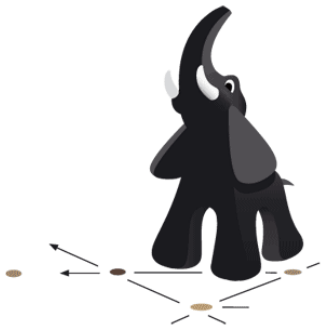
Вивчить собі Хаскела на велике щастя!

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



2017-05-21T00:06:46Z
Версія v4.7-54-gda41cf2

Зміст

Розділ 10

Функційне розв'язання задач

Переклад українською Ганни Лелів

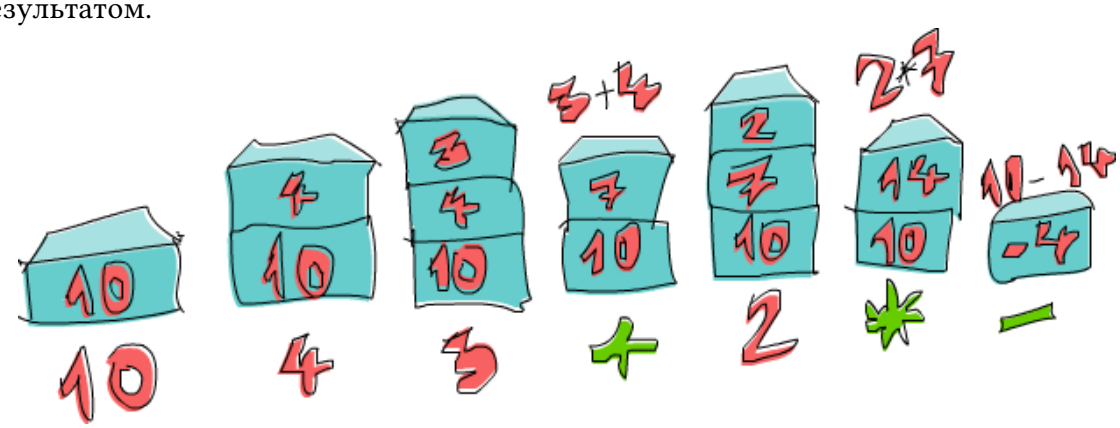
У цьому розділі ми розглянемо кілька цікавих завдань і поміркуємо функційно, аби розв'язати їх якомога елегантніше. Ми не будемо вводити ніяких нових концепцій, а просто пограємо трохи новою хаскельною мускулатурою, яку ми натренували раніше, та відшліфовуватимемо наші набуті навички кодування. У кожному підрозділі ми матимемо справу з новим завданням. Спершу ми його опишемо, а тоді **визначимо** [to find out], який є найкращій (або принаймні, найменш поганий) спосіб його розв'язання.

10.1 Калькулятор зворотного польського запису

Коли у школі записують математичні вирази, то зазвичай роблять це інфіксно. До прикладу, пишуть `10 - (4 + 3) * 2`. `+`, `*` і `-` — це інфіксні оператори, такі ж як інфіксні функції у Хаскелі (`+`, `` `elem` `` і так далі.). Це зручно, бо ми — люди — можемо подумки легко провести синтаксичний аналіз, просто поглянувши на ці вирази. З іншого боку, цей підхід має недолік — треба деякі речі брати в дужки, щоб був правильний порядок обчислень.

Зворотній польський запис — це ще один спосіб запису математичних виразів. Спочатку він виглядатиме трохи дивно, але насправді він дуже простий і приємний до використання, тому що не потрібно брати нічого в дужки, і тому ним зручно користуватися при розрахунках на калькуляторі. Хоча сучасні калькулятори здебільшого використовують інфіксний запис, дехто досі обожнює калькулятори зворотного польського запису (ЗПЗ-калькулятори). Ось як попередній інфіксний вираз виглядає у ЗПЗ: `10 4 3 + 2 * -`. Як вирахувати результат цього виразу? Згадайте стек. Ви читаєте вираз зліва направо. Як тільки стикаєтеся із числом — заштовхуєте його на стек. Коли дійдете до оператора,

берете два числа з гори стеку (іншими словами, *виштовхуєте* їх), використовуєте оператор і ті два числа в обрахунку, а тоді заштовхуєте отримане число назад на стек. Коли ви дійдете до кінця виразу, то у вас має залишитися однеєдине число — за умови, що вираз був коректно написаним. Саме це число і є результатом.



Проаналізуймо вираз `10 4 3 + 2 * -` разом! Спочатку ми заштовхуємо `10` на стек, і тепер стек дорівнює `10`. Наступний елемент — `4`, то ж ми і його заштовхуємо на стек. Тепер стек дорівнює `10, 4`. Робимо те ж саме з `3`, і отримуємо стек `10, 4, 3`. А тепер ми дійшли до оператора, тобто `+`! Виштовхуємо два горішні числа зі стеку (тепер стек є просто `10`), додаємо ці числа та заштовхуємо результат на стек. Тепер стек має значення `10, 7`. Ми заштовхуємо `2` на стек, стек тепер — це `10, 7, 2`. Знову дійшли до оператора, тому виштовхнемо `7` і `2` зі стеку, помножимо їх і заштовхнемо результат на стек. Добуток `7` і `2` — `14`, отож стек тепер дорівнює `10, 14`. Насамкінець — `-`. Виштовхуємо `10` і `14` зі стеку, віднімаємо `14` від `10` і заштовхуємо результат назад. Тепер число на стеку дорівнює `-4`, і позаяк вираз не містить ані інших чисел, ані операторів, це і є наш результат!

Ми вже знаємо, як вручну вирахувати будь-який ЗПЗ-вираз. А тепер напишімо функцію на Хаскелі, яка б приймала як параметр рядок, що містить ЗПЗ-вираз, наприклад `"10 4 3 + 2 * -"`, і повертала б результат обрахунку того ЗПЗ-виразу.

Яким буде тип такої функції? Ми хочемо, щоб вона приймала рядок як параметр і повертала число як результат. Мабуть, ця функція матиме отаку типосиґнатуру: `solveRPN :: Num a => String -> a`.

Профі-порада: Спочатку краще подумати, яким має бути оголошення типу нашої майбутньої функції, а вже потім думати власне про її реалізацію. Оскільки Хаскел має дуже міцну систему типів, оголошення типу розповість нам про

функцію чимало.

Круто. Перед розв'язанням завдання на Хаскелі не зашкодить подумати про те, як ми розв'язували це завдання вручну — для початку, щоб було від чого відштовхнутися. У цьому випадку ми вважали окремим елементом кожне число або оператор, якщо його було відділено пробілом. Тому можемо спочатку поділити наш рядок `"10 4 3 + 2 * −"` на елементи і отримати список з елементів: `["10","4","3","+","2","*","−"]`.

Що ми далі робили з цим списком з елементів (роблячи розрахунки в голові)? Ми читали його зліва направо і в роботі використовували структуру даних — стек. Попереднє речення вам нічого не нагадує? Пам'ятаєте, як у підрозділі **??** про згортки було сказано, що будь-яку функцію, де ви проходите список зліва направо або справа наліво, елемент за елементом, і будуєте (накопичуєте) якийсь результат (число, список, стек, що завгодно), можна реалізувати за допомогою згортка?

У цьому випадку, ми використаємо лівий згорток, тому що ми читаємо список зліва направо. Накопичувачем буде стек, отож результат згортка також буде стеком. Щоправда, як ми вже бачили, наприкінці розрахунку він міститиме тільки один елемент.

Ще потрібно подумати, як той стек реалізувати. Я пропоную скористатися списком. Також пропоную тримати гору стеку на початку списку. Адже додавання до списку (коли елемент з'являється в голові) — операція значно швидша, ніж приєднання до списку (елемент з'являється в кінці). Отож, якщо ми маємо стек, скажімо, `10, 4, 3`, його реалізацією буде список `[3,4,10]`.

Тепер у нас достатньо інформації, щоб схематично окреслити нашу функцію. Вона братиме рядок, наприклад `"10 4 3 + 2 * −"`, і ділитиме його на список елементів за допомогою `words`, щоб отримати `["10","4","3","+","2","*","−"]`. Далі ми згортатимемо цей список зліва та отримуватимемо стек із одним елементом, тобто `[−4]`. Беремо цей єдиний елемент зі списку — і це і буде наш кінцевий результат!

Ось чернетка із нарисом такої функції:

```
import Data.List

solveRPN :: Num a => String -> a
solveRPN expression = head (foldl foldingFunction [] (words expression))
    where   foldingFunction stack item = ...
```

Беремо вираз і перетворюємо його на список з елементів. Тоді згортаємо цей список елементів за допомогою якоїсь згортаючої функції. Не забуваємо про `[]`, що є початковим значенням накопичувача. Накопичувач — це наш стек, тому `[]` відповідатиме порожньому стекові, із яким ми починаємо нашу роботу. Отримавши стек-результат із одним-єдиним елементом, ми викликаємо `head` по цьому списку, щоб видобути той елемент.

Тепер залишилося реалізувати згортаючу функцію `foldingFunction`, яка братиме стек, наприклад `[4,10]`, і елемент, скажімо `"3"`, і повертатиме новий стек `[3,4,10]`. Якщо стек є `[4,10]`, а елемент — `"*"`, тоді функція муситиме повернути `[40]`. Але спершу перетворімо нашу функцію `solveRPN` на безточкову (дивіться підрозділ **??**), адже зараз вона містить цілу купу дужок, які мене дратують:

```
import Data.List

solveRPN :: Num a => String -> a
solveRPN = head . foldl foldingFunction [] . words
    where   foldingFunction stack item = ...
```

Ось так. Набагато краще. Отож, згортаюча функція братиме стек і елемент і повертатиме новий стек. Скористаймося тепер зіставленням із взірцем — не тільки для отримання елементів з гори стеку, а й також «виловлювання» операторів, як от `"*"` і `"-"`.

```
solveRPN :: (Num a, Read a) => String -> a
solveRPN = head . foldl foldingFunction [] . words
    where   foldingFunction (x:y:ys) "*" = (x * y):ys
            foldingFunction (x:y:ys) "+" = (x + y):ys
            foldingFunction (x:y:ys) "-" = (y - x):ys
            foldingFunction xs numberString = read numberString:xs
```

Ми написали чотири взірці. Взірці перевірятимуться згори донизу. Спочатку згортаюча функція перевірить, чи поточний елемент дорівнює `"*"`. Якщо так, то вона візьме список на кшталт `[3,4,9,3]` і поіменує його перші два елементи `x` і `y`, відповідно. У цьому випадку `x` дорівнюватиме `3`, а `y` — `4`. `ys` іменуватиме залишок `[9,3]`. Функція поверне список, майже точнісінько такий самий як `ys` — із хвостом `ys` і добутком `x` і `y` в голові. Саме за рахунок цього означення ми і виштовхуємо два горішні числа зі стеку, множимо їх і заштовхуємо результат назад на стек. Якщо елемент не дорівнює `"*"`, зіставлення зі взірцем не відбудеться, і перевірятиметься наступний взірець із `"+"`, і так далі.

Якщо елемент не є оператором, ми припускаємо, що це рядок, який містить

рядкове представлення числа. Якщо цей елемент не є оператором, ми застосовуємо до нього `read`, щоб перетворити той рядок на число, а тоді повертаємо попередній стек, із тим числом заштовхнутим на його гору.

Ось і все! Зверніть увагу, що ми додали додаткову умову типокласу `Read a` в оголошення функції — вона має там бути, адже ми застосовуємо `read` до нашого рядка, щоб перетворити його на число. Згідно з цим оголошенням, результат може належати до будь-якого типу, що входить до типокласів `Num` і `Read` (наприклад, `Int`, `Float` і так далі).

Для випадку, де списком на вході є `["2","3","+"]` події загортаються якось отак. Наша функція почне згортати його зліва. Початковий стек буде `[]`. Функція `foldl` викличе згортаючу функцію `foldingFunction` із `[]` у ролі стека (накопичувача) і `"2"` у ролі елемента. Оскільки цей елемент не є оператором, ми викликаємо по ньому `read` і додамо результат до початку `[]`. Отож, новий стек тепер дорівнює `[2]`. Викликаємо згортаючу функцію по `[2]` у ролі стека і `["3"]` у ролі елемента. Отримуємо новий стек `[3,2]`. Викликаємо функцію утретє — з `[3,2]` у ролі стека та `"+"` у ролі елемента. Це спричинює виштовхування цих двох чисел зі стеку, після чого відбувається додавання і заштовхування результату назад на стек. Врешті-решт, стек дорівнює `[5]`, і саме це число ми і повертаємо.

Побавмося з нашою функцією:

```
ghci> solveRPN "10 4 3 + 2 * -"
-4
ghci> solveRPN "2 3 +"
5
ghci> solveRPN "90 34 12 33 55 66 + * - +"
-3947
ghci> solveRPN "90 34 12 33 55 66 + * - + -"
4037
ghci> solveRPN "90 34 12 33 55 66 + * - + -"
4037
ghci> solveRPN "90 3 -"
87
```

Клас, працює! Чудово, що цю функцію можна легко модифікувати і додати підтримку різних інших операторів. Нові оператори не обов'язково мають бути бінарними. До прикладу, можемо створити оператор `"log"`, який виштовхує всього лиш одне число зі стеку і заштовхує назад його логарифм. Можемо також створити тернарний[†] оператор, який виштовхуватиме три числа зі стеку і

[†]Від латинського ternarius, що означає «потрійний».

заштовхуватиме назад результат, або оператор на кшталт `"sum"`, який виштовхує всі числа і заштовхує назад їхню суму.

Перепишемо нашу функцію так, щоб вона підтримувала ще декілька операторів. Щоб трохи спростити собі життя, ми змінимо оголошення типу функції, щоб вона повертала число типу `Float`.

```
import Data.List

solveRPN :: String -> Float
solveRPN = head . foldl foldingFunction [] . words
    where   foldingFunction (x:y:ys) "*" = (x * y):ys
            foldingFunction (x:y:ys) "+" = (x + y):ys
            foldingFunction (x:y:ys) "-" = (y - x):ys
            foldingFunction (x:y:ys) "/" = (y / x):ys
            foldingFunction (x:y:ys) "^" = (y ** x):ys
            foldingFunction (x:xs) "ln" = log x:xs
            foldingFunction xs "sum" = [sum xs]
            foldingFunction xs numberString = read numberString:xs
```

Ого, круто! `/` — це, звичайно ж, ділення, а `**` — це плавомкове піднесення до степеня. У випадку оператора логарифму `"ln"`, взірець викусює єдиний елемент зі стеку, адже нам потрібен тільки один елемент, щоб порахувати натуральний логарифм за допомогою функції `log`. Для випадку оператора суми `"sum"`, повертаємо стек, який містить лише один елемент, що дорівнює сумі елементів, що вони є у тому стекові на теперішній момент.

```
ghci> solveRPN "2.7 ln"
0.9932518
ghci> solveRPN "10 10 10 10 sum 4 /"
10.0
ghci> solveRPN "10 10 10 10 10 sum 4 /"
12.5
ghci> solveRPN "10 2 ^"
100.0
```

Зауважте, що ми можемо включити числа з плаваючою комою до нашого виразу, бо `read` знає, як їх читати.

```
ghci> solveRPN "43.2425 0.5 ^"
6.575903
```

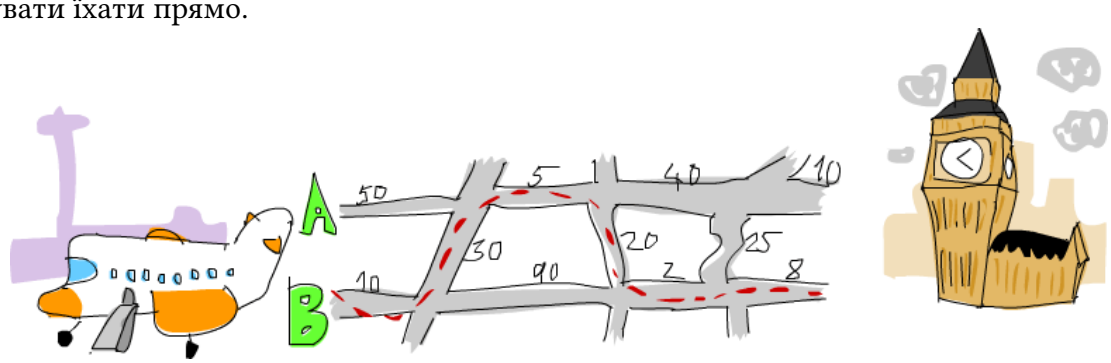
Я гадаю, що написати функцію з 10 рядочків тексту, яка вираховує ЗПЗ-вирази з чисел із плаваючою комою і яку можна легко модифікувати і підтримувати, — це круто.

Усе ж, варто зазначити, що ця функція насправді не є відмовостійкою. Якщо подати якісь безглузді вхідні дані, вона відразу дасть збій і завалить разом із собою все, що зможе. Ми напишемо відмовостійку версію цієї функції, яка матиме оголошення типу `solveRPN :: String -> Maybe Float`, як тільки познайомимося з монадами (вони не страшні, чесно!). Ми б могли написати таку функцію негайно, але це трохи марудно, бо на кожному кроці доведеться перевіряти, чи не є часом результат `Nothing`. Але якщо вам море по коліна, тоді уперед! Підказка: скористайтеся функцією `reads`, щоб перевірити, чи читання було успішним чи ні.

10.2 З Гітроу до Лондона

Ось наше наступне завдання: ваш літак щойно приземлився в Англії і ви взяли напрокат авто. Ви поспішаєте на зустріч і тому мусите добратися з аеропорту Гітроу до Лондона якомога швидше (але безпечно!).

З Гітроу до Лондона ведуть дві головні траси. Їх перетинає низка регіональних доріг. Шлях від одного перехрестя до іншого займає вам одну й ту ж кількість часу. Потрібно знайти оптимальний шлях — такий, що дозволить вам дістатися до Лондона якнайшвидше! Ви починаєте їхати з лівого боку і можете або повернути на перехресті і переїхати на іншу головну дорогу, або продовжувати їхати прямо.



Як ви бачите на малюнку, найкоротший шлях із Гітроу до Лондона — почати їхати по трасі B, звернути на перехресті і переїхати на A, далі рухатися прямо по трасі A, знову звернути повернутися на B, і проїхати два перехрестя на B. Цей шлях займе 75 хвилин. Якби ми обрали будь-який інший шлях, то витратили б більше часу.

Наше завдання — написати програму, що бере вхідні дані, які представляють цю систему доріг, і виводить найкоротший шлях. Ось як виглядатимуть вхідні дані у цьому випадку:

```
50
10
```

```
30
5
90
20
40
2
25
10
8
0
```

Щоб проаналізувати вхідний файл подумки, розбийте його по три елементи і поділіть дорогу на відтинки. Кожен відтинок складається із траси A, траси B і дороги, що їх перетинає. Щоб все гарно ділилося на три, скажемо, що існує останнє перехрестя, яке можна проїхати за 0 хвилин. Це тому, що нам байдуже, де саме ми в'їжджаємо до Лондона. Основне — що ми в Лондоні.

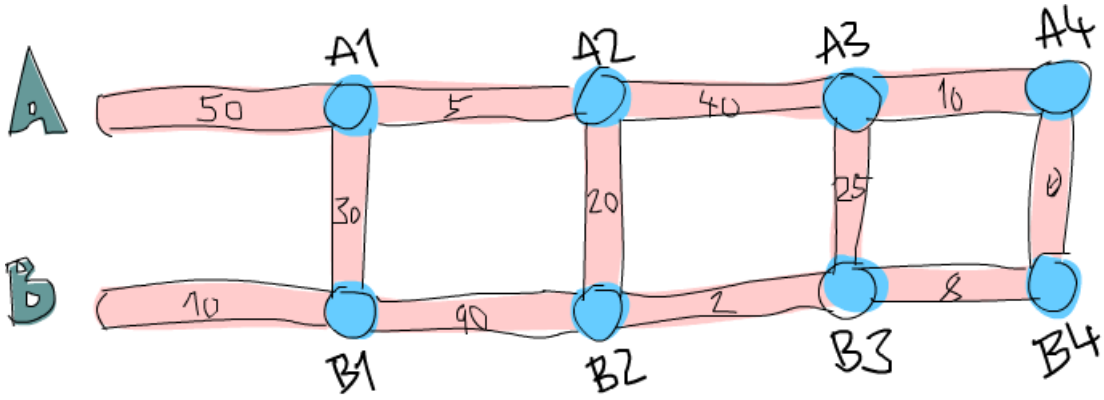
Спробуймо розв'язати це завдання за три кроки, як задачу про калькулятор зворотного польського запису:

1. на хвильку забудьте про Хаскел і подумайте, як розв'язати це завдання вручну;
2. поміркуйте, як представити наші дані в Хаскелі;
3. придумайте, як опрацювати ці дані в Хаскелі, щоб отримати розв'язок.

У розділі про ЗПЗ-калькулятор ми спершу дійшли висновку, що коли ми вираховуватимемо вираз вручну, то триматимемо в голові такий собі стек, і будемо проходити вираз елемент за елементом. Ми вирішили представити наш вираз як список рядків. Врешті-решт, ми скористалися лівим згортком, щоб пройтися тим списком рядків, тримаючи водночас стек, і отримати розв'язок.

Гаразд, як **визначити** [to figure out] найкоротший шлях із Гітроу до Лондона вручну? Можемо просто поглянути на малюнок і спробувати вгадати, який шлях найкоротший. Раптом нам пощастить. Таке рішення підходить для дуже малих вхідних даних. А якби ми мали дорогу з 10000 перехресть? Пфе! Крім того, працюючи навмання, ми ніколи не будемо впевнені в тому, що наше рішення оптимальне. Можемо тільки запевнити (на кшталт «я ДУЖЕ впевнений, що…»), але не довести.

Тож такий підхід… не піде! Ось як можна візуалізувати нашу систему доріг схематично:



Можете сказати, який найкоротший шлях до першого перехрестя на трасі A (перша блакитна крапка на A, позначена як *A1*)? Доволі просто. Треба просто подивитися, чи краще їхати прямо по A чи прямо по B, а тоді звернути на перехресті і переїхати на A. Звичайно ж, «дешевше» їхати прямо по B, а тоді звернути, бо цей шлях займе 40 хвилин. А якщо ми поїдемо по A, то витратимо 50 хвилин. А як щодо перехрестя *B1*? Те ж саме. Набагато дешевше їхати прямо по B (і витратити всього 10 хвилин), ніж їхати по A, а тоді звернути, адже непрямий шлях займе нам аж 80 хвилин!

Тепер ми знаємо найдешевший шлях до *A1* (їхати по B, а тоді переїхати на A; тобто, відтепер ми казатимемо, що це є шлях `B, C` і він коштує 40) і знаємо найдешевший шлях до *B1* (прямо по B, тобто, шлях `B`, який коштує 10). Чи це знання допоможе нам, якщо ми захочемо дізнатися найдешевший шлях до наступних перехресть на обох трасах? Чорт забирай, так!

Отож, яким буде найкоротший шлях до *A2*? Щоб дістатися *A2*, ми або поїдемо прямо до *A2* з *A1* або поїдемо прямо з *B1*, а пізніше переїдемо на A (пам'ятаймо: ми можемо тільки рухатися прямо або переїжджати на інший бік). Оскільки ми знаємо вартість проїзду до *A1* і *B1*, то можемо легко визначити найкращий шлях до *A2*. Ми витратимо 40, щоб дістатися до *A1*, і ще 5, щоб переїхати від *A1* до *A2*. Тобто, маємо `B, C, A` з вартістю 45. Дістатися до *B1* коштує тільки 10, але далі ми витратимо додатково 110 хвилин, якщо будемо їхати до *B2* і переїжджатимемо! Отож, найдешевший шлях до *A2* — це `B, C, A`. Так само, найдешевша дорога до *B2* — їхати прямо з *A1*, а потім переїхати на B.

Можливо вас мучить питання: а якби дістатися до *A2* отак: спершу, звернувши на перехресті *B1*, доїхати до *A1*, і далі рухатись прямо? Насправді, ми вже врахували переїзд від *B1* до *A1*, коли шукали найкращого шляху до *A1* у попередньому кроці, тому нам не потрібно розглядати цей шлях знову у наступному.

Маючи найкращий шлях до *A2* і *B2*, можемо повторювати цей крок безкінечно, поки не дійдемо кінця. Тепер ми знайшли найліпші дороги до *A4* і *B4*, і та, що дешевша, і буде оптимальною!

По суті, на другому кроці ми повторили те саме, що зробили на першому, от лише ми взяли до уваги попередній розв'язок — найліпші шляхи до перехресть на A і B. Можна поглянути на перший крок отак: ми там врахували найкращі шляхи до A і B також, тільки от вони в цьому випадку були порожніми шляхами із вартістю 0, і шукати їх не треба було.

Підведемо підсумки. Щоб прокласти найкращий маршрут з Гітроу до Лондона, ми робимо ось що: спочатку дивимося, який найкращий шлях до наступного перехрестя на трасі A. Два можливі маршрути — їхати прямо або почати рух на протилежній трасі, їхати прямо і переїхати. Запам'ятовуємо вартість і шлях для кожного маршруту. Далі робимо те саме, щоб перевірити, який найліпший маршрут до наступного перехрестя на трасі B. Далі перевіряємо, чи шлях до наступного перехрестя на A буде дешевшим, якщо їхати від попереднього перехрестя з A чи якщо їхати від попереднього перехрестя з B, а тоді переїхати. Запам'ятовуємо дешевший маршрут і, знов-таки, робимо те ж саме на протилежному перехресті. Робимо вищезазначені підрахунки для кожного відтинку, аж поки не дійдемо кінця. Коли ми закінчили, то дешевший із двох маршрутів — це й буде наш оптимальний шлях!

По суті, ми вираховуємо найкоротший шлях на трасі A і найкоротший шлях на трасі B, а коли закінчуємо обрахунки, то коротший із цих шляхів — це наш розв'язок. Тепер ми вміємо вирахувати найкоротший маршрут вручну. Якби ви мали досить часу, папір і олівці, то змогли б вирахувати найкоротший шлях у системі доріг із будь-якою кількістю відтинків.

Наступний етап розв'язання задачі! Як відобразити цю систему доріг за допомогою типів даних Хаскела? Спосіб номер один: хай вихідні точки і перехрестя будуть вузлами графу, а дороги — ребрами, що їх з'єднують. Кожен вузол матиме два орієнтовані ребра, що «виходитимуть» з нього. Тобто, кожне перехрестя (або вузол) «вказуватиме» тим орієнтованим ребром на вузол з протилежного боку (на протилежній трасі) та на наступний вузол зі свого боку (на своїй). За винятком останніх вузлів — там ребра вказуватимуть тільки на протилежний бік.

```
data Node = Node Road Road | EndNode Road
data Road = Road Int Node
```

Вузол — це або «нормальний» вузол `Node`, який має інформацію про (1) дорогу, що веде до іншої траси, і (2) дорогу, що веде до наступного вузла, або ж вузол то є кінцевий вузол `EndNode`, який містить інформацію тільки про дорогу, що веде до вузла (перехрестя) на іншій трасі. Дорога `Road` містить інфор-

мацію про те, якої вона довжини, і до якого вузла вона веде. До прикладу, перша частина дороги на трасі A дорівнюватиме `Road 50 a1`, де `a1` буде вузлом `Node x y`, де `x` і `y` — це дороги, що ведуть до *B1* і *A2*, відповідно.

Інший підхід — використати `Maybe` в реалізації для доріг, що ведуть уперед. Кожен вузол має дорогу, що веде на інший бік, але тільки ті вузли, які не є кінцевими, матимуть на додачу дорогу, що веде уперед.

```
data Node = Node Road (Maybe Road)
data Road = Road Int Node
```

Це непоганий спосіб відобразити систему доріг на Хаскелі. Можемо розв'язати завдання і таким чином, але можливо є якийсь простіший спосіб? Пригадаймо, як ми розв'язали завдання вручну: ми завжди одночасно перевіряли довжину трьох доріг: дорогу між двома перехрестями на трасі A, протилежну їй відповідну дорогу на трасі B і дорогу C, яка з'єднує відповідні перехрестя. Коли ми шукали найкоротшого маршруту до *A1* і *B1*, то мали справу тільки з довжиною перших трьох частин, завдовжки 50, 10 і 30. Назвімо це одним відтинком (`Section`). Отож, систему доріг можна представити як чотири відтинки: `50, 10, 30`, `5, 90, 20`, `40, 2, 25`, і `10, 8, 0`.

Добра стара весела програмістська із-означенням-в-означенні мудрість згадалася в цьому контексті: треба шукати для розв'язку такі структури даних, що уможливлюють розв'язання але й одночасно є найпростішими з усіх можливих, але не більш простими ніж *такі* (що уможливлюють розв'язання але й одночасно є найпростішими з усіх можливих).[†]

```
data Section = Section { getA :: Int, getB :: Int, getC :: Int }
                   deriving (Show)
type RoadSystem = [Section]
```

Майже ідеально! Страшенно просто і чудово, і в мене зараз таке відчуття, що таке представлення даних ідеально підтримуватиме наш алгоритм пошуку оптимального розв'язку. Відтинок `Section` — це простий алгебраїчний тип даних, який містить три цілі числа, що відповідають довжинам трьох доріг, з яких складається цей відтинок. Ми також вводимо тип-синонім: кажемо, що `RoadSystem` — це список відтинків.

Примітка: Ми могли представити відтинок дороги і за допомогою триплету `(Int, Int, Int)`. Використання кортежів замість написання власних алгебраїчних типів даних більше підходить до розв'язання якогось маленького локалізованого завдання. А для такої задачі, як ця, ліпше створити новий тип. Він

[†] «As simple as possible, but not any simpler».

дає системі типів більше інформації про що є що. `(Int, Int, Int)` може представляти відтинок дороги або вектор у 3D просторі. Якщо з ними працювати одночасно, то їх буде легко переплутати. Використовуючи типи даних `Section` і `Vector`, ми не зможемо, наприклад, випадково додати вектор до відтинка системи доріг.

Наша система доріг із Гітроу до Лондона може бути представлена ось так:

```
heathrowToLondon :: RoadSystem
heathrowToLondon = [ Section 50 10 30
                   , Section 5 90 20
                   , Section 40 2 25
                   , Section 10 8 0]
```

Тепер потрібно всього лиш реалізувати рішення, до якого ми дійшли раніше, на Хаскелі. Яким має бути оголошення типу функції, що вираховує найкоротший маршрут для довільної системи доріг? Функція мусить приймати систему доріг як параметр і повертати шлях. Хай шлях в нашій програмі буде представлено як список. Введемо тип `Label`, який буде переліком з `A`, `B` і `C`. Напишемо також синонім типу: `Path`.

```
data Label = A | B | C deriving (Show)
type Path = [(Label, Int)]
```

Отже, наша функція, яку ми назвемо `optimalPath`, мусить мати оголошення типу `optimalPath :: RoadSystem -> Path`. Якщо її викликати із системою доріг `heathrowToLondon`, вона повинна повернути ось такий маршрут:

```
[(B,10),(C,30),(A,5),(C,20),(B,2),(B,8)]
```

Нам треба опрацювати список відтинків зліва направо, зберігаючи оптимальний шлях на A і оптимальний шлях на B. Ми накопичуватимемо найкращий шлях, переходячи списком зліва направо. Це вам нічого не нагадує? Чуєте дзвіночок? Так, це ЛІВИЙ ЗГОРТОК!

Коли ми шукали розв'язок вручну, то весь час повторювали одне і те саме знову і знову. Ми оновлювали вже знайдені оптимальні шляхи на A і B дорогами з поточного відтинку, щоб визначити нові оптимальні шляхи на A і B. До прикладу, спочатку оптимальні шляхи дорівнювали `[]` і `[]` для трас A і B відповідно. Ми проаналізували відтинок `Section 50 10 30` і дійшли висновку, що новий оптимальний шлях до *A1* — це `[(B,10),(C,30)]`, а оптимальний шлях до *B1* — це `[(B,10)]`. Якщо подивитися на цей крок як на функцію, то

вона бере пару шляхів і відтинок та повертає нову пару шляхів. Отож, тип буде `(Path, Path) -> Section -> (Path, Path)`. Реалізуймо цю функцію, бо вона точно стане нам у пригоді.

Підказка: вона буде корисною, бо `(Path, Path) -> Section -> (Path, Path)` можна використовувати як бінарну функцію для лівого згортка, яка має тип `a -> b -> a`.

```
roadStep :: (Path, Path) -> Section -> (Path, Path)
roadStep (pathA, pathB) (Section a b c) =
    let priceA = sum $ map snd pathA
        priceB = sum $ map snd pathB
        forwardPriceToA = priceA + a
        crossPriceToA = priceB + b + c
        forwardPriceToB = priceB + b
        crossPriceToB = priceA + a + c
        newPathToA = if forwardPriceToA <= crossPriceToA
                     then (A,a):pathA
                     else (C,c):(B,b):pathB
        newPathToB = if forwardPriceToB <= crossPriceToB
                     then (B,b):pathB
                     else (C,c):(A,a):pathA
    in  (newPathToA, newPathToB)
```

Що тут відбувається? Спершу ми вираховуємо ціну оптимального шляху, що його вже було знайдено (і передано як параметр до цієї функції), для траси A. Те ж саме робимо для оптимального шляху, що веде до перехрестя на трасі B. Наприклад, ми рахуємо `sum $ map snd pathA`, і, якщо `pathA` є щось типу `[(A,100),(C,20)]`, то `priceA` вирахується собі `120`. `forwardPriceToA` — це ціна, яку б ми заплатили, якби їхали з попереднього перехрестя на A до наступного перехрестя на A по трасі A. Ця ціна дорівнює найкращій ціні на A дотепер (до поточного перехрестя), плюс довжина частини дороги, що з'єднує відповідні перехрестя на A. `crossPriceToA` — це ціна, яку ми заплатили б, якщо поїхали б до наступного перехрестя на A вирушаючи з попереднього перехрестя на трасі B, рухаючись уперед по B, а тоді переїжджа-

ючи на A. Ця ціна дорівнює найкращій ціні до попереднього перехрестя на трасі B плюс довжина дороги, що з'єднує це перехрестя із наступним перехрестям на B, плюс довжина дороги C, що з'єднує те перехрестя із відповідним йому перехрестям на A. Ми визначаємо `forwardPriceToB` і `crossPriceToB` таким самим чином.

Тепер ми вже знаємо найкращі шляхи до перехресть на A та B — але є одне але: мусимо занотувати ці результати на мові шляхів і оновити шляхи A і B. Якщо дешевше їхати прямо по A, ми покладаємо `newPathToA` рівним `(A,a):pathA`. Іншими словами, ми приєднуємо `Label A` і довжину дороги `a` до поточного оптимального шляху на A. По суті, ми кажемо, що найкращий шлях до наступного перехрестя на A — це шлях до попереднього перехрестя на A, а далі — прямо по тій самій трасі. Пам'ятайте, що `A` — це всього лиш мітка, тоді як `a` має тип `Int`. Чому ми приєднуємо до списку (елемент з'являється в голові), а не додаємо (елемент з'являється в кінці, як от `pathA ++ [(A,a)]`)? Справа в тому, що приєднання елементу до списку є операція набагато швидша ніж додавання елементу до списку. В наслідок такої оптимізаційної халепи після згортання результат, що його поверне згорток, буде в зворотньому порядку, але це легко виправити — просто розвернемо його в кінці. Якщо ж буде дешевше дістатися до наступного перехрестя на A по-іншому (а саме: їхати прямо по трасі B, а потім переїхати на A), тоді `newPathToA` дорівнюватиме ось чому: це буде старий шлях до перехрестя на B, який далі іде прямо по B, а потім звертає на A. Після того, як розібралися із A, ми рахуємо `newPathToB` — це операція схожа до обрахування `newPathToA`, із точністю до дзеркального відображення.

Насамкінець, повертаємо `newPathToA` і `newPathToB` в парі.

Запустимо цю функцію по першому відтинку `heathrowToLondon`. Позаяк це перший відтинок, параметр, що відповідає найкращім шляхам на A і B, буде парою порожніх списків.

```
ghci> roadStep ([], []) (head heathrowToLondon)
([(C,30),(B,10)],[(B,10)])
```

Не забувайте, що шляхи повертаються розверненими, тому їх треба читати справа наліво. Ми бачимо, що найкращий шлях до наступного перехрестя на A — це почати рух на B і звернути на A. А найкращий шлях до наступного перехрестя на B — просто рухатися прямо з вихідної точки на B.

Оптимізаційна порада: виконуючи `priceA = sum $ map snd pathA`, ми рахуємо вартість шляху щокроку. Нам би не довелося цього робити, якби ми реалізували `roadStep` як функцію

`(Path, Path, Int, Int) -> Section -> (Path, Path, Int, Int)`, де цілі числа в кінці — найкращі ціни шляхів для A і B.

Отож, ми маємо функцію, що бере пару шляхів і відтинок і прокладає нові оптимальні шляхи. Тепер ми можемо легко згорнути список відтинків лівим згортком. До `roadStep` подаються два параметри — `([],[])` і перший відтинок, а повертає ця функція пару оптимальних шляхів, беручи до уваги цей відтинок. Далі цю функцію викликаємо із цією пара шляхів і наступним відтинком, і так далі. Коли ми перейдемо через усі відтинки, то отримаємо пару оптимальних шляхів. Коротший із них — це і є наша відповідь. Озброївшись цим знанням — реалізуймо `optimalPath`!

```
optimalPath :: RoadSystem -> Path
optimalPath roadSystem =
    let (bestAPath, bestBPath) = foldl roadStep ([],[]) roadSystem
    in  if sum (map snd bestAPath) <= sum (map snd bestBPath)
            then reverse bestAPath
            else reverse bestBPath
```

Згортаємо `roadSystem` (це список відтинків) зліва. Початковий накопичувач — це пара порожніх шляхів. Результат згортка — пара шляхів, тож зіставляємо із взірцем, щоб ті шляхи з пари витягнути. Далі перевіряємо, який з них дешевший, і повертаємо його. Перед власне поверненням, ми змінюємо напрям шляху, оскільки оптимальні шляхи досі пливли по плюграмі розвернені, завдяки нашій оптимізації (коли ми обрали приєднання, а не додавання до списку).

Ну що — тестуймо?

```
ghci> optimalPath heathrowToLondon
[(B,10),(C,30),(A,5),(C,20),(B,2),(B,8),(C,0)]
```

Це саме той результат, який ми хотіли отримати! Чудово! Він трішки відрізняється від того, на що ми сподівалися, тому що наприкінці є крок `(C,0)`. Це означає, що коли ми в'їжджаємо до Лондона, то звертаємо на іншу трасу, але оскільки цей поворот нічого не коштує, то наш результат цілком правильний.

Ми маємо функцію, які шукає оптимальний шлях. Тепер треба прочитати текстове представлення системи доріг з потоку стандартного введення (stdin), перетворити його на тип `RoadSystem`, запустити по ньому функцією `optimalPath` і вивести шлях в потік стандартного виведення (stdout).

Спершу напишемо функцію, що бере список і ділить його на групи однакової довжини. Назвімо її `groupsOf`. Для параметру `[1..10]`, `groupsOf 3` має повернути `[[1,2,3],[4,5,6],[7,8,9],[10]]`.

```
groupsOf :: Int -> [a] -> [[a]]
groupsOf 0 _ = ⊥
groupsOf _ [] = []
groupsOf n xs = take n xs : groupsOf n (drop n xs)
```

Стандартна рекурсивна функція. Якщо `xs` є `[1..10]` і `n` є `3`, `groupsOf 3 [1..10]` дорівнюватиме `[1,2,3] : groupsOf 3 [4,5,6,7,8,9,10]`. Коли рекурсія завершиться, ми отримаємо список, поділений на групи із трьох елементів. Ось наша функція `main`, яка читає дані, будує з них систему доріг `RoadSystem` і виводить найкоротший шлях:

```
import Data.List

main = do
    contents <- getContents
    let threes = groupsOf 3 (map read $ lines contents)
        roadSystem = map (\[a,b,c] -> Section a b c) threes
        path = optimalPath roadSystem
        pathString = concat $ map (show . fst) path
        pathPrice = sum $ map snd path
    putStrLn $ "The best path to take is: " ++ pathString
    putStrLn $ "The price is: " ++ show pathPrice
```

Спочатку ми читаємо все, що можна прочитати з потоку stdin. Далі викликаємо `lines` по прочитаному, щоб перетворити `"50\n10\n30\n...` на `["50","10","30"..`. Врешті-решт, відображаємо то за допомогою `read`, щоб отримати список чисел. Викликаємо на ньому `groupsOf 3`, щоб дістати список списків, де кожен підсписок є завдовжки 3. Відображаємо цей список списків за допомогою лямбди `([a,b,c] -> Section a b c)`. Як бачите, лямбда бере список завдовжки 3 і перетворює його на відтинок. Отож, `roadSystem` — це наша система доріг, яка навіть має правильний тип, тобто `RoadSystem` (або `[Section]`). Викликаємо `optimalPath`, отримуємо шлях і ціну у гарному текстовому представленні та виводимо їх.

Зберігаємо ось цей текст

```
50
10
30
5
90
20
40
```

```
2
25
10
8
0
```

у файлі під назвою `paths.txt` і подаємо цей файл програмі.

```
$ cat paths.txt | runhaskell heathrow.hs
The best path to take is: BCACBBC
The price is: 75
```

Працює! Можете використати своє знання модуля `Data.Random` в побудові значно довшої випадкової системи доріг, і погратися, взадаючи її цій програмі. Якщо трапиться переповнення стеку, спробуйте використати `foldl'` замість `foldl` (`foldl'` — завзятий кум лінивого `foldl`).

Покажчик