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

2017-05-21T00:06:31Z
Версія v4.7-54-gda41cf2

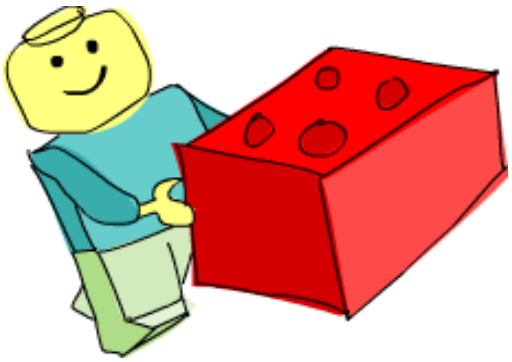
Зміст

Розділ 7

Модулі

Переклад українською Тетяни Богдан

7.1 Завантаження модулів



Модуль в Хаскелі — це купка споріднених функцій, типів та типокласів. Програма в Хаскелі — це набір модулів, в якій головний модуль завантажує інші модулі і потім використовує функції, що означені в них, для виконання роботи. Розподілення коду по модулях — дуже корисна штука. Якщо модуль достатньо загально написаний, функції, які він «виставляє назовні» (іншими словами, експортує), можуть бути використані в безлічі різних програм. Якщо ваш власний код розподілено між самодостатніми модулями, які не дуже залежать один від одного (можна також сказати, що вони слабко зчеплені), ці модулі можна буде перевикористати пізніше. Програмування значно полегшується, коли код поділено на декілька частин, кожна з яких має якусь мету.

Стандартна бібліотека Хаскела розподілена на модулі, кожен за яких містить функції та типи, які певним чином пов'язані чи разом розв'язують якусь задачу. Так, існує модуль для обробки списків, модуль для паралельного програмування, модуль для боротьби з комплексними числами і так далі. Всі функції, типи та типокласи, які ми досі зустрічали, належать до модуля `Prelude`, який імпортується за замовчуванням. В цьому розділі ми познайомимося із декількома корисними модулями та дослідимо функції і типи, які ці модулі

експортують. Але спочатку ми маємо навчитися імпортувати модулі.

Синтаксис для імпорту модулів в хаскельний код отакий: `import <<module name>>`. Імпорти мають передувати означенням функцій, тому імпорти як правило робляться на початку файлу. Кожна інструкція імпорту повинна бути окремим рядком коду. Отже, імпортуймо модуль `Data.List`, в якому міститься чимало функцій корисних для роботи із списками, і використаймо одну з них — функцію `nub` — в написанні функції, що рахуватиме кількість унікальних елементів у списку.

```
import Data.List

numUniques :: Eq a => [a] -> Int
numUniques = length . nub
```

Коли виконується `import Data.List`, всі функції, які `Data.List` експортує, стають доступними в глобальному просторі імен, тобто тепер їх можна викликати в будь-якому місці цього файла. Функція `nub` означена в `Data.List`; вона бере список і повертає список-результат, схожий на список на вході, от тільки елементи у ньому не повторюються — перший елемент з ланцюжку повторів лишається як є, а всі наступні — вилучаються. За допомогою композиції функцій `length` та `nub` отримуємо функцію `length . nub`, яка еквівалентна `\xs -> length (nub xs)`.

Функції з модулів можуть потрапити в глобальний простір імен також при користуванні GHCi. Протягом сесії в GHCi, якщо ви хочете отримати доступ до функцій, які експортує модуль `Data.List`, зробіть оце:

```
ghci> :m + Data.List
```

Працюючи в GHCi, аби завантажити функції з декількох модулів не треба виконувати `:m +` декілька разів — можна завантажити декілька модулів за раз.

```
ghci> :m + Data.List Data.Map Data.Set
```

Тим часом, якщо ви завантажуєте програму, яка вже імпортує якийсь модуль, немає потреби імпортувати його окремо за допомогою `:m +` аби достукатися до нього.

Якщо вам потрібні лише кілька конкретних функцій з якогось модуля, можна тільки їх і імпортнути. Наприклад, якщо треба доступ лише до `nub` та `sort` з модуля `Data.List`, можна імпортувати їх вибірково:

```
import Data.List (nub, sort)
```

Також існує можливість імпортувати всі функції якогось модуля за винятком деяких. Це стає у пригоді в ситуації, коли декілька модулів містять функції з одним і тим самим ім'ям і треба позбутися непотрібних однойменників.

Як от у нас вже є власна функція на ім'я `nub`, а нам, скажімо, треба імпортувати всі функції з `Data.List` окрім функції `nub`:

```
import Data.List hiding (nub)
```

Інший спосіб боротьби з конфліктами імен — імпорти в підпростір імен. Модуль `Data.Map`, який містить структури даних для пошуку значень за ключем, експортує купу функцій з тими самими іменами, що й модуль `Prelude`. Наприклад — `filter` та `null`. Отже, якщо після імпорту `Data.Map` викликати `filter`, компілятор не знатиме яку функцію використовувати. Ось в який спосіб цьому можна зарадити:

```
import qualified Data.Map
```

Тепер, якщо ми хочемо викликати `filter` із модуля `Data.Map`, маємо гукнути її отак: `Data.Map.filter`. Просте ім'я «filter» все іще належить нашій давно знайомій і улюбленій функції `filter` (з `Prelude`). Але видруковувати `Data.Map` перед кожною функцією з модуля `Data.Map` доволі нудно. Ось чому в нас є можливість дати коротше прізвисько цьому підпростору:

```
import qualified Data.Map as M
```

Тепер аби доступитися до `filter` із `Data.Map` можна писати просто `M.filter`.

Ось за цим корисним посиланнячком можна знайти перелік модулів стандартної бібліотеки. Прегарний спосіб набиратися нових знань з Хаскела — це просто проглядати стандартну бібліотеку і досліджувати модулі та їхні функції. Сирці всіх модулів також доступні для перегляду. Читання сирців — також непоганий спосіб вивчення Хаскела: перегляд сирців деяких модулів допоможе вам зміцнити ваше відчуття «реального» хаскельного коду.

Для пошуку функцій або місць їхнього проживання використовуйте Hoogle. Це по-справжньому крутий пошуковий сервер — він шукає за іменами функцій, іменами модулів і навіть сиґнатурами типу.

7.2 Data.List

Модуль `Data.List`, звичайно, присвячений спискам. Він містить доволі корисні функції для роботи з ними. Ми вже зустрічалися із деякими з них (як от `map` та `filter`), оскільки модуль `Prelude` експортує певні функції з `Data.List`, для зручності. Не треба імпортувати `Data.List` в підпростір імен, бо `Prelude` і `Data.List` не містять однакових імен (окрім тих, що `Prelude` вже підкрадає в

`Data.List` [†]). А тепер — перегляньмо деякі функції, яких ми ще не зустрічали.

`intersperse` бере елемент та список і вставляє той елемент між кожною парою елементів списку. Ось як вона працює:

```
ghci> intersperse '.' "MONKEY"
"M.O.N.K.E.Y"
ghci> intersperse 0 [1,2,3,4,5,6]
[1,0,2,0,3,0,4,0,5,0,6]
```

`intercalate` приймає список і список списків. Вона вставляє перший аргумент-список між всіма списками у другому аргументі, а потім розморщує (сплощує) список-результат (перетворює список із вкладеними списками у «одношаровий» список).

```
ghci> intercalate " " ["hey","there","guys"]
"hey there guys"
ghci> intercalate [0,0,0] [[1,2,3],[4,5,6],[7,8,9]]
[1,2,3,0,0,0,4,5,6,0,0,0,7,8,9]
```

`transpose` транспонує список списків. Якщо поглянути на список списків як на двовимірну матрицю, то ця функція «обертає» матрицю навколо її діагоналі — стовпчики матриці стають рядками і навпаки.

```
ghci> transpose [[1,2,3],[4,5,6],[7,8,9]]
[[1,4,7],[2,5,8],[3,6,9]]
ghci> transpose ["hey","there","guys"]
["htg","ehu","yey","rs","e"]
```

Нехай у нас є поліноми $3x^2 + 5x + 9$, $10x^3 + 9$ та $8x^3 + 5x^2 + x - 1$ і нам треба їх додати одне до одного. В Хаскелі ці поліноми можна представити списками `[0,3,5,9]`, `[10,0,0,9]` та `[8,5,1,-1]`. А для того, щоб додати їх, треба просто виконати:

```
ghci> map sum $ transpose [[0,3,5,9],[10,0,0,9],[8,5,1,-1]]
[18,8,6,17]
```

Після транспозиції списків 3-ті степені стоятимуть в першому рядку матриці, 2-гі — в другому і так далі. Відображення за допомогою `sum` у такому представлені повертає саме те що треба.

`foldl'` та `foldl1'` є завзятішими версіями лінивих `foldl` та `foldl1`, відповідно. Якщо згортати по-справжньому здоровезні списки ліниво,

[†]Функції для роботи зі списками з `Prelude` насправді живуть в `Data.List`, а `Prelude` просто їх реекспортує.

то іноді можна отримати переповнення стеку. Поясню чому: через лінивість згортків значення накопичувача не оновлюється поки список згортається, а, насправді, накопичувач обіцяє, що зробить обрахунок, але лише тоді, коли, власне, буде треба результат того обрахунку. Такі обрахунки-обіцянки також називають подумками. Подумки створюються для кожного проміжного значення накопичувача, і, зрештою, переповнюють стек. Завзяті згортки за природою не ледацюги, і тому обраховують проміжні значення в процесі згортання, замість того, щоб переповнювати ваш стек подумками. Отже, у випадку, коли стек переповнюється через згорток-ледащо, спробуйте замінити той лінивий згорток на його завзятого родича.

`concat` розморщує список — перетворює список списків на список з елементів.

```
ghci> concat [”foo”,”bar”,”car”]
”foobarcar”
ghci> concat [[3,4,5],[2,3,4],[2,1,1]]
[3,4,5,2,3,4,2,1,1]
```

Ця операція прибирає лише один рівень вкладеності. Якщо потрібно повністю розморщити, наприклад, `[[[2,3],[3,4,5],[2]],[[2,3],[3,4]]]`, який є списком списків списків, треба застосувати `concat` двічі.

Функція `concatMap` — це те саме, що спочатку відобразити список в інший список (за допомогою якоїсь функції), а потім прибрати один рівень вкладеності зі списку-результату за допомогою `concat`.

```
ghci> concatMap (replicate 4) [1..3]
[1,1,1,1,2,2,2,2,3,3,3,3]
```

`and` бере список булевих виразів і повертає `True` лише якщо всі значення в списку є `True`.

```
ghci> and $ map (>4) [5,6,7,8]
True
ghci> and $ map (==4) [4,4,4,3,4]
False
```

`or` схожа на `and`, тільки от вона повертає `True`, якщо якийсь з булевих виразів в списку після обчислення приймає значення `True`.

```
ghci> or $ map (==4) [2,3,4,5,6,1]
True
ghci> or $ map (>4) [1,2,3]
False
```

`any` бере предикат і перевіряє, чи є принаймні один елемент елемент у списку, що задовольняє йому. `all` — схожа: бере предикат і перевіряє, чи всі елементи списку задовольняють йому. Зазвичай ці дві функції використовуються замість відображення списку із подальшим застосуванням `and` чи `or`.

```
ghci> any (==4) [2,3,5,6,1,4]
True
ghci> all (>4) [6,9,10]
True
ghci> all (`elem` ['A'..'Z']) "HEYGUYSwhatsup"
False
ghci> any (`elem` ['A'..'Z']) "HEYGUYSwhatsup"
True
```

`iterate` бере функцію та початкове значення, і застосовує цю функцію до початкового значення, потім — застосовує цю ж функцію до результату застосування цієї функції у попередньому кроці і так далі. Всі результати повертаються у вигляді нескінченного списку.

```
ghci> take 10 $ iterate (*2) 1
[1,2,4,8,16,32,64,128,256,512]
ghci> take 3 $ iterate (++ "haha") "haha"
["haha","hahahaha","hahahahahaha"]
```

`splitAt` бере число, n, та список і розбиває цей список на два підсписки так, щоб перший був довжини n. Два підсписки повертаються у кортежі.

```
ghci> splitAt 3 "heyman"
("hey","man")
ghci> splitAt 100 "heyman"
("heyman","")
ghci> splitAt (-3) "heyman"
("","heyman")
ghci> let (a,b) = splitAt 3 "foobar" in b ++ a
"barfoo"
```

`takeWhile` — це дуже корисна маленька функційка. Вона бере елементи зі списку поки вони задовольняють умові. Щойно трапляється елемент, який не задовольняє умові, `takeWhile` обриває список. Із досвіду скажу, це дуже корисна штука.

```
ghci> takeWhile (>3) [6,5,4,3,2,1,2,3,4,5,4,3,2,1]
[6,5,4]
ghci> takeWhile (/=' ') "This is a sentence"
"This"
```

Припустимо, ми хочемо порахувати суму всіх натуральних чисел в кубі (себто — чисел, піднесених до третього степеня), які за значенням є менші за 10000. Ми не можемо відобразити за допомогою `(^3)` по списку `[1..]`, профільтрувати, а потім порахувати суму, бо фільтрування нескінченного списку ніколи не завершиться. Ви може й знаєте, що елементи списку `[1..]` подано у висхідному порядку, а от Хаскел — ні. Замість цього ми можемо зробити отак:

```
ghci> sum $ takeWhile (<10000) $ map (^3) [1..]
53361
```

Ми застосовуємо `(^3)` до нескінченного списку, але коли натрапляємо на значення більше за 10000, `takeWhile` обриває його. Проблем із розрахунком суми такого списку не виникатиме.

У функції `dropWhile` схожа поведінка: вона *викидає* зі списку елементи доки предикат є істинним. Щойно предикат поверне значення `False`, `dropWhile` завершує роботу і повертає решту списку. Надзвичайно корисна та чарівна функція!

```
ghci> dropWhile (/=' ') "This is a sentence"
" is a sentence"
ghci> dropWhile (<3) [1,2,2,2,3,4,5,4,3,2,1]
[3,4,5,4,3,2,1]
```

Ми маємо список, який описує зміну вартості цінних паперів в часі. Список складається з кортежів, де перша компонента — це ціна акцій, а друга — рік, третя — місяць, четверта — день. Треба дізнатися, коли вартість акцій вперше перевищила тисячу доларів!

```
ghci> let stock = [(994.4,2008,9,1)
                  ,(995.2,2008,9,2)
                  ,(999.2,2008,9,3)
                  ,(1001.4,2008,9,4)
                  ,(998.3,2008,9,5)]
ghci> head (dropWhile (\(val,y,m,d) -> val < 1000) stock)
(1001.4,2008,9,4)
```

`span` трошечки схожа на `takeWhile`, лише вона повертає пару списків. Перший із списків містить той самий список, що повернула би `takeWhile`, якби її

викликали із тим самим списком і тим самим предикатом. Другий із списків містить частину вхідного списку, яку `takeWhile` викинула б.

```
ghci> let (fw, rest) = span (/=' ') "This is a sentence" in
    "First word:" ++ fw ++ ", the rest:" ++ rest
"First word: This, the rest: is a sentence"
```

Якщо `span` хрумає список доки предикат є істинним, `break` перестає його хрумати, коли предикат вперше набуває значення `True`. Тобто, `break p` є еквівалентом `span (not . p)`.

```
ghci> break (==4) [1,2,3,4,5,6,7]
([1,2,3],[4,5,6,7])
ghci> span (/=4) [1,2,3,4,5,6,7]
([1,2,3],[4,5,6,7])
```

Другий із отриманих списків, що його повертає `break`, починається з першого елемента, який задовольнив умові.

`sort` просто сортує список. Тип елементів цього списку має належати до типокласу `Ord`, бо, якщо елементи списку не можна розташувати в якомусь порядку, то список не може бути відсортований.

```
ghci> sort [8,5,3,2,1,6,4,2]
[1,2,2,3,4,5,6,8]
ghci> sort "This will be sorted soon"
"    Tbdeehiillnooorssstw"
```

`group` бере список та групує елементи-сусіди в підсписки, якщо ті елементи дорівнюють одне одному.

```
ghci> group [1,1,1,1,2,2,2,2,3,3,2,2,2,5,6,7]
[[1,1,1,1],[2,2,2,2],[3,3],[2,2,2],[5],[6],[7]]
```

Якщо відсортувати список перед його групуванням, то можна дізнатися скільки разів кожний з елементів з'являється у списку.

```
ghci> map (\l@(x:xs) -> (x,length l)) . group . sort $
    [1,1,1,1,2,2,2,2,3,3,2,2,2,5,6,7]
[(1,4),(2,7),(3,2),(5,1),(6,1),(7,1)]
```

Функції `inits` та `tails` схожі на `init` та `tail`, окрім того, що вони працюють рекурсивно, і вираховують всі можливі голови і хвости (іншими словами, голови голів і хвости хвостів). Погляньте:

```
ghci> inits "w00t"
["","w","w0","w00","w00t"]
ghci> tails "w00t"
```

```
["w00t","00t","0t","t",""]
ghci> let w = "w00t" in zip (inits w) (tails w)
[("","w00t"),("w","00t"),("w0","0t"),("w00","t"),("w00t","")]
```

Використаймо згорток для пошуку підсписку у списку.

```
search :: Eq a => [a] -> [a] -> Bool
search needle haystack =
    let nlen = length needle
    in  foldl (\acc x -> if take nlen x == needle then True else acc)
            False (tails haystack)
```

Спочатку ми годуємо `tails` списком, в якому виконується пошук — маємо всі можливі недогризки для випадку, коли їсти починаємо той список з голови. Потім ми перевіряємо кожен недогризок — чи починається він з елементів, які ми шукаємо?

У такий спосіб ми власне створили функцію, яка поводиться як `isInfixOf`. `isInfixOf` шукає підсписок у списку та повертає `True`, якщо знаходить.

```
ghci> "cat" `isInfixOf` "im a cat burglar"
True
ghci> "Cat" `isInfixOf` "im a cat burglar"
False
ghci> "cats" `isInfixOf` "im a cat burglar"
False
```

`isPrefixOf` (`isSuffixOf`) перевіряє, чи починається (закінчується) список підсписком.

```
ghci> "hey" `isPrefixOf` "hey there!"
True
ghci> "hey" `isPrefixOf` "oh hey there!"
False
ghci> "there!" `isSuffixOf` "oh hey there!"
True
ghci> "there!" `isSuffixOf` "oh hey there"
False
```

`elem` та `notElem` перевіряють, чи належить даний елемент до списку, чи ні.

`partition` бере список та предикат та повертає пару списків. Перший із цих списків містить всі елементи, що задовольняють умові, а другий — всі інші.

```
ghci> partition (`elem` ['A'..'Z']) "BOBsidneyMORGANeddy"
("BOBMORGAN","sidneyeddy")
ghci> partition (>3) [1,3,5,6,3,2,1,0,3,7]
([5,6,7],[1,3,3,2,1,0,3])
```

Важливо зрозуміти різницю між цією функцією та `span`-ом і `break`-ом:

```
ghci> span (`elem` ['A'..'Z']) "BOBsidneyMORGANeddy"
("BOB","sidneyMORGANeddy")
```

Якщо `span` та `break` завершують роботу, як тільки натинаються на перший елемент, що задовольняє або не задовольняє умові, то `partition` обробляє увесь список та «ділить» його елементи на дві групи, залежно від того, яке значення повертає предикат.

`find` бере список та предикат та повертає перший елемент, для якого значення предикату є істинним. Але вона повертає той елемент, загорнутий в значення `Maybe`. Ми заглибимося в **алгебраїчні типи даних** [algebraic data types] в наступному розділі, але на цій стадії ось що вам треба зрозуміти: є два значення `Maybe` — `Just <<something>>` або `Nothing`[†]. Так само, як список може бути порожнім списком або списком з елементами, `Maybe` може набувати значень «жодного елемента» або «єдиний елемент». І так само, як тип списку цілих чисел є `[Int]`, тип чогось, що *можливо* містить ціле число, є `Maybe Int`. Отже, давайте обкатаємо нашу `find`:

```
ghci> find (>4) [1,2,3,4,5,6]
Just 5
ghci> find (>9) [1,2,3,4,5,6]
Nothing
ghci> :t find
find :: (a -> Bool) -> [a] -> Maybe a
```

Зверніть увагу на тип `find`. Її результатом є `Maybe a`. Це схоже на тип `[a]`, за єдиною відмінністю: значення типу `Maybe` може містити або один елемент, або жодного, а от список може містити один елемент, або жодного, або ж — декілька.

Пам'ятаєте, як ми шукали моменту, коли ціна акцій вперше досягла 1000 доларів? Ми тоді розв'язали цю задачу отак: `head (dropWhile (\(val,y,m,d) -> val < 1000) stock)`. Зауважте, що використання `head` буває небезпечним. Що трапиться у випадку, коли ціна акцій ніколи не перевалить за тисячу доларів? Виконання `dropWhile` поверне пустий список і обрахунок голови пустого списку завершиться аварією. Але якщо переписати вираз як `find (\(val,y,m,d) -> val > 1000) stock`, можна почуватися спокійніше. Якщо ціна акцій ніколи не перевищувала тисячну відмітку (отже жоден з елементів не задовольнить умові), ми отримаємо

[†] «Just» тут вжито в значенні «Лише одне». «Nothing» перекладається як «Нічого». Отож, зрозумілою нам (літературною!) мовою, є два конструктори значень типу «Можливо є, а можливо — ні»: «Одне-єдине саме-самісеньке <<щось>>», або «Зовсім нічогісеньки-нічого».

`Nothing`. Але якщо в списку міститься правильна відповідь, отримаємо, наприклад, `Just (1001.4,2008,9,4)`.

`elemIndex` схожа на `elem`, але вона повертає не булеве значення, а, *можливо*, індекс елемента, що ми його шукаємо. Якщо такого елемента немає в списку, повертається `Nothing`.

```
ghci> :t elemIndex
elemIndex :: Eq a => a -> [a] -> Maybe Int
ghci> 4 `elemIndex` [1,2,3,4,5,6]
Just 3
ghci> 10 `elemIndex` [1,2,3,4,5,6]
Nothing
```

`elemIndices` така сама, як `elemIndex`, але вона повертає список індексів (на випадок, коли потрібний елемент трапляється в списку декілька разів). Оскільки індекси повертаються в списку, нам не потрібен тип `Maybe`, тому що аварійне завершення може бути представлене як пустий список, що є еквівалентним `Nothing`.

```
ghci> ' ' `elemIndices` "Where are the spaces?"
[5,9,13]
```

`findIndex` схожа на `find`, але вона *можливо* повертає індекс першого елемента, який задовольняє умові. `findIndices` повертає індекси всіх елементів, що задовольняють умові, у вигляді списку.

```
ghci> findIndex (==4) [5,3,2,1,6,4]
Just 5
ghci> findIndex (==7) [5,3,2,1,6,4]
Nothing
ghci> findIndices (`elem` ['A'..'Z']) "Where Are The Caps?"
[0,6,10,14]
```

Ми вже обговорювали `zip` та `zipWith`. Зокрема, було зазначено, що вони «застібають» два списки докупи в один список, елементи попарно потрапляють у кортежі, або передаються бінарній функції (функція, що приймає два параметри). А що робити, коли треба застібнути разом три списки? Або застібнути три списки функцією, що приймає три параметри? Ну, для того маємо `zip3`, `zip4` і так далі, а також `zipWith3`, `zipWith4` і так далі. Варіації на цю тему сягають аж сімки. Хоча це рішення виглядає доволі коструватим, насправді працює доволі непогано, тому що не так вже й часто треба застібати вісім списків докупи. До того ж, існує дуже елегантний спосіб застібання нескінченної кількості списків, але ми іще не достатньо розумні для цієї розмови.

```
ghci> zipWith3 (\x y z -> x + y + z) [1,2,3] [4,5,2,2] [2,2,3]
[7,9,8]
ghci> zip4 [2,3,3] [2,2,2] [5,5,3] [2,2,2]
[(2,2,5,2),(3,2,5,2),(3,2,3,2)]
```

Як і із звичайним застібанням (за допомогою `zip`), списки, довші за найкоротший в групі, вкорочуються до довжини найкоротшого.

`lines` використовується при роботі з файлами або якимись вхідними даними. Вона бере рядок та повертає кожний рядок тексту того рядка окремим списком.

```
ghci> lines "first line\nsecond line\nthird line"
["first line","second line","third line"]
```

В Юніксі `'\n'` — це є представлення символу нового рядка тексту[†]. Зворотні скісні риски мають особливе семантичне навантаження в хаскелівських рядках та символах.

`unlines` — то є обернена функція до функції `lines`. Вона приймає список рядків та та з'єднує їх із допомогою `'\n'`.

```
ghci> unlines ["first line", "second line", "third line"]
"first line\nsecond line\nthird line\n"
```

`words` та `unwords` живуть для того, щоб розбивати рядок тексту на слова та з'єднувати список слів у рядок тексту, відповідно. Уособлена корисність!

```
ghci> words "hey these are the words in this sentence"
["hey","these","are","the","words","in","this","sentence"]
ghci> words "hey these           are    the words in this\nsentence"
["hey","these","are","the","words","in","this","sentence"]
ghci> unwords ["hey","there","mate"]
"hey there mate"
```

Ми вже згадували `nub`. Вона бере список та висапує з нього елементи, що повторюються, і повертає список, в якому кожний елемент — хехе — неповторна сніжинка. У цієї функції таке дивне ім'я. Виявляється, що «nub» означає «маленький кусень» або ж «суть» чогось — от! Якщо мене спитати, то Вони мають використовувати Справжні Слова в назвах функцій, а не хтозна-які пристаркуваті.

```
ghci> nub [1,2,3,4,3,2,1,2,3,4,3,2,1]
[1,2,3,4]
ghci> nub "Lots of words and stuff"
```

[†]Зворотні скісні екранують наступний символ, наділяючи його спеціальним значенням і утворюють із ним представлення одного символа.

```
"Lots fwrdanu"
```

`delete` бере елемент та список і вилучає перший примірник того елемента із списку.

```
ghci> delete 'h' "hey there ghang!"
"ey there ghang!"
ghci> delete 'h' . delete 'h' $ "hey there ghang!"
"ey tere ghang!"
ghci> delete 'h' . delete 'h' . delete 'h' $ "hey there ghang!"
"ey tere gang!"
```

`\\` — це функція різниці списків. Вона фактично поводиться як різниця множин. Для кожного елемента списку, що стоїть праворуч від `\\`, вона перевіряє, чи він присутній у списку, що стоїть ліворуч, і, якщо так — видаляє його з лівого списку.

```
ghci> [1..10] \\ [2,5,9]
[1,3,4,6,7,8,10]
ghci> "Im a big baby" \\ "big"
"Im a  baby"
```

Вираз `[1..10] \\ [2,5,9]` еквівалентний

```
delete 2 . delete 5 . delete 9 $ [1..10]
```

Так само, `union` поводиться як об'єднання множин — вона повертає об'єднання двох списків. Фактично вона обробляє кожен з елементів другого списку і приєднує його до першого списку, якщо той елемент ще не присутній у ньому. Іншими словами — стережіться: повторні елементи з другого списку до першого не додаються!

```
ghci> "hey man" `union` "man what's up"
"hey manwt'sup"
ghci> [1..7] `union` [5..10]
[1,2,3,4,5,6,7,8,9,10]
```

`intersect` поводиться як перетин множин. Вона повертає тільки ті елементи, що присутні у обох списках.

```
ghci> [1..7] `intersect` [5..10]
[5,6,7]
```

`insert` приймає елемент та список елементів, які можна відсортувати, та втуляє той елемент в останню з можливих позицію, в якій він все іще менший або дорівнює наступному елементу. Інакше кажучи, `insert` починає з початку списку та переглядає його, доки не знайде елемент, більший або рівний

до елементу, який потрібно вставити, і власне вставляє його прям перед цим елементом.

```
ghci> insert 4 [3,5,1,2,8,2]
[3,4,5,1,2,8,2]
ghci> insert 4 [1,3,4,4,1]
[1,3,4,4,4,1]
```

У першому прикладі `4` було вставлено після `3` та перед `5`, а в другому — між `3` та `4`.

Якщо `insert` вставляє в відсортований список, то список-результат теж буде відсортованим.

```
ghci> insert 4 [1,2,3,5,6,7]
[1,2,3,4,5,6,7]
ghci> insert 'g' $ ['a'..'f'] ++ ['h'..'z']
"abcdefghijklmnopqrstuvwxyz"
ghci> insert 3 [1,2,4,3,2,1]
[1,2,3,4,3,2,1]
```

`length`, `take`, `drop`, `splitAt`, `!!` та `replicate` об'єднує те, що вони всі приймають `Int` як один з параметрів (або повертають `Int`), хоча вони могли б бути більш загальними та зручними в користуванні, аби вони приймали будь-який тип із типокласу `Integral` чи `Num` (залежно від функції). Така поведінка спричинена минулим. Але якщо її виправити, то мабуть зламається багато вже написаного коду. Ось тому `Data.List` містить узагальнені еквіваленти цих функцій `genericLength`, `genericTake`, `genericDrop`, `genericSplitAt`, `genericIndex` та `genericReplicate`. Наприклад, сиґнатура типу `length` — це `length :: [a] -> Int`. Якщо спробувати порахувати середнє списку чисел з допомогою `let xs = [1..6] in sum xs / length xs`, ми отримаємо помилку типу, тому що не можна викликати `/` із `Int`. З іншого боку, сиґнатура типу `genericLength :: Num a => [b] -> a`. І оскільки `Num` може поводитися як число з плаваючою комою, обрахунок середнього із `let xs = [1..6] in sum xs / genericLength xs` спрацьовує без проблем.

У `nub`, `delete`, `union`, `intersect` та `group` теж є більш загальні родичі `nubBy`, `deleteBy`, `unionBy`, `intersectBy` та `groupBy`. Перша група використовує `==` для перевірки рівності, а ось друга група — ватага із суфіксом *By* — приймає функцію рівності як параметр. Отже `group` — це теж саме що і `groupBy (==)`.

Наприклад, у нас є список, що містить значення якоїсь функції в кожну секунду часу. Хай нам треба групувати елементи-сусіди в підсписки так, щоб значення функції більше нуля були разом, а значення менше — разом. Звичай-

на `group` просто погрупує рівнозначні суміжні значення. А нам треба згрупувати залежно від того, чи негативні числа, чи ні. Ось тут нам і знадобиться `groupBy`! Функція рівності, яку приймають панове *By*, візьме два елементи однакового типу і поверне `True`, якщо вважатиме їх рівними за її «стандартом рівності».

```
ghci> let values = [-4.3, -2.4, -1.2, 0.4, 2.3, 5.9, 10.5
                  , 29.1, 5.3, -2.4, -14.5, 2.9, 2.3]
ghci> groupBy (\x y -> (x > 0) == (y > 0)) values
[[-4.3,-2.4,-1.2],[0.4,2.3,5.9,10.5,29.1,5.3],[-2.4,-14.5],[2.9,2.3]]
```

Із результату легко побачити, в яких сегментах списку значення функції додатні, а в яких — від'ємні. Означена нами функція рівності бере два елементи і повертає `True` лише коли вони обидва від'ємні або обидва додатні. Функцію рівності також можна подати як `\x y -> (x > 0) && (y > 0) || (x <= 0) && (y <= 0)`, хоча я вважаю, що попереднє формулювання читабельніше. Ще виразніше записати функцію рівності для функцій *By* можна за допомогою імпорту `Data.Function`. Функцію `on` означено отак:

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

Отже, виконання `(==) `on` (>0)` повертає функцію рівності `\x y -> (x > 0) == (y > 0)`. Функція `on` часто використовується із функціями із суфіксом *By* тому, що разом вони здатні ось на що:

```
ghci> groupBy ((==) `on` (> 0)) values
[[-4.3,-2.4,-1.2],[0.4,2.3,5.9,10.5,29.1,5.3],[-2.4,-14.5],[2.9,2.3]]
```

Правда, читабельно!? Вираз дуже легко озвучити: Згрупуй це за рівністю по ознаці «більше за нуль».

Аналогічно, `sort`, `insert`, `maximum` та `minimum` також мають свої більш загальні еквіваленти. Функції на зразок `groupBy` приймають функцію, що **визначає** [to determine] рівність двох елементів. `sortBy`, `insertBy`, `maximumBy` та `minimumBy` приймають функцію, що визначає, коли один елемент більший за інший, менший за інший, або вони рівнозначні. Сиґнатура типу `sortBy` — це `sortBy :: (a -> a -> Ordering) -> [a] -> [a]`. Пам'ятаєте, тип `Ordering` може мати значення `LT`, `EQ` або `GT`? `sort` еквівалентний до `sortBy compare`, тому що `compare` просто бере два елементи, чий тип належить до типокласу `Ord`, та повертає `Ordering` (тобто — «порядок»).

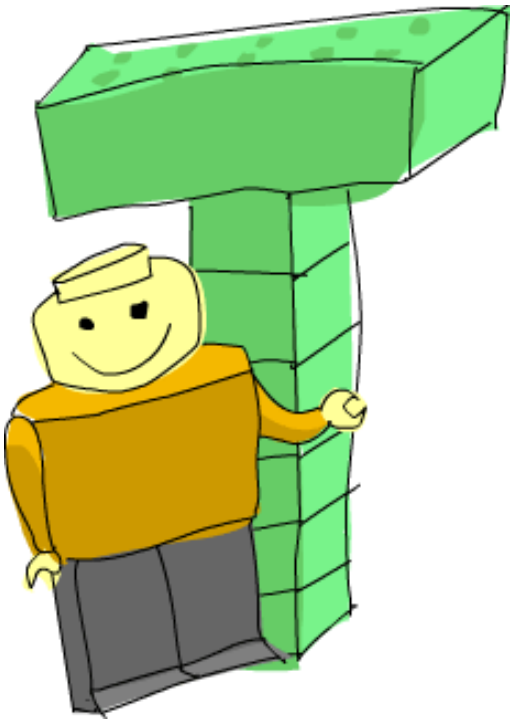
Списки теж можна порівнювати, і таке порівняння відбувається лексикографічно. А що ж робити, коли треба відсортувати список списків, але не за

змістом вкладених списків, а, наприклад, за їхньою довжиною? Ви мабуть вже здогадалися — ми використаємо функцію `sortBy`.

```
ghci> let xs = [[5,4,5,4,4],[1,2,3],[3,5,4,3],[],[2],[2,2]]
ghci> sortBy (compare `on` length) xs
[[],[2],[2,2],[1,2,3],[3,5,4,3],[5,4,5,4,4]]
```

Паморочливо! `compare `on` length` ... їй-бо, це наближається до звичайного речення англійською мовою (перекладається як «порівняти `за` довжиною»). На всяк випадок, пояснюю — `on` тут працює так: вираз `compare `on` length` еквівалентний `\x y -> length x `compare` length y`. Для функцій *By*, які беруть функцію рівності, як правило ми пишемо `(==) `on` <<something>>`, а для функцій *By*, які беруть як параметр функцію впорядкування, ми зазвичай пишемо `compare `on` <<something>>`.

7.3 Data.Char



Модуль `Data.Char`, як не дивно, експортує функції, пов'язані з символами. Він також стає у нагоді, коли треба відфільтрувати рядки або відобразити по них, бо рядки — це ж просто списки символів.

`Data.Char` експортує купу предикатів для роботи із символами. Тобто, функцій, які беруть символ та відповідають, чи є якесь припущення стосовно нього правдивим чи ні. Ось які вони:

`isControl` перевіряє є чи даний символ управляючим символом.

`isSpace` перевіряє чи належить даний символ до символів-пробілів, себто пробілів, знаків табуляції та нового рядка, і такого іншого.

`isLower` перевіряє чи є символ малою літерою.

`isUpper` перевіряє чи символ є заголовною (великою) літерою.

`isAlpha` перевіряє чи є символ літерою.

`isAlphaNum` перевіряє чи є символ літерою або числом.

`isPrint` перевіряє чи є символ друкованим. Керівні символи, наприклад, не друкуються.

`isDigit` перевіряє чи є символ цифрою.

`isOctDigit` перевіряє чи є символ вісімковою цифрою.

`isHexDigit` перевіряє чи є символ шістнадцятковою цифрою.

`isLetter` перевіряє чи є символ літерою[†].

`isMark` перевіряє на наявність діакритичних знаків Unicode. Це знаки, що сполучаються із попередньою літерою для утворення літер із позначками чи наголосами. Словом, для французів.

`isNumber` перевіряє чи є символ цифрою.

`isPunctuation` перевіряє на присутність знаків пунктуації.

`isSymbol` перевіряє чи є даний символ якимось вигадливим математичним чи валютним символом.

`isSeparator` перевіряє чи є даний символ пробілом або символом нової лінії чи нового параграфу.

`isAscii` перевіряє чи належить символ до перших 128 символів з таблиці символів Unicode.

`isLatin1` перевіряє чи належить символ до перших 256 символів з таблиці символів Unicode.

`isAsciiUpper` перевіряє чи належить символ до перших 128 символів з Unicode і одночасно є заголовною літерою.

`isAsciiLower` перевіряє чи належить символ до перших 128 символів з Unicode і одночасно є малою літерою.

Сиґнатурами типу всіх цих предикатів є `Char -> Bool`. Здебільшого вони використовуватимуться для фільтрування рядків. Наприклад, ми пишемо програму, що приймає ім'я користувача. Це ім'я може складатися виключно із літер та цифр. Можна скористатися функцією `all` із модуля `Data.List` разом із предикатами із `Data.Char` аби перевірити «легітимність» імені.

```
ghci> all isAlphaNum "bobby283"
True
ghci> all isAlphaNum "eddy the fish!"
False
```

Пекельно! Якщо пригадуєте, `all` бере предикат і список та повертає `True` лише коли предикат справджується для кожного елемента того списку.

Функцією `isSpace` можна імітувати роботу функції `words` із модуля `Data.List`.

[†]Те саме, що й `isAlpha`, але різна реалізація.

```
ghci> words ”hey guys its me”
[”hey”,”guys”,”its”,”me”]
ghci> groupBy ((==) `on` isSpace) ”hey guys its me”
[”hey”,” ”,”guys”,” ”,”its”,” ”,”me”]
ghci>
```

Хмм, це начебто схоже на роботу `words`, але ж ми залишились з елементами, що містять лише пробіл. І що ж нам робити?..

...

Ідея! — відфільтруємо цю собацюру.

```
ghci> filter (not . any isSpace) . groupBy ((==) `on` isSpace) $
    ”hey guys its me”
[”hey”,”guys”,”its”,”me”]
```

А, відлягло від серця.

The `Data.Char` також експортує тип даних доволі схожий на `Ordering`. Тип `Ordering` може набувати значення `LT`, `EQ` або `GT`. Щось на кшталт переліченя[†], що містить можливі наслідки порівняння двох елементів. Тип `GeneralCategory` — це теж перелічення. Він містить декілька можливих категорій, до яких символ може належати. Основна функція для **визначення** [determination] загальної категорії символу — це `generalCategory`. Її тип — `generalCategory :: Char -> GeneralCategory`. Там є десь 31 категорія, не буду тут усе перелічувати (ненавмисний каламбур!), краще пограємося із цією функцією.

```
ghci> generalCategory ’ ’
Space
ghci> generalCategory ’A’
UppercaseLetter
ghci> generalCategory ’a’
LowercaseLetter
ghci> generalCategory ’.’
OtherPunctuation
ghci> generalCategory ’9’
DecimalNumber
ghci> map generalCategory ” \t\nA9?|”
[Space,Control,Control,UppercaseLetter
,DecimalNumber,OtherPunctuation,MathSymbol]
```

[†]Перелічення (enumeration; enumerated type) — тип даних, множиною значень якого є невпорядкована скінченна множина.

Оскільки тип `GeneralCategory` належить до типокласу `Eq`, ми можемо тестувати на кшталт `generalCategory c == Space`.

`toUpper` перетворює літеру на велику. Пробіли, числа і все таке інше лишаються незмінними.

`toLower` перетворює літери на малі літери.

`toTitle` переводить символ у заголовний регістр. Для більшості символів, заголовний регістр є те саме, що й верхній регістр[†].

`digitToInt` перетворює символ на `Int`. Перетворення успішно виконується лише коли даний символ належить до діапазонів `'0'..'9'`, `'a'..'f'` чи `'A'..'F'` (шістнадцяткові числа).

```
ghci> map digitToInt "34538"
[3,4,5,3,8]
ghci> map digitToInt "FF85AB"
[15,15,8,5,10,11]
```

`intToDigit` — обернена функція до `digitToInt`. Вона бере `Int` у діапазоні `0..15` та перетворює його на символ (повертається в нижньому регістрі, якщо символ — не цифра).

```
ghci> intToDigit 15
'f'
ghci> intToDigit 5
'5'
```

Функції `ord` та `chr` перетворюють символи у відповідні їм числа і навпаки:

```
ghci> ord 'a'
97
ghci> chr 97
'a'
ghci> map ord "abcdefgh"
[97,98,99,100,101,102,103,104]
```

Різниця між значеннями `ord` для двох символів дорівнює відстані між ними в таблиці Unicode.

Шифр Цезаря — це примітивний метод шифрування повідомлень зсувом кожної з літер на фіксовану кількість позицій у абетці. Ми можемо легко ство-

[†]Заголовний регістр (titlecase) — спосіб форматування речення, у якому перша літера кожного неслужбового слова є заголовною, а решта літер — малі. Іноді знаки двох літер об'єднуються в один друкований символ (в так звані лігатури) для гармонійнішого вигляду шрифту в наборі. Якщо слово починається з лігатури, перша літера може «писатися» порізному, залежно від того, чи є наступна літера заголовною, чи малою. Ось тому поняття заголовний регістр і «переповзло» на рівень літер.

рити свій власний шифр, схожий на шифр Цезаря, де ми не будемо обмежувати себе символами з абетки.

```
encode :: Int -> String -> String
encode shift msg =
    let ords = map ord msg
        shifted = map (+ shift) ords
    in  map chr shifted
```

Отже, спочатку перетворюємо рядок на список чисел. Потім додаємо зсув до кожного числа, і перетворюємо цей новий список чисел назад у рядок. Пасіонарії композиції серед вас можуть переписати тіло цієї функції як `map (chr . (+shift) . ord) msg`. Зашифруймо декілька повідомлень.

```
ghci> encode 3 "Heeeeey"
"Khhhhh|"
ghci> encode 4 "Heeeeey"
"Liiiii}"
ghci> encode 1 "abcd"
"bcde"
ghci> encode 5 "Marry Christmas! Ho ho ho!"
"Rfww~%Hmwnxyrfx&%Mt%mt%mt&"
```

Здається зашифрувалося. Повідомлення розшифровуються фактично поверненням літер на свої місця, себто, зсувом назад на ту саму кількість позицій, із якою було зашифровано.

```
decode :: Int -> String -> String
decode shift msg = encode (negate shift) msg
```

```
ghci> encode 3 "Im a little teapot"
"Lp#d#olwwoh#whdsrw"
ghci> decode 3 "Lp#d#olwwoh#whdsrw"
"Im a little teapot"
ghci> decode 5 . encode 5 $ "This is a sentence"
"This is a sentence"
```

7.4 Data.Map

Асоціативні списки — це списки, що використовуються для зберігання пар ключ-значення, без зазначення порядку, в якому вони мають бути збережені. Наприклад, асоціативний список можна пристосувати для зберігання телефонних номерів, де номер телефону — це значення, а ім'я абонента — це ключ.

Байдуже, в якому порядку вони зберігаються, нам лише важливо, що кожній людині в списку відповідає «правильний» номер телефону (себто — її власний номер, а не чийсь інший).

Найпростіше представлення асоціативних списків в Хаскелі — це список пар. Перша складова пари — це ключ, а друга — значення. Ось асоціативний список телефонних номерів:

```
phoneBook =
     [("betty","555-2938")
    ,("bonnie","452-2928")
    ,("patsy","493-2928")
    ,("lucille","205-2928")
    ,("wendy","939-8282")
    ,("penny","853-2492")
    ]
```

Незважаючи на начебто дивні відступи, це просто список пар рядків. Серед операцій із асоціативними списками найчастіше трапляється пошук значення за ключем. Тож побудуймо функцію, що знаходить значення за ключем.

```
findKey :: Eq k => k -> [(k,v)] -> v
findKey key xs = snd . head . filter (\(k,v) -> key == k) $ xs
```

Не занадто складно. Функція бере ключ і список, відфільтровує список так, щоб лишилися тільки підходящі ключі, хапає першу відповідну пару ключ-значення та повертає значення з неї. А що трапиться, коли потрібний нам ключ не міститься у асоціативному списку? Гммм. Тут, якщо ключа немає в асоціативному списку, ми спробуємо отримати голову пустого списку і спіймаємо облизня, тобто, — помилку виконання. Але ж нам не варто писати програми, які так легко завалюються, тому використаймо тут тип даних `Maybe`. Якщо ключ не знайшовся, повернемо `Nothing`. Якщо знайшовся — повернемо `Just <<something>>`, де `<<something>>` — це значення, що відповідає тому ключеві.

```
findKey :: Eq k => k -> [(k,v)] -> Maybe v
findKey key [] = Nothing
findKey key ((k,v):xs) = if key == k
                         then Just v
                         else findKey key xs
```

Погляньте на оголошення типу. Функція бере ключ, який можна порівнювати, та асоціативний список і *можливо* повертає значення. Виглядає непогано.

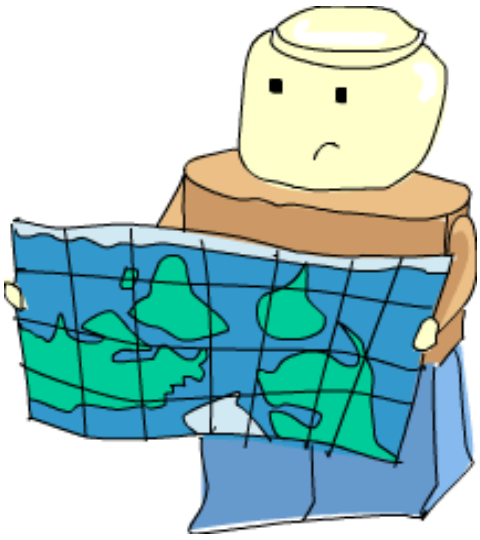
Це стандартна рекурсивна функція, що працює із списком. Крайовий випадок, розбиття списку на голову та хвіст, рекурсивні виклики — всі тут родичі

гарбузові. Це класична ідіома із використанням згортка, отже погляньмо, як можна реалізувати це за його допомогою.

```
findKey :: Eq k => k -> [(k,v)] -> Maybe v
findKey key = foldr (\(k,v) acc -> if key == k then Just v else acc) Nothing
```

Примітка: Зазвичай краще використовувати згортки для таких стандартних рекурсивних задач ніж явно виписувати рекурсію, тому що їх легше читати та розпізнавати. Всі знають, що це згорток, коли бачать виклик `foldr`, а для читання (і розуміння) явної рекурсії треба трохи більше напружувати мізки.

```
ghci> findKey "penny" phoneBook
Just "853-2492"
ghci> findKey "betty" phoneBook
Just "555-2938"
ghci> findKey "wilma" phoneBook
Nothing
```



Прегарно спрацьовує! Якщо ми маємо номер телефону дівчини, нам його просто повернуть (`Just` означає «просто»), а якщо ні — зась (`Nothing` означає «нічого»)!

Ми щойно реалізували функцію `lookup` із `Data.List`. Якщо треба знайти значення, що відповідає певному ключеві, ми маємо йти тим списком, елемент за елементом, аж доки не знайдемо його. Модуль `Data.Map` пропонує набагато більш швидкі асоціативні контейнери — так звані мапи (більш швидкі, бо вони реалізовані на базі дерев), разом із купою допоміжних функцій. Відтепер, ми будемо казати «мапа», коли працюватимемо з асоціативними списками, бо будемо насправді користуватися більш швидким аналогом асоціативних списків — асоціативним контейнером з модуля `Data.Map`.

Оскільки функції з `Data.Map` конфліктують із функціями з `Prelude` та `Data.List`, ми зробимо імпорт в підпростір імен.

```
import qualified Data.Map as Map
```

Додайте цю інструкцію імпорту в скрипт та завантажте його у GHCi.

Отже наважимося і поглянемо на скарби `Data.Map`! Ось декілька функцій з цього модуля, із коротеньким описанням.

Функція `fromList` бере асоціативний список (у вигляді хаскельного списку) та повертає мапу з тими самими «асоціаціями».

```
ghci> Map.fromList [("betty","555-2938")
                   ,("bonnie","452-2928")
                   ,("lucille","205-2928")]
fromList [("betty","555-2938"),("bonnie","452-2928"),("lucille","205-2928")]
ghci> Map.fromList [(1,2),(3,4),(3,2),(5,5)]
fromList [(1,2),(3,2),(5,5)]
```

Якщо у вихідному асоціативному списку деякі ключі повторюються, дублікати при побудові мапи просто відкидаються. Ось така сиґнатура типу `fromList`:

```
Map.fromList :: Ord k => [(k, v)] -> Map.Map k v
```

Вона каже, що ця функція бере список пар, де тип першого елемента пари є `k`, а другого — `v`, та повертає мапу, яка відображає ключі, що мають тип `k`, в значення, що мають тип `v`. Зверніть увагу: коли ми працюємо із асоціативними списками на базі простих списків, ми лише вимагаємо, щоб ключі можна було порівнювати (їхній тип має належати до типокласу `Eq`), але тепер треба, щоб ключі можна було ще й впорядкувати (типоклас `Ord`). В модулі `Data.Map` це є дуже важлива умова — якщо ключі не можна впорядкувати, з них не можна побудувати дерево.

Раджу завжди використовувати `Data.Map` для асоціювання ключів і значень, за винятком ситуацій, коли ключі не належать до типокласу `Ord`.

`empty` реалізує пусту мапу. Ця функція не приймає аргументів, а просто повертає пусту мапу.

```
ghci> Map.empty
fromList []
```

`insert` бере ключ, значення і мапу та повертає нову мапу, точнісінько таку, як стара, от лише додано нову асоціацію — стару мапу оновлено парою ключ-значення, що було передано `insert`.

```
ghci> Map.empty
fromList []
ghci> Map.insert 3 100 Map.empty
fromList [(3,100)]
ghci> Map.insert 5 600 (Map.insert 4 200 ( Map.insert 3 100  Map.empty))
```

```
fromList [(3,100),(4,200),(5,600)]
ghci> Map.insert 5 600 . Map.insert 4 200 . Map.insert 3 100 $ Map.empty
fromList [(3,100),(4,200),(5,600)]
```

Ми можемо реалізувати власну `fromList`, із використанням пустої мапи, функції `insert` та згортка. Погляньте-но:

```
fromList' :: Ord k => [(k,v)] -> Map.Map k v
fromList' = foldr (\(k,v) acc -> Map.insert k v acc) Map.empty
```

Цей згорток — простий як двері: починаємо із пустої мапи і згортаємо її справа, по дорозі вставляючи пари ключ-значення в накопичувач.

`null` перевіряє чи мапа є пустою.

```
ghci> Map.null Map.empty
True
ghci> Map.null $ Map.fromList [(2,3),(5,5)]
False
```

`size` доповідає про розмір мапи (кількість асоціацій).

```
ghci> Map.size Map.empty
0
ghci> Map.size $ Map.fromList [(2,4),(3,3),(4,2),(5,4),(6,4)]
5
```

`singleton` бере ключ та значення та створює мапу, що містить одну-єдину асоціацію.

```
ghci> Map.singleton 3 9
fromList [(3,9)]
ghci> Map.insert 5 9 $ Map.singleton 3 9
fromList [(3,9),(5,9)]
```

`lookup` працює як `Data.List`-івська `lookup`, лише вона оперує з мапами. Вона повертає `Just <<something>>`, якщо знаходить ключ, та `Nothing`, якщо не знаходить.

`member` — це предикат, який бере ключ та мапу і доповідає, чи ключ є присутній у цій мапі, чи ні.

```
ghci> Map.member 3 $ Map.fromList [(3,6),(4,3),(6,9)]
True
ghci> Map.member 3 $ Map.fromList [(2,5),(4,5)]
False
```

`map` та `filter` поводяться так само як їхні списківські еквіваленти.

```
ghci> Map.map (*100) $ Map.fromList [(1,1),(2,4),(3,9)]
fromList [(1,100),(2,400),(3,900)]
ghci> Map.filter isUpper $ Map.fromList [(1,'a'),(2,'A'),(3,'b'),(4,'B')]
fromList [(2,'A'),(4,'B')]
```

`toList` — обернена функція до `fromList`.

```
ghci> Map.toList . Map.insert 9 2 $ Map.singleton 4 3
[(4,3),(9,2)]
```

`keys` та `elems` повертають списки ключів та значень, відповідно. `keys` еквівалентна до `map fst . Map.toList`; `elems` — до `map snd . Map.toList`.

`fromListWith` — спритна маленька функційка! Вона працює наче як `fromList`, але вона не викидає повторні ключі, а обробляє їх згідно із функцією, яка їй надається. Припустимо, у якоїсь пані може бути декілька номерів, і наш асоціативний список виглядає як:

```
phoneBook =
    [("betty","555-2938")
    ,("betty","342-2492")
    ,("bonnie","452-2928")
    ,("patsy","493-2928")
    ,("patsy","943-2929")
    ,("patsy","827-9162")
    ,("lucille","205-2928")
    ,("wendy","939-8282")
    ,("penny","853-2492")
    ,("penny","555-2111")
    ]
```

Якщо використати просто `fromList` аби перетворити то на мапу, ми ж загубимо декілька номерів! Отже, ось що ми зробимо:

```
phoneBookToMap :: Ord k => [(k, String)] -> Map.Map k String
phoneBookToMap xs =
    Map.fromListWith (\number1 number2 -> number1 ++ ", " ++ number2) xs
```

```
ghci> Map.lookup "patsy" $ phoneBookToMap phoneBook
"827-9162, 943-2929, 493-2928"
ghci> Map.lookup "wendy" $ phoneBookToMap phoneBook
"939-8282"
ghci> Map.lookup "betty" $ phoneBookToMap phoneBook
"342-2492, 555-2938"
```

Якщо було знайдено дублікат ключа, нами надана функція комбінує відповідні значення в нове. Як альтернатива: ми, звичайно ж, могли б зробити всі значення в асоціативному списку односписками, а потім, за допомогою `++`, поєднати всі номери в один список.

```
phoneBookToMap :: Ord k => [(k, a)] -> Map.Map k [a]
phoneBookToMap xs = Map.fromListWith (++) $ map (\(k,v) -> (k,[v])) xs
```

```
ghci> Map.lookup "patsy" $ phoneBookToMap phoneBook
["827-9162","943-2929","493-2928"]
```

Зграбно, чи не так? Іще одна типова ситуація: ми перетворюємо асоціативний список номерів на мапу і, коли трапляється подвійний ключ, залишаємо найбільше із відповідних значень.

```
ghci> Map.fromListWith max [(2,3),(2,5),(2,100),(3,29)
                           ,(3,22),(3,11),(4,22),(4,15)]
fromList [(2,100),(3,29),(4,22)]
```

А можливо нам заманеться додавати докупи значення однакових ключів?

```
ghci> Map.fromListWith (+) [(2,3),(2,5),(2,100),(3,29)
                           ,(3,22),(3,11),(4,22),(4,15)]
fromList [(2,108),(3,62),(4,37)]
```

`insertWith` має такі взаємини із `insert`, які має `fromListWith` із `fromList`. Вона вставляє в мапу пару ключ-значення, але, у випадку, коли такий ключ уже присутній у мапі, вона використовує надану їй функцію аби вирішити, що робити.

```
ghci> Map.insertWith (+) 3 100 $ Map.fromList [(3,4),(5,103),(6,339)]
fromList [(3,104),(5,103),(6,339)]
```

Це лише декілька функцій з модуля `Data.Map`. Повний список функцій можна переглянути отут.

7.5 Data.Set

Модуль `Data.Set` пропонує нам, як не дивно, множини. Множини, як в математиці. Множини — це наче гібрид між списками і мапами. Кожен із елементів множини неповторний. І оскільки в Хаскелі множини закодовані із допомогою дерев (точнісінько як і `Data.Map`), елементи в них є впорядковані. Перевірка

на приналежність до множини, додавання або видалення виконуються набагато швидше, ніж із списками. Працюючи із множинами, найчастіше доводиться додавати елементи до множини, перевіряти на приналежність, та перетворювати множину на список.

Оскільки імена в `Data.Set` конфліктують із багатьма іменами в `Prelude` та `Data.List`, ми зробимо імпорт в підпростір імен.

Додайте цю інструкцію імпорту до скрипта:

```
import qualified Data.Set as Set
```

та завантажте скрипт у GHCi.

Нехай у нас буде два шматки тексту і ми хочемо дізнатися, які символи присутні в обох з них.

```
text1 = "I just had an anime dream. Anime... "
      ++ "Reality... Are they so different?"
text2 = "The old man left his garbage can out "
      ++ "and now his trash is all over my lawn!"
```

Ви вже здогадались, що робить функція `fromList`? (Для тих, хто не здогадався: вона бере список та перетворює його на множину.)

```
ghci> let set1 = Set.fromList text1
ghci> let set2 = Set.fromList text2
ghci> set1
fromList " .?AIRadefhijlmnorstuy"
ghci> set2
fromList " !Tabcdefghilmnorstuvwy"
```

Як бачите, елементи впорядковані і кожен із них неповторний. Тепер можна скористатися функцією `intersection` аби визначити, які в них елементи спільні.

```
ghci> Set.intersection set1 set2
fromList " adefhilmnorstuy"
```

Можна також вжити функцію `difference` аби визначити, які із символів першої множини відсутні у другій, і навпаки.

```
ghci> Set.difference set1 set2
fromList ".?AIRj"
ghci> Set.difference set2 set1
fromList "!Tbcgvw"
```

Або можна познаходити всі неповторні літери, що є присутні в обох реченнях, за допомогою `union`.

```
ghci> Set.union set1 set2
fromList " !.?AIRTabcdefghijlmnorstuvwy"
```

Функції `null`, `size`, `member`, `empty`, `singleton`, `insert` і `delete` все там перевіряють, додають і видаляють, так як і має бути.

```
ghci> Set.null Set.empty
True
ghci> Set.null $ Set.fromList [3,4,5,5,4,3]
False
ghci> Set.size $ Set.fromList [3,4,5,3,4,5]
3
ghci> Set.singleton 9
fromList [9]
ghci> Set.insert 4 $ Set.fromList [9,3,8,1]
fromList [1,3,4,8,9]
ghci> Set.insert 8 $ Set.fromList [5..10]
fromList [5,6,7,8,9,10]
ghci> Set.delete 4 $ Set.fromList [3,4,5,4,3,4,5]
fromList [3,5]
```

Також можна перевіряти на підмножини або власну підмножину. Множина A є підмножиною множини B, коли B містить всі елементи з A. Множина A є власною підмножиною множини B, коли B містить всі елементи з A, але $A \neq B$.

```
ghci> Set.fromList [2,3,4] `Set.isSubsetOf` Set.fromList [1,2,3,4,5]
True
ghci> Set.fromList [1,2,3,4,5] `Set.isSubsetOf` Set.fromList [1,2,3,4,5]
True
ghci> Set.fromList [1,2,3,4,5] `Set.isProperSubsetOf`
    Set.fromList [1,2,3,4,5]
False
ghci> Set.fromList [2,3,4,8] `Set.isSubsetOf` Set.fromList [1,2,3,4,5]
False
```

Ми також можемо `map`-увати множини та `filter`-увати їх.

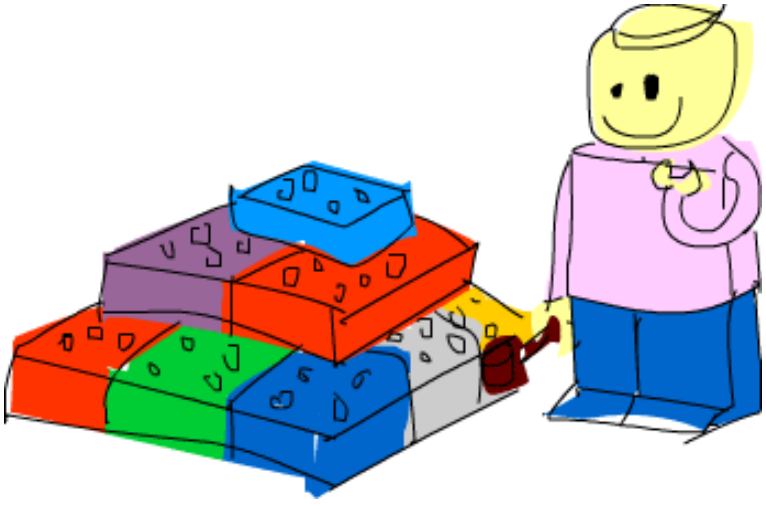
```
ghci> Set.filter odd $ Set.fromList [3,4,5,6,7,2,3,4]
fromList [3,5,7]
ghci> Set.map (+1) $ Set.fromList [3,4,5,6,7,2,3,4]
fromList [3,4,5,6,7,8]
```

Множини часто використовуються аби «виполоти» зі списку елементи, що повторюються, спочатку створивши із нього множину із застосуванням `fromList`, а потім навпаки — список за допомогою `toList`. `Data.List`-івська функція `nub` може сама це робити, але видалення повторів з великих списків набагато швидше, якщо запхнути їх у множину, а потім повернути знов у список, ніж використовувати `nub`. Але знову ж є одне але: `nub` лише вимагає від елементів списку членства у типокласі `Eq`, а от увійти в ексклюзивний клуб множин можуть лише типи, наділені титулом `Ord`.

```
ghci> let setNub xs = Set.toList $ Set.fromList xs
ghci> setNub "HEY WHATS CRACKALACKIN"
" ACEHIKLNRSTWY"
ghci> nub "HEY WHATS CRACKALACKIN"
"HEY WATSCRKLIN"
```

Іще одна деталь. Загалом `setNub` моторніший за `nub` для великих списків, але, як бачите, `nub` зберігає впорядкування елементів списку, а `setNub` — ні.

7.6 Майстрування власних модулів



Ми вже розглянули декілька готових некепських модулів, але ж як зробити власний модуль? Майже будь-яка мова програмування дозволяє розкинути код по декількох файлах, і Хаскел — не виключення. Серед програмістів вважається добрим правилом відокремлювати функції та типи, об'єднані спільною метою, у модуль. Таким чином дуже легко перевикористати ці функції в інших програмах — треба просто зробити імпорт цього модуля.

Отже, для практики, побудуймо собі власний модуль, що міститиме функції обрахунку об'єму та площі деяких геометричних фігур. Для початку створімо файл `Geometry.hs`.

От, ми кажемо — модуль *експортує* функції. Це значить, що коли я імпортую той модуль, я можу користуватися функціями, що він експортує. Модуль може означувати й функції, які не експортуються і використовуються виключно іншими функціями з того ж модуля.

На початку модуля зазначається його ім'я. Якщо файл називається `Geometry.hs`, то наш модуль має називатися `Geometry`. Потім ми зазначаємо, які функції модуль експортує, а потім власне писатимемо означення функцій. Наш модуль починатиметься отак:

```
module Geometry
( sphereVolume
, sphereArea
, cubeVolume
, cubeArea
, cuboidArea
, cuboidVolume
) where
```

Як бачимо, рахуватимемо площу та об'єм для куль, кубів та прямокутних паралелепіпедів. Отже, візьмемося та означимо наші функції:

```
module Geometry
( sphereVolume
, sphereArea
, cubeVolume
, cubeArea
, cuboidArea
, cuboidVolume
) where

sphereVolume :: Float -> Float
sphereVolume radius = (4.0 / 3.0) * pi * (radius ^ 3)

sphereArea :: Float -> Float
sphereArea radius = 4 * pi * (radius ^ 2)

cubeVolume :: Float -> Float
cubeVolume side = cuboidVolume side side side

cubeArea :: Float -> Float
cubeArea side = cuboidArea side side side

cuboidVolume :: Float -> Float -> Float -> Float
cuboidVolume a b c = rectangleArea a b * c
```

```
cuboidArea :: Float -> Float -> Float -> Float
cuboidArea a b c = rectangleArea a b * 2
                 + rectangleArea a c * 2
                 + rectangleArea c b * 2

rectangleArea :: Float -> Float -> Float
rectangleArea a b = a * b
```

Тут пересічна геометрія без всяких приколів. Але є декілька деталей, на які варто звернути увагу. Оскільки куб — це особливий випадок прямокутного паралелепіпеда, ми означили його площу та об'єм, як для паралелепіпеда з всіма рівними ребрами. Ми також означили допоміжну функцію `rectangleArea`, яка рахує площу прямокутника, використовуючи довжини його ребер. Все дуже просто, бо все це — просто операції множення. Зауважте, що ми використали `rectangleArea` в інших функціях модуля, а саме в `cuboidArea` та `cuboidVolume`, але ми її не експортуємо! Оскільки ми хочемо, аби наш модуль надавав функції для роботи лише із тривимірними фігурами, ми використали `rectangleArea`, але не експортували її.

Коли будують модуль, то зазвичай експортують лише ті функції, які грають роль інтерфейсу до модуля, аби деталі реалізації були приховані. Коли хтось користується модулем `Geometry`, їх не обходять функції, які ми не експортуємо. Нам, можливо, спаде на думку змінити кардинально ті функції, або взагалі видалити (наприклад видалити `rectangleArea` і замість цього просто використовувати `*`), і ніхто не обуриться, адже ми не експортували ті функції з самого початку, і тому ніхто з користувачів того модуля їх не бачив.

Аби почати роботу із нашим модулем, просто зробіть:

```
import Geometry
```

Зауважте, що файл `Geometry.hs` має розташовуватися в тій самій директорії, що й програма, яка його імпортує.

Модулі також можуть мати ієрархічну структуру. Модуль може містити кілька підмодулів, а ті в свою чергу — власні підмодулі. Розіб'ємо наші функції таким чином, щоб у модулі `Geometry` були три підмодулі, по одному на фігуру.

Спочатку створімо директорію `Geometry`. Зауважте, що літера G велика. До неї додамо три файли: `Sphere.hs`, `Cuboid.hs` та `Cube.hs`. Ось зміст цих файлів:

`Sphere.hs`

```
module Geometry.Sphere
( volume
, area
```

```
) where

volume :: Float -> Float
volume radius = (4.0 / 3.0) * pi * (radius ^ 3)

area :: Float -> Float
area radius = 4 * pi * (radius ^ 2)
```

`Cuboid.hs`

```
module Geometry.Cuboid
( volume
, area
) where

volume :: Float -> Float -> Float -> Float
volume a b c = rectangleArea a b * c

area :: Float -> Float -> Float -> Float
area a b c = rectangleArea a b * 2
           + rectangleArea a c * 2
           + rectangleArea c b * 2

rectangleArea :: Float -> Float -> Float
rectangleArea a b = a * b
```

`Cube.hs`

```
module Geometry.Cube
( volume
, area
) where

import qualified Geometry.Cuboid as Cuboid

volume :: Float -> Float
volume side = Cuboid.volume side side side

area :: Float -> Float
area side = Cuboid.area side side side
```

Та-а-к, з утіхи затираємо руки. Спочатку, `Geometry.Sphere`. Зауважте, як ми розмістили цей файл у директорії `Geometry`, а потім означили назву модуля `Geometry.Sphere`. Те ж саме і для паралелепіпеда. Також, зауважте, що у всіх

трьох підмодулях ми означили функції з однаковими іменами. Ми можемо так робити, бо вони є окремими модулями. Але тепер, якщо забажалося використати функції з `Geometry.Cuboid` у `Geometry.Cube`, то не можна просто зробити `import Geometry.Cuboid`, тому що цей модуль експортує функції з такими ж іменами як і `Geometry.Cube`. Але в нас є імпорт у підпростір імен! Імпортуємо в підпростір — і все в житті знову добре.

Отже, у файлі, що знаходиться в тій самій директорії, що й `Geometry`, ми можемо, наприклад, надряпати:

```
import Geometry.Sphere
```

І потім можемо викликати `area` і `volume` і вони будуть нам рахувати площу та об'єм кулі. А коли хочете використовувати декілька підмодулів одночасно, робіть імпорт у підпростір імен. Щось на кшталт:

```
import qualified Geometry.Sphere as Sphere
import qualified Geometry.Cuboid as Cuboid
import qualified Geometry.Cube as Cube
```

І тепер можемо викликати `Sphere.area`, `Sphere.volume`, `Cuboid.area` і так далі, і кожна буде нам рахувати площу чи об'єм відповідної геометричної фігури.

Якщо в майбутньому раптом помітите, що працюєте із здоровезним файлом із великою кількістю функцій, придивіться, чи не об'єднує деякі функції спільна мета, і чи не можна їх виокремити у модуль. Якщо вдасться винести цей функціонал в окремий модуль — не тільки спроститься ваше фараонівське творіння (те, звідки ви цей код прибрали), але й підвищиться модульність, і отже, наступного разу, можна буде просто імпортувати той модуль, якщо стане потреба у таких самих функціях, а не перебудовувати ваші піраміди з нуля.

Покажчик