їх докупи, щоб той, хто читатиме функцію в майбутньому, швидше зміг зрозуміти, як вона працює. Ліпше робити так, ніж утворювати довжелезні ланцюжки композицій.

У розділі про відображення і фільтри ми розв'язали задачу про те, як отримати суму усіх непарних квадратів, менших за 10000. Ось як виглядатиме розв'язок, якщо його записати однією функцією:

```
oddSquareSum :: Integer
oddSquareSum = sum (takeWhile (<10000) (filter odd (map (^2) [1..])))
```

Оскільки я страшенно люблю композицію функцій, то я б мабуть написав це ось так:

просторами, що складаються з точок, і функціями, що відображають одні простори в інші. В безточкових означеннях точки не згадуються — ці означення є написані виключно мовою просторів.

```
oddSquareSum :: Integer
oddSquareSum = sum . takeWhile (<10000) . filter odd . map (^2) $ [1..]
```

Але якби хтось інший мав потім читати мій код, я б написав так:

```
oddSquareSum :: Integer
oddSquareSum =
    let oddSquares = filter odd $ map (^2) [1..]
        belowLimit = takeWhile (<10000) oddSquares
    in  sum belowLimit
```

Так, я б не виграв змагання з гольф-програмування[†], але полегшив би життя тому, хто читатиме цей код після мене.

[†] «Гольф-програмування» — програмування для розваги, де метрикою успіху, окрім коректності, є довжина програми. Метою гри є написання найкоротшої програми, яка втілює заданий алгоритм.

Покажчик

(.)
функція **(.)**, 22
Collatz sequences
послідовності Коллатца, 11
Show
типоклас **Show**, 4
accumulator
накопичувач, 15
ascending order
висхідний порядок, 21
bug
вада, 2
clever trick
спритний викрутас, 1
command prompt
командне запрошення; командне про́шу, 4
curried function
карійована функція, 1
debug version
дебаг версія, 20
descending order
порядок спадання, 10
edge case
граничний випадок; крайовий випадок, 12
edge condition
гранична умова; крайова умова, 6
filtered list
профільтрований список, 10
filter
функція **filter**, 8

final result
остаточний результат, 20
flip
переверт, 7
foldl1
функція **foldl1**, 18
foldl
функція **foldl**, 15
foldr1
функція **foldr1**, 18
foldr
функція **foldr**, 17
fold
згорток, 15
function f takes argument x
функція f бере аргумент x; функція f приймає аргумент x, 18
function f takes parameter x
функція f бере параметр x; функція f приймає параметр x, 18
function application
застосування функції, 2, 21
function body
тіло функції, 5
function composition
композиція функцій, 22
golf programming
гольф-програмування, 25
higher-order function
функція вищого порядку, 1
inferred type
виведений тип, 6
innermost function
найглибше вкладена функція, 23
joining function
сполучна функція, 6
lambda
лямбда, 13
left fold
лівий згорток, 15
left scan
лівий скан, 20

list of elements of type A
список з елементів типу A, 6
mapping
відображення, 10
map
функція **map**, 8, 17
nested loops
вкладені цикли, 10
operand
операнд, 3
partially applied function
частково застосована функція, 2
pattern (common programming idiom)
ідіома (поширений прийом чи розв'язок в програмуванні), 15
pivot
стрижень, 9
point-free style
безточковий стиль; безточковий лад; безточковий спосіб, 23
predicate
предикат, 8
prepend to a list (tuple)
приєднати до списку (кортежу), 17, 19
prompt (command prompt)
запрошення; про́шу (командне запрошення; командне про́шу), 4
quicksort algorithm
алгоритм швидкого сортування, 9
release version
реліз версія, 20
return type
тип результату, 22
return value (of a function)
значення-результат (функції), 1, 22
reverse (e.g., of a list)
розвернення (напр., списку), 18, 19
right fold
правий згорток, 17
right scan
правий скан, 20
runtime error
помилка (періоду) виконання, 14

scanl
функція **scanl**, 20
scanr
функція **scanr**, 20
scan
скан, 20
sections of an infix function
розтин інфіксної функції, 3
sections
розтин, 3
space
пробіл, 2
starting value
початкове значення, 15
string representation
рядкове представлення, 4
takeWhile
функція **takeWhile**, 10
textual representation
текстове представлення; текстовий вигляд, 4
to abstract
абстрагувати, 7
to compose
компонувати, 23
to join
сполучати, 6
to negate
змінити знак, 22
to reduce
зводити, 15
to seed (a function, a random number generator)
зарядити (функцію, генератор випадкових чисел), 2
x evaluates to y
x після обчислення приймає значення y; x знаходить значення y, 9
x після обчислення приймає значення y; x знаходить значення y
x evaluates to y, 9
абстрагувати
to abstract, 7
алгоритм швидкого сортування
quicksort algorithm, 9

безточковий стиль; безточковий лад; безточковий спосіб
point-free style, 23
вада
bug, 2
виведений тип
inferred type, 6
висхідний порядок
ascending order, 21
вкладені цикли
nested loops, 10
відображення
mapping, 10
гольф-програмування
golf programming, 25
гранична умова; крайова умова
edge condition, 6
граничний випадок; крайовий випадок
edge case, 12
дебаг версія
debug version, 20
запрошення; про́шу (командне запрошення; командне про́шу)
prompt (command prompt), 4
зарядити (функцію, генератор випадкових чисел)
to seed (a function, a random number generator), 2
застосування функції
function application, 2, 21
зводити
to reduce, 15
згорток
fold, 15
змінити знак
to negate, 22
значення-результат (функції)
return value (of a function), 1, 22
карійована функція
curried function, 1
командне запрошення; командне про́шу
command prompt, 4
композиція функцій
function composition, 22

компонувати
to compose, 23
лямбда
lambda, 13
лівий згорток
left fold, 15
лівий скан
left scan, 20
найглибше вкладена функція
innermost function, 23
накопичувач
accumulator, 15
операнд
operand, 3
остаточний результат
final result, 20
переверт
flip, 7
помилка (періоду) виконання
runtime error, 14
порядок спадання
descending order, 10
послідовності Коллатца
Collatz sequences, 11
початкове значення
starting value, 15
правий згорток
right fold, 17
правий скан
right scan, 20
предикат
predicate, 8
приєднати до списку (кортежу)
prepend to a list (tuple), 17, 19
пробіл
space, 2
профільтрований список
filtered list, 10
реліз версія
release version, 20

розвернення (напр., списку)
reverse (e.g., of a list), 18, 19
розтин інфіксної функції
sections of an infix function, 3
розтин
sections, 3
рядкове представлення
string representation, 4
скан
scan, 20
список з елементів типу A
list of elements of type A, 6
сполучати
to join, 6
сполучна функція
joining function, 6
спритний викрутас
clever trick, 1
стрижень
pivot, 9
текстове представлення; текстовий вигляд
textual representation, 4
тип результату
return type, 22
типоклас *Show*, 4
тіло функції
function body, 5
функція f бере аргумент x; функція f приймає аргумент x
function f takes argument x, 18
функція f бере параметр x; функція f приймає параметр x
function f takes parameter x, 18
функція *(.)*, 22
функція *filter*, 8
функція *foldl1*, 18
функція *foldl*, 15
функція *foldr1*, 18
функція *foldr*, 17
функція *map*, 8, 17
функція *scanl*, 20
функція *scanr*, 20

функція *takeWhile*, 10
функція вищого порядку
    higher-order function, 1
частково застосована функція
    partially applied function, 2
ідіома (поширений прийом чи розв'язок в програмуванні)
    pattern (common programming idiom), 15