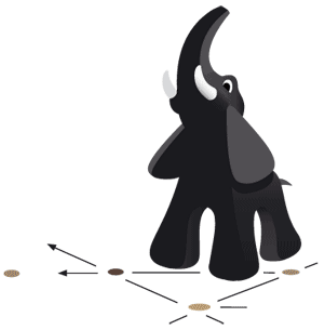
Вивчить собі Хаскела на велике щастя!

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



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

Зміст

Переклад присвячено Національному університетові «Києво-Могилянська академія».

Передн̀є слово

0.1 Щодо мотивації

Мотивацій створення української версії «Learn You a Haskell for Great Good!» було декілька. Першопочатково дуже хотілося привернути увагу українського програміста до цієї абсолютно дивовижної мови і допомогти її опанувати. Хаскел — сама по собі непроста мова програмування, і бар'єрів розуміння там предостатньо й так — тож нам дуже хотілося прибрати хоча б один із них — мовний.

У Великобританії Хаскел викладають на перших курсах деяких університетів. Першокурсники — і ті, хто вже добре програмує імперативно, і ті, хто взагалі ще не програмував, — на цих курсах починають разом «з нуля». І синтаксис, і абстракції, і функційний стиль — все це настільки далеке від «звичайного», що, вкупі із браком досвіду і знань, отримуємо старт без гандикапу для усіх. Це допомагає як викладачам, так і студентам. Але є іще одне: **закон першості** [primacy law] в навчанні: речі, вивчені уперше, справляють найсильніше враження, яке потім дуже важко стерти. Тому вивчити C, а потім Хаскел — це не одне й те саме, що вивчити Хаскел, а потім C.

В мене є мрія, щоб Хаскел почали викладати і в українських навчальних закладах — і цей переклад є одним із кроків до її здійснення. В Могилянці ми писали на Fortran, C/C++, Mathematica і Maple, а Хаскел я вивчив через десять років власноруч. Але якби я міг повернутися назад в 1997-ий, я б вивчив саме Хаскел. Чому? Хаскел — це фундаментальна, безкомпромісна мова, яка вимагає якісних програм, а не просто «дає змогу писати якісні програми». Вчити інші мови програмування, а потім все життя запам'ятовувати ідіоми, які дозволяють не вистрелити собі у ногу в невдалий момент — це один із неоптимальних шляхів до програмістської нірвани. Кращий шлях — вивчити функційне програмування одразу і перейти на новий рівень в боротьбі із **складністю** [complexity], якою і є, по суті, програмістське життя.

Але мені хотілося б, щоб студенти вчили Хаскел не тільки тому, що він такий класний. Насправді, Хаскел має й недоліки (в кого їх немає?). Головним є те, що Хаскел документує чесноти, які варто берегти, і власне розуміння то-

го, із чим Хаскел бореться і за що воює, — це вже неабиякий крок до **просвіти** [enlightenment], незалежно від того, чи перемагає Хаскел в цій боротьбі, чи програє, і це стане в пригоді навіть якщо ви далі будете писати код якимись іншими мовами.

0.2 Про цей переклад

Половину цієї книги переклала Ганна Лелів — і без її внеску ми б ніколи не завершили переклад. Іноді вона перекладала швидше, ніж ми вичитували і компілювали сирці LaTeX. Так чи так — Ганна Лелів або суперлюдина суперперекладач, або це команда перекладачів, яка працює під цим ім'ям.

Ми намагалися не надавати переваги усталеній термінології тільки тому, що вона усталена, а обирали термінологію, балансуючи усталеність із доречністю. Через то деяку термінологію було розроблено спеціально для цього перекладу. Неабиякий внесок в розробку нових термінів зробила Тетяна Богдан (всі терміни, які вам здаються влучними і доречними, розробила вона, а невдала термінологія — то є насправді компроміс, який було досягнуто в результаті довгої редакторської боротьби із іншими редакторами і читачами :D). Тетяна розробила набагато більше термінів, ніж нам вдалося тут використати, але ми їх усіх десь зберегли. Тож, якщо вам потрібен новий термін — напишіть нам — можливо він в нас вже є!

Оригінал містить неабияку кількість жартів і вони, разом із малюнками й стилем подання матеріалу, і роблять цю книгу цією книгою. Більшість з цих жартів перекласти «в лоб» було неможливо. Для деяких вдалося знайти заміну «на місці» — наприклад, підрозділ шостого розділу «Згортком і батька легше бити» в оригіналі називався «Лише згортки і коні», перекручуючи фразу «Лише дурні і коні» («Only fools and horses»). Хай там як — чи то «дурням везе на перегонах», чи то «тільки дурні й коні працюють» — в оригінал цієї книги ця фраза потрапила лише тому, що (1) вона дуже розповсюджена в англомовних колах (ще й британський сітком так називається) і (2) бо слова «fool» і «fold» трохи схожі. В таких випадках ми намагалися знайти схожий за стилем український відповідник. Але бувало, що жарти занадто «впліталися» в зміст книги, і «прикрутити» український жарт в тому ж місці в перекладі означало б менш природні формулювання чи втрату змісту заради жарту. В таких випадках ми перекладали зміст із втратою жарту, але намагалися додати власний жарт десь іще (закон збереження кількості жартів в LUaH :-)).

Багато неточностей в перекладі траплялося через плутанину навколо дієслів означа́ти, озна́чити, озна́чувати, познача́ти, позна́чити, визнача́ти і ви́-

значити[*]. Хоча в українській мові іменники «означе́ння» і «ви́значення» для деяких значень цих багатозначних слів[†] є взаємозамінними еквівалентами, в цій книзі ми використовували іменник «означе́ння» суто для перекладу англійського іменника «definition». Відповідні дієслова до іменника «означе́ння» — «означи́ти» і «означа́ти» (доконаний і недоконаний види, відповідно) — використовувалися отак: «We need to define…» перекладалося доконаним «Нам треба означи́ти…», в той час як *процес* надання означення перекладався недоконаним «означа́ти» (а не підступним «означа́ти»[‡]!): «When defininings functions we…» — «Коли ми означа́ємо функції…». Загальне «We define…» також перекладалося недоконаним: «We define this only when…» — «Ми означа́ємо це лише коли…». Щодо іменника «ви́значення», то він використовувався виключно в значенні «процес з'ясовування чогось», як-от, наприклад, в «оператор визначення зони видимості» чи в «функція для визначення загальної категорії». Відповідні йому дієслова — ви́значити і визнача́ти — перекладали англійські дієслова «to determine», «to figure out», «to find out» і таке інше. Синонімічне «to infer» не перекладалося як «визнача́ти» — скрізь використовувалося більш точне в контексті Хаскела «вивóдити». Дієслово «означа́ти» перекладало англійське «to mean», а «познача́ти» і «позна́чити» — «to denote».

Англомовний варіант називає конструкції із ключовими словами instance і class оголошеннями (instance declarations і typeclass declarations, відповідно), а ми перекладаємо їх як instance-означення і class-означення. У цих випадках межа між оголошенням і означенням може бути доволі розмитою і, здається, для загальних концепцій «означення» і «оголошення», ці конструкції є ближчими до означень ніж до оголошень, бо в конструкціях із instance власне й означуються тіла функцій, яких типоклас, що втілюється, вимагає, і в такий спосіб означується втілення, а в конструкціях із ключовим словом class, хоча там зазвичай оголошуються функції, яких вимагає типоклас, все ж таки, в такий спосіб і означується типоклас, — себто, оголошенням функцій, яких він вимагає. До того ж, в означеннях типокласів можна надавати й означення тіл функцій, яких вимагає той типоклас, за замовчуванням, і це робить цю кон-

[*]Іноді кришталева ясність щодо розуміння цих слів поодинці і в цілому в деяких крайових випадках переходила в туманну аморфність; нас турбувала доля в українській мові слів «позна́чування», «визна́чування», «позна́чувати» і «визна́чувати» (яких тут не вистачає для повного комплекту), і непокоїло те, що в слові «ви́значити» для симетрії наголос мав би бути в іншому місці.

[†]No pun intended.

[‡] Якби ми не були знайомі із парочкою «означа́ти» і «означи́ти», то могли б подумати, що вони є недоконаним і доконаним видами дієслова якогось одного значення (за аналогією із «познача́ти» і «позна́чити» та «визнача́ти» і «ви́значити» [от тільки в «ви́значити» наголос не там, хай йому ґрець!]). Але насправді вони є недоконаним і доконаним видами дієслів різних значень.

струкцію ще близькішою до концепції «означення».

«Як» в українській мові перевантажено його синонімією до «що». Для «як» в значенні «що», речення «Ми побачили, як типокласи можуть представляти круті концепції» і «Ми побачили, що типокласи можуть представляти круті концепції» означають одне й те саме, і перекладаються як «we've seen *that* typeclasses can represent cool concepts». Проте для його значення «в який спосіб» ці два речення перекладаються англійською як «we've seen *how* typeclasses can represent cool concepts» і «we've seen *that* typeclasses can represent cool concepts», відповідно. В другому випадку, «ми побачили, що типокласи можуть представляти...» і все, а в першому — ще й дізналися, *як саме* вони «можуть представляти». В цій книзі ми намагалися використовувати «що» замість ««як» в значенні «що»», тому, якщо ви бачите «як», швидше за все йдеться про «в який спосіб».

В хаскельному коді розрізняти параметри й аргументи не так важливо (в порівнянні, наприклад, із C++, де параметр і відповідний аргумент можуть мати різні типи, і які саме перетворення аргументів в параметри відбуваються при переході крізь цей інтерфейс має неабияке значення). Але ми в цьому перекладі все одно використали точнішу термінологію. Наприклад, в означенні `f x = x * x` і виразі `f y where y = 2` маємо: `x` є параметром функції `f`, а `y` — аргументом функції `f`. `y` має значення `2` і є доступним в тілі функції `f` під ім'ям `x`. Отже, параметри «живуть» в типосиґнатурі і тілі функції, і це є те, із чим аргументи зв'язуються в точці виклику функції — і тому функції можуть брати і аргументи, і параметри — залежно від того, чи йдеться мова про те, як функцію було означено, чи про те, як саме вона викликається.

Рядок — багатозначне слово в контексті цієї книги — може означати як структуру даних (`String`), так і просто рядок — тексту чи коду. Хоча зазвичай з контексту зрозуміло, про який рядок йдеться, ми структуру даних позначили просто «рядок», а в решті випадків писали «рядок тексту» і «рядок коду».

Мені особисто довелося прочитати цю книгу в середньому шість разів (на рівні розділів: двічі англійською, і тричі українською — перший раз англійською, потім українською і англійською одночасно, потім українською повільне читання, а потім українською — швидке; і один раз усієї книги українською). «В середньому», бо деякі місця довелося перекладати `n` разів «до збіжності» :), а деякі — повертатися і переперекладати, бо з'являлися зауваження і пропозиції редакторів і читачів, чи то просто нові ідеї. Іноді редагували, доки усі вимоги не було задоволено — редакторів, читачів, перекладачів, або було доведено, що немає розв'язку і тоді лишали авторський варіант (перекладача), або не перекладали. Хоча й текст перекладено багатьма, ми спробували витримати деякі інваріанти — консистентності мови, стилю і термінології по усій книзі — тому довелося повертатися багато разів до вже відшліфованого матеріалу,

зітхати, ламати його і будувати знову. Тож ми сподіваємося, що ви прочитаєте цю книгу принаймні двічі, а деякі місця цитуватимете при нагоді в повсякденному житті! :D (жартую). Якщо серйозно — якщо ви помітили помилку, неточність, втрату змісту в перекладі, або ж маєте якісь загальні коментарі чи пропозиції щодо покращення змісту, будь ласка, листуйте за адресою semen [at] trygub.com, і ми втілимо слушні зауваження в життя. Звичайно ж, якщо вам сподобалася книга і переклад — однаково пишіть, для балансу :) та й нам цікаво буде почути, кому і в який спосіб ця книга стала в пригоді!

Автори українського перекладу «Вивчить собі Хаскела...» складають щиру подяку Дмитру Сіренку, без критичних зауваг якого обробка результатів вичитування ранніх версій була б завершена на два порядки швидше, і Тарасу Бунику, хто був першим «справжнім» читачем цього підручника, а також (в автобіографічному порядку): Олегу Бурхаю, Дмитру Рудзону, Петру Єрмоленку, Олексію Сивоконю, Матвєю Аксьонову, і Михайлу Іванкову.

Семен Тригубенко

Розділ 1

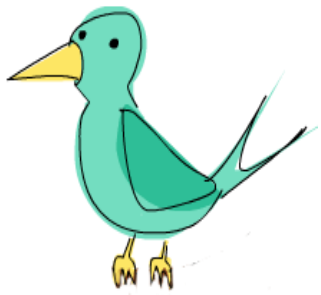
Передмова

Переклад українською Тетяни Богдан

1.1 Про цей підручник

Ласкаво просимо до **Вивчить собі Хаскела на велике щастя!** Якщо ви читаєте ці рядки тексту, то напевно хочете вивчити Хаскел... І, справді, ви прийшли туди, куди треба! Але давайте відкладемо навчання на хвильку, і спершу поговорімо про зміст цього посібника.

Я вирішив написати цей підручник аби закріпити власне володіння Хаскелом, а також аби допомогти тим, хто не знайомий з Хаскелом, подивитися на нього крізь призму мого досвіду. В Інтернеті тиняється досить багато посібників з Хаскела. Коли я починав вчитися, я користувався кількома джерелами. На кожну тему я читав декілька посібників чи статей, які пояснювали один і той же матеріал по-різному. Скориставшись кількома джерелами, я міг скласти окремі деталі в одне ціле. І раптом усе ставало зрозумілим. Отож, цей посібник — це спроба створити ще один корисний ресурс з вивчення Хаскела, щоб читачі мали змогу підібрати посібник на свій смак.



Цей посібник для тих, хто має досвід програмування імперативними мовами (C, C++, Java, Python...), але ще не стикався з функційними мовами програмування (Haskell, ML, OCaml...). Хоча я певен, що навіть без серйозного досвіду програмування така розумна людина, як мій читач, зможе розібратися і вивчити Хаскел!

Канал #haskell в мережі freenode — це чудовий ресурс для тих, хто застряг на якомусь питанні і хоче

про щось запитати. Люди там дуже добрі, терплячі і ставляться до новачків з розумінням.

Поки я нарешті збагнув, що таке Хаскел, я двічі зазнав поразки, намагаючись його вивчити, тому що все здавалось страшенно дивним, і я ніяк не міг зрозуміти, що до чого. Але одного дня все раптом «розвиднілось», а коли я подолав перші перешкоди, усе пішло, як по маслу. Я веду до того, що Хаскел — це класна мова, і якщо вас справді цікавить програмування, то цю мову варто вивчити, навіть якщо все спочатку здається дивним. Учити Хаскел — це як уперше вчитися програмувати — це прикольно! Хаскел примушує вас думати по-новому, і саме про це йдеться у наступному розділі...

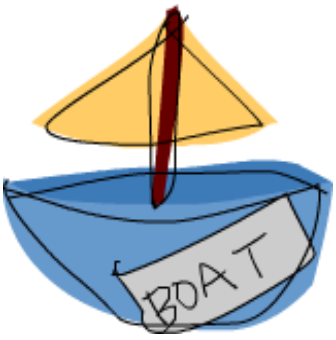
1.2 Отже, що воно таке Хаскел?

Хаскел — це **чистофункційна мова програмування** [purely functional programming language]. В імперативних мовах ви даєте комп'ютеру перелік завдань, і він їх виконує. Виконуючи завдання, він може змінювати стан. Наприклад, ви задаєте змінній `a` значення 5, виконуєте якусь операцію, і задаєте їй інше значення. Є також засоби керування потоком, — наприклад, для виконання певної операції декілька разів. У чистофункційному програмуванні комп'ютеру не кажуть, що робити, а описують, що ϵ що. Факторіал числа — це добуток усіх чисел від одиниці до того числа, сума списку чисел — це перше число плюс сума решти чисел, і так далі. Операції виражаються в формі функцій. Також не можна задавати змінній одне значення, а потім змінювати його на інше. Якщо ви сказали, що `a` дорівнює 5, то потім не можете сказати, що вона дорівнює чомусь іншому, бо ви щойно сказали, що це 5. Ви ж не брехун/брехуха, чи не так? Отже, у чистофункційних мовах програмування функція не має побічних ефектів. Все, що може функція, — це щось порахувати і повернути результат. На перший погляд здається, що функція має обмежені можливості, але насправді ефект просто чудовий: якщо двічі викликати функцію з одними й тими ж параметрами, вона гарантовано поверне один і той же результат. Ця поведінка називається **прозорістю посилань** [referential transparency], і вона не тільки дозволяє компіляторові розмірковувати про поведінку програми, а й легко відстежити (чи навіть довести), що функція правильна, а тоді будувати складніші функції, склеюючи прості функції докупи.

Хаскел — **лінивий** [lazy]. Тобто якщо йому нічого не наказати, Хаскел не виконуватиме функції та не робитиме обчислень до тих

пір, поки його не змусять показати результат. Це чудово поєднується з прозорістю посилань і дозволяє думати про програми як про низки **перетворень даних** [transformations on data]. Це також уможливлює існування нескінченних структур даних. Скажімо, у вас є **незміненний** [immutable] список чисел `xs = [1,2,3,4,5,6,7,8]` і функція `doubleMe`, яка множить кожен елемент на 2 і повертає новий список. Якщо ви помножите список на 8, імперативною мовою, виконавши `doubleMe(doubleMe(doubleMe(xs)))`, то програма пройдеться списком один раз, зробить копію і поверне її як результат. Тоді вона пройдеться тим списком ще пару разів, і поверне результат. Натомість у лінивій мові на ваше прохання виконати `doubleMe` без виводу результату програма фактично відповідає: «Ага, при нагоді порахую!». Але щойно ви захочете побачити результат, перша `doubleMe` скаже другій, що вимагає результату негайно! Друга це передає третій, і третя неохоче повертає подвоєну 1, тобто 2. Друга функція отримує її і повертає 4 першій. Перша, своєю чергою, сповіщає вам, що перший елемент списку дорівнює 8. Отже, програма проходиться списком лише один раз і лише тоді, коли це справді потрібно. Таким чином, коли ви хочете щось зробити в лінивій мові, просто візьміть початкові дані і успішно перетворюйте їх доти, поки не отримаєте потрібний результат.

Хаскел — **статично типізований** [statically typed]. Обробляючи програму, компілятор знає, який шматок коду є числом, який — рядком, і так далі. Тобто під час компіляції ви знайдете чимало можливих помилок. Якщо ви спробуєте додати число і рядок, компілятор вас насварить. Хаскел використовує дуже гарну систему типів, яка підтримує **виведення типів** [type inference]. Це означає, що не обов'язково вказувати тип кожного шматку коду, система типів достатньо розумна, щоб самій про багато що здогадатися. Якщо ви написали `a = 5 + 4`, то не треба зазначати, що `a` — це число, Хаскел сам це зрозуміє. Також, завдяки виведенню типів ваш код стає більш загальним. Якщо функція приймає два параметри і додає їх один до одного, то не треба явно вказувати їхній тип, адже функція здатна приймати два будь-яких параметри, які поводяться як числа.



Хаскел **елегантний і стислий**. Оскільки Хаскел — це високорівнева мова програмування, написані на ньому програми коротші за їхні імперативні еквіваленти. А коротші програми легше підтримувати, і в них трапляється менше

помилок.

Хаскел був розроблений **дуже розумними хлопцями** (з кандидатськими ступенями). Розробка Хаскела почалась в 1987 році, коли комітет розробників зібрався аби створити таку собі некепську мову. В 2003 було опубліковано Haskell Report, який означив стабільну версію мови.

1.3 Що треба, щоб негайно узятися до роботи

Текстовий редактор і компілятор Хаскела. У вас вже мабуть є улюблений текстовий редактор, тому не будемо на цьому зупинятися. У цьому посібнику ми використовуватимемо GHC, найпоширеніший компілятор Хаскела. Найпростіше завантажити Haskell Platform — компілятор з усім необхідним.

GHC читає файли із сирцями Хаскела (вони здебільшого мають розширення .hs) і компілює їх, а також має інтерактивний режим, в якому можна інтерактивно взаємодіяти з програмою. Інтерактивно! Можна викликати функції з підвантажених програм і одразу ж бачити результат обрахунку. Для навчання це набагато швидше і зручніше, аніж компілювати і запускати програму після кожної зміни. Інтерактивний режим викликається командою **ghci**. Якщо означити якісь функції у файлі **myfunctions.hs**, ці функції можна завантажити командою **:l myfunctions** і тоді бавитися з ними, за умови, що файл **myfunctions.hs** знаходиться в тій самій директорії, з якої був запущений **ghci**. Якщо .hs змінився, його можна перезавантажити командою **:l myfunctions** або **:r**, яка перевантажує поточну програму. Як правило, я так і працюю: означую певні функції в якомусь .hs файлі, завантажую файл, бавлюся з функціями, а потім змінюю .hs файл і так далі. Надалі ми робитимемо так само.

Розділ 2

Перші кроки

Переклад українською Ганни Лелів

2.1 На старт, увага, руш!

Гаразд, до праці! Якщо ви з тих жахливих людей, які ніколи не читають інструкцій, і ви й тут їх пропустили, я б усе-таки радив вам прочитати останню частину вступу, тому що вона пояснює, як працювати з цим посібником і як завантажувати функції. Спершу ми зайдемо в інтерактивний режим ghc і викличемо якусь функцію, щоб спробувати Хаскела на смак. Запустіть термінал і наберіть `ghci`. На екрані з'явиться щось отаке:

```
GHCi, version 6.8.2: http://www.haskell.org/ghc/  :? for help
Loading package base ... linking ... done.
Prelude>
```

Вітаємо, ви у GHCi! Запрошення до вводу тут — це `Prelude>`, але воно стане довшим, якщо в сесію почати щось завантажувати, тому ми використовуватимемо `ghci>`. Якщо ви хочете мати таку ж підказку, просто наберіть `:set prompt "ghci>"`.

Ось кілька простих обчислень.

```
ghci> 2 + 15
17
ghci> 49 * 100
4900
```

```
ghci> 1892 - 1472
420
ghci> 5 / 2
2.5
ghci>
```

Як на мене, все зрозуміло. Ми також можемо використати кілька операторів в одному рядку тексту, дотримуючись звичних правил пріоритету. Можемо поставити дужки, щоб позначити або змінити пріоритет.

```
ghci> (50 * 100) - 4999
1
ghci> 50 * 100 - 4999
1
ghci> 50 * (100 - 4999)
-244950
```

Круто, чи не так? Я знаю, що ні, але хвильку зачекайте. Тут на нас чекає невеличка пастка — від'ємні числа. Якщо вам потрібне від'ємне число, його варто взяти в дужки. Якщо ви напишете `5 * -3`, GHCi накричить на вас, а от із `5 * (-3)` не буде жодних проблем.

Булева алгебра теж доволі зрозуміла. Ви мабуть знаєте, що `&&` означає булеве *і*, а `||` означає булеве *або*. `not` заперечує `True` або `False`.

```
ghci> True && False
False
ghci> True && True
True
ghci> False || True
True
ghci> not False
True
ghci> not (True && True)
False
```

Перевірка на рівність робиться ось так:

```
ghci> 5 == 5
True
ghci> 1 == 0
False
ghci> 5 /= 5
False
ghci> 5 /= 4
True
```

```
ghci> "hello" == "hello"
True
```

А як щодо `5 + "llama"` чи `5 == True`? Ну, перший шматок коду видасть страшне повідомлення про помилку!

```
No instance for (Num [Char])
arising from a use of `+' at <interactive>:1:0-9
Possible fix: add an instance declaration for (Num [Char])
In the expression: 5 + "llama"
In the definition of `it': it = 5 + "llama"
```

Ой! GHCi каже, що `"llama"` — це не число, і тому він не знає, як додати його до 5. Навіть якби це був не `"llama"`, а `"four"` чи `"4"`, Хаскел не вважав би це числом. `+` очікує, що справа та зліва будуть числа. Якщо ми спробуємо виконати `True == 5`, GHCi скаже, що типи не співпадають. `+` працює тільки з числами, тоді як `==` працює з будь-якими двома об'єктами, які можна порівняти. Але штука в тім, що вони мусять належати до одного типу. Не можна порівняти яблука та апельсини. Дещо пізніше ми розглянемо типи детальніше. Зверніть увагу: `5 + 4.0` можна виконати, тому що `5` хитре і може поводитись як ціле число або **число з плаваючою комою** [floating-point number]. `4.0` не може поводитись як ціле число, тому `5` мусить підлаштуватися під нього.

Ви помітили, що ми увесь час використовували функції? До прикладу, `*` — це функція, що бере два числа і множить їх. Ми викликаємо цю функцію, вставивши її між двома числами. Це так звана *інфіксна* функція. Більшість функцій, що не використовуються із числами — це *префіксні* функції. Зараз ми їх розглянемо.

Функції зазвичай префіксні, тому надалі ми не будемо увесь час писати, що функція має префіксну форму, а припустимо, що так є. В імперативних мовах функцію викликають, написавши ім'я функції, а тоді її параметри — в дужках і через кому. В Хаскелі функцію викликають так: пишуть ім'я функції, ставлять пробіл, і далі пишуть параметри, відокремлені пробілами. Спробуймо викликати одну із найнудніших функцій у Хаскелі.

```
ghci> succ 8
9
```

Функція `succ` бере все, для чого означено наступний елемент, і повертає той наступний елемент. Як бачите, ми всього лиш від-

окремлюємо ім'я функції від параметру пробілом. Викликати функцію із кількома параметрами так само легко. Функції `min` і `max` беруть два об'єкти, які можна впорядкувати (як числа!). `min` повертає менший за значенням об'єкт, а `max` — більший. Переконайтеся самі:

```
ghci> min 9 10
9
ghci> min 3.4 3.2
3.2
ghci> max 100 101
101
```

Застосування функції (тобто, виклик функції вставлянням пробілу після неї, а тоді поданням параметрів) має найвищий пріоритет. Це означає, що оці дві інструкції — еквівалентні:

```
ghci> succ 9 + max 5 4 + 1
16
ghci> (succ 9) + (max 5 4) + 1
16
```

Але якщо ми хочемо отримати наступний елемент добутку чисел 9 і 10, ми не можемо написати `succ 9 * 10`, тому що то буде наступний елемент після 9, якого буде помножено на 10. Тобто 100. Щоб отримати 91, треба написати `succ (9 * 10)`.

Якщо функція бере два параметри, її можна викликати як інфіксну функцію, оточивши її ім'я спадними наголосами. Наприклад, функція `div` бере два цілі числа та робить цілочисельне ділення. `div 92 10` видасть 9. Але якщо ми викличемо функцію у такий спосіб, то іноді може виникнути питання щодо яке число ділять на яке. Тому, для покращення розуміння, ми також можемо викликати її як інфіксну функцію ось так: `92 `div` 10`.

Чимало людей, які раніше програмували імперативними мовами, вважають, що дужки повинні позначати застосування функції. До прикладу, в C, дужки використовують, щоб викликати такі функції, як `foo()`, `bar(1)` чи `baz(3, "haha")`. Як я вже казав, у Хаскелі застосування функції позначається пробілом. Ті ж самі функції виглядатимуть у Хаскелі ось так: `foo`, `bar 1` і `baz 3 "haha"`. Отож, якщо перед вами `bar (bar 3)`, це не значить, що `bar` викликають із параметрами `bar` і `3`. Це означає, що спочатку ми викликаємо функцію `bar` з параметром `3`, щоб отримати якийсь результат, а тоді знову викликаємо `bar` подаючи цей результат як параметр. У C, це виглядало б ось так: `bar(bar(3))`.

2.2 Перші функції малюка

У попередньому розділі ми спробували викликáти функції. А тепер створімо свою власну функцію! Запустіть свій улюблений текстовий редактор і наберіть там ось цю функцію, яка бере число і множить його на два.

```
doubleMe x = x + x
```

Функції означують подібно до того, як їх викликають. Після імені функції ідуть параметри, відокремлені пробілами. Але коли ми означуємо функцію, то ставимо `=`, а після цього означуємо, що ця функція робить. Збережіть це як `baby.hs`, і запустіть `ghci` з директорії в якій ви зберегли цей файл. Тепер, вже в GHCi, запустіть `:l baby`. Коли завантажився код, побавимося із функцією, яку ми означили.

```
ghci> :l baby
[1 of 1] Compiling Main             ( baby.hs, interpreted )
Ok, modules loaded: Main.
ghci> doubleMe 9
18
ghci> doubleMe 8.3
16.6
```

`+` працює і з цілими числами, і з числами з плаваючою комою (по суті з усім, що може вважатися числом), тому наша функція теж працює з будь-яким числом. Напишімо функцію, яка бере два числа, множить кожне на два, а тоді додає їх.

```
doubleUs x y = x*2 + y*2
```

Просто. Ми також могли означити її як `doubleUs x y = x + x + y + y`. Перевірка дає цілком очікувані результати (не забудьте долучити цю функцію до файлу `baby.hs`, зберегти, а тоді виконати `:l baby` в GHCi).

```
ghci> doubleUs 4 9
26
ghci> doubleUs 2.3 34.2
73.0
ghci> doubleUs 28 88 + doubleMe 123
478
```

Ви можете викликати свої власні функції з інших функцій, які ви написали. Пам'ятаючи про це, переозначмо `doubleUs` ось так:

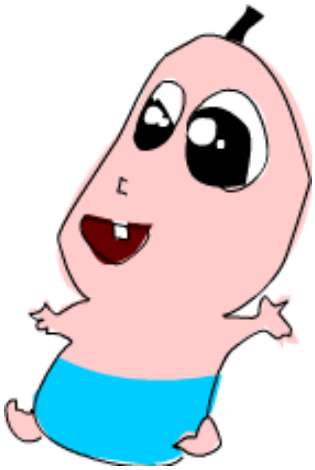
```
doubleUs x y = doubleMe x + doubleMe y
```

Це простенький приклад ідіоми, із якою ви часто зустрічатиметься у Хаскелі. Написати прості, правильні функції, а тоді поєднати їх у складніші. Таким чином ми уникаємо повторів. А раптом якісь математики доведуть, що 2 — це насправді 3, і вам доведеться переписувати цілу програму? Тоді ви просто переозначите `doubleMe` на `x + x + x`, і оскільки `doubleUs` викликає `doubleMe`, ця функція автоматично працюватиме у дивному, новому світі, де 2 дорівнює 3.

У Хаскелі функції не мусять бути подані у певному порядку, тому немає значення, чи ви спершу означите `doubleMe`, а пізніше `doubleUs`, чи навпаки.

А тепер ми напишемо функцію, яка множить число на 2 тільки якщо це число менше або дорівнює 100, адже числа, більші за 100, вже й так великі!

```
doubleSmallNumber x = if x > 100
                      then x
                      else x*2
```



Тут ми ввели інструкцію розгалуження у Хаскелі. Мабуть ви знайомі із інструкцією розгалуження із інших мов. Різниця між інструкцією розгалуження у Хаскелі та імперативних мовах полягає в тому, що в Хаскелі частина «інакше» — обов'язкова. В імперативних мовах якщо умову не задовільнено, то кілька кроків можна пропустити, тоді як у Хаскелі кожен вираз і функція мають щось повернути. Інструкцію розгалуження можна було написати і в один рядок тексту, але мені так її легше читати. До того ж, у Хаскелі інструкція розгалуження — це також *вираз*. Вираз — це по суті шматок коду, що повертає значення. `5` — вираз, тому що він повертає 5, `4 + 8` — вираз, `x + y` — вираз, оскільки він повертає суму `x` і `y`. Оскільки «інакше» — обов'язкове, інструкція розгалуження завжди щось поверне, тому вона і є виразом. Якби ми захотіли додати одиницю до кожного числа, яке нам дала попередня функція, ми б написали тіло функції ось так:

```
doubleSmallNumber' x = (if x > 100 then x else x*2) + 1
```

Якби ми не поставили дужок, функція додала б одиницю тільки якщо `x` не був більшим за 100. Зверніть увагу на `'` в кінці імені функції. Апостроф не має особливого значення у синтаксисі Хаскела. Цей символ можна використовувати в імені функції. `'` зазвичай використовують, щоб позначити завзяту версію функції (тобто, версію яка не є лінивою) або дещо змінену версію функції чи змінної. Оскільки символ `'` можна використовувати у назвах функцій,

можемо написати ось таку функцію:

```
conanO'Brien = "It's a-me, Conan O'Brien!"
```

Зверніть увагу на дві речі. По-перше, в імені функції ми не писали ім'я Конана з великої літери. Це тому що функції не можуть починатися із великих літер. Пізніше я поясню чому. По-друге, ця функція не бере жодних параметрів. Якщо функція не приймає жодних параметрів, ми зазвичай кажемо, що це є *означення імені* (або просто — *ім'я*). Ми не можемо змінити значення імені (і функції) після того, як ми їх означили, тому `conanO'Brien` і рядок `"It's a-me, Conan O'Brien!"` можна вживати взаємозамінно.

2.3 Вступ до списків

Списки в Хаскелі такі ж корисні, як і списки покупок у реальному житті. Це найпоширеніша структура даних, і її можна пристосувати багатьма різними способами до моделювання і розв'язання цілої низки задач. Списки ТАКІ класні! У цьому розділі ми неглибоко заглибимося в списки, рядки (які є списками) і спискові характери.

У Хаскелі списки — це **однорідна** структура даних. Вона зберігає кілька елементів одного типу. Тобто ми можемо мати список цілих чисел або список символів, але не можемо мати списку з кількома цілими числами і кількома символами. А тепер — список!

Примітка: Нам треба використовувати ключове слово `let` для означування імен безпосередньо в GHCi. Виконати `let a = 1` в GHCi — це те ж саме, що написати `a = 1` в скрипті, а тоді його завантажити.

```
ghci> let lostNumbers = [4,8,15,16,23,42]
ghci> lostNumbers
[4,8,15,16,23,42]
```

Як бачите, списки позначаються квадратними дужками, а значення у списках відокремлюються комами. Якби ми створили ось такий список `[1,2,'a',3,'b','c',4]`, Хаскел поскаржився б, що символи (які, до речі, в Хаскелі беруться в ординарні лапки) не є числами. Що ж до символів, то рядки — це всього лиш списки символів. `"hello"` — це синтаксичний цукор для

`['h','e','l','l','o']`. Оскільки рядки є списками, ми можемо застосовувати до них функції означені для списків, що дуже зручно.

Часто потрібно скласти два списки докупи. Це можна зробити за допомогою оператора `++`.

```
ghci> [1,2,3,4] ++ [9,10,11,12]
[1,2,3,4,9,10,11,12]
ghci> "hello" ++ " " ++ "world"
"hello world"
ghci> ['w','o'] ++ ['o','t']
"woot"
```

Будьте уважні, якщо ви часто застосовуєте оператор `++` до довгих рядків. Коли ви поєднуєте два списки (навіть якщо ви додаєте до списку одноелементний список [односписок], як-от `[1,2,3] ++ [4]`), за лаштунками, Хаскел мусить пройтися усім списком ліворуч від `++`. Якщо список короткий — нема проблем. Але якщо ви додаватимете до списку довжиною в п'ятдесят мільйонів символів, то змарнуєте купу часу. Водночас, приєднання до списку (схоже на додавання до списку, але не в кінці, а на початку) можна зробити якщо скористатись оператором `:` (він також відомий під назвою оператор «cons»). Приєднання до списку за допомогою `:` — миттєва операція.

```
ghci> 'A':" SMALL CAT"
"A SMALL CAT"
ghci> 5:[1,2,3,4,5]
[5,1,2,3,4,5]
```

Зверніть увагу, що `:` бере число та список чисел або символ і список символів, тоді як `++` бере два списки. Навіть якщо ви додаєте елемент до списку за допомогою `++`, його треба взяти в квадратні дужки, щоб перетворити його на список.

`[1,2,3]` — це по суті синтаксичний цукор для `1:2:3:[]`. `[]` — це порожній список. Якщо ми приєднаємо до нього `3`, то отримаємо `[3]`. Якщо до цього приєднаємо `2`, то матимемо `[2,3]`, і так далі.

Примітка: `[]`, `[[]]` і `[[],[],[]]` — це різні речі. Перше — це порожній список, друге — список, що містить один порожній список, а третє — це список, що містить три порожні списки.

Щоб отримати елемент зі списку за його індексом, використовуйте `!!`. Індексація починається з 0.

```
ghci> ”Steve Buscemi” !! 6
’B’
ghci> [9.4,33.2,96.2,11.2,23.25] !! 1
33.2
```

Але якщо ви спробуєте отримати шостий елемент зі списку, що містить тільки чотири елементи, то отримаєте помилку. То ж будьте уважні!

Списки також можуть містити списки. Вони можуть містити списки, що містять списки, що містять списки...

```
ghci> let b = [[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3]]
ghci> b
[[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3]]
ghci> b ++ [[1,1,1,1]]
[[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3],[1,1,1,1]]
ghci> [6,6,6]:b
[[6,6,6],[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3]]
ghci> b !! 2
[1,2,2,3,4]
```

Підсписки (списки усередині списку) можуть мати різну довжину, але не різний тип. Так само як не можна мати список із кількома символами та кількома числами, не можна мати список, що має кілька списків символів і кілька списків чисел.

Списки можна порівняти, якщо можна порівняти їхнє наповнення. Якщо порівнювати списки за допомогою `<`, `<=`, `>` і `>=`, то зміст цих списків буде порівнюватись лексикографічно. Спочатку порівнюють голови. Якщо вони однакові — тоді порівнюються другі елементи, і так далі.

```
ghci> [3,2,1] > [2,1,0]
True
ghci> [3,2,1] > [2,10,100]
True
ghci> [3,4,2] > [3,4]
True
ghci> [3,4,2] > [2,4]
True
ghci> [3,4,2] == [3,4,2]
True
```

Що ще можна робити зі списками? Ось кілька основних функцій, що працюють зі списками.

`head` бере список і повертає його голову. Голова списку — це його перший елемент.

```
ghci> head [5,4,3,2,1]
5
```

`tail` бере список і повертає його хвіст. Іншими словами, вона відрубує спискові голову.

```
ghci> tail [5,4,3,2,1]
[4,3,2,1]
```

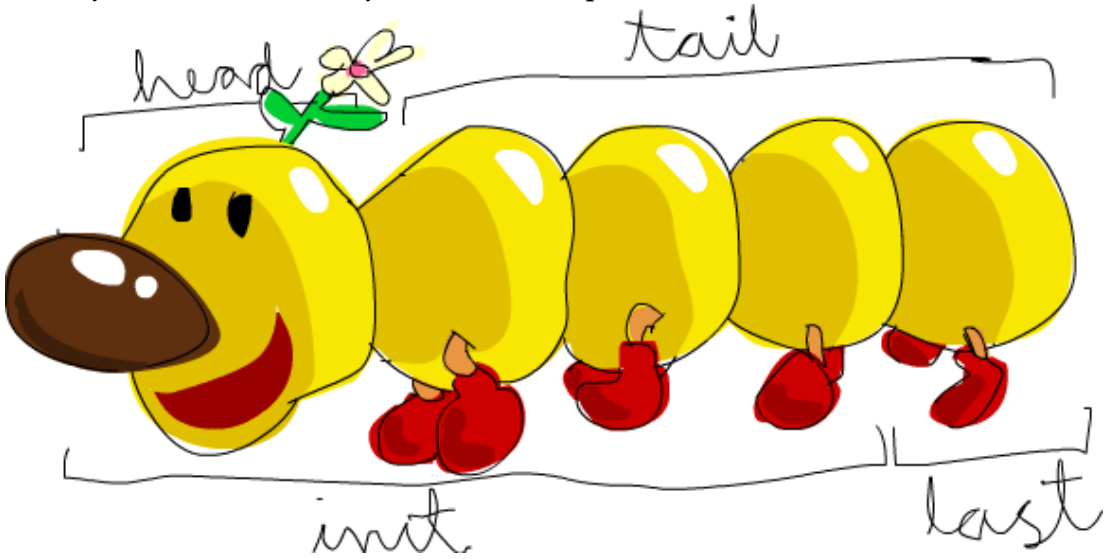
`last` бере список і повертає його останній елемент.

```
ghci> last [5,4,3,2,1]
1
```

`init` бере список і повертає усе, крім його останнього елемента.

```
ghci> init [5,4,3,2,1]
[5,4,3,2]
```

Якщо уявити собі список у вигляді монстра, ось як він виглядатиме:



Але що трапиться, якщо ми спробуємо отримати голову порожнього списку?

```
ghci> head []
*** Exception: Prelude.head: empty list
```

О Боже! Все пропало! Якщо немає монстра, то й немає голови. Будьте обережні з `head`, `tail`, `last` і `init` та не використовуйте їх із порожніми списками. Цю помилку неможливо вловити під час компіляції, тому то є старим добрим правилом програміста писати код обережно, щоб прохання до Хаскелу дати вам кілька елементів із порожнього списку просто не виникали.

Очевидно, що `length` бере список і повертає його довжину.

```
ghci> length [5,4,3,2,1]
5
```

`null` перевіряє, чи список порожній. Якщо так, то вона повертає `True`, якщо ж ні — то `False`. Використовуйте `null xs` замість `xs == []` (де `xs` — ім'я вашого списку).

```
ghci> null [1,2,3]
False
ghci> null []
True
```

`reverse` розвертає список.

```
ghci> reverse [5,4,3,2,1]
[1,2,3,4,5]
```

`take` бере число n та список. Вона витягає n елементів із початку списку. Дивіться:

```
ghci> take 3 [5,4,3,2,1]
[5,4,3]
ghci> take 1 [3,9,3]
[3]
ghci> take 5 [1,2]
[1,2]
ghci> take 0 [6,6,6]
[]
```

Бачите — якщо ми хочемо взяти більше елементів, ніж є у списку, вона просто повертає весь список. Спробуймо взяти 0 елементів — і отримаємо порожній список.

`drop` працює подібним чином, відкидаючи перші n елементів списку.

```
ghci> drop 3 [8,4,2,1,5,6]
[1,5,6]
ghci> drop 0 [1,2,3,4]
[1,2,3,4]
ghci> drop 100 [1,2,3,4]
[]
```

`maximum` бере список речей, які можна розмістити в певному порядку, і повертає найбільшу з них.

`minimum` повертає найменшу.

```
ghci> minimum [8,4,2,1,5,6]
1
ghci> maximum [1,9,2,3,4]
9
```

`sum` бере список чисел і повертає їхню суму.

`product` бере список чисел і повертає їхній добуток.

```
ghci> sum [5,2,1,6,3,2,5,7]
31
ghci> product [6,2,1,2]
24
ghci> product [1,2,5,6,7,9,2,0]
0
```

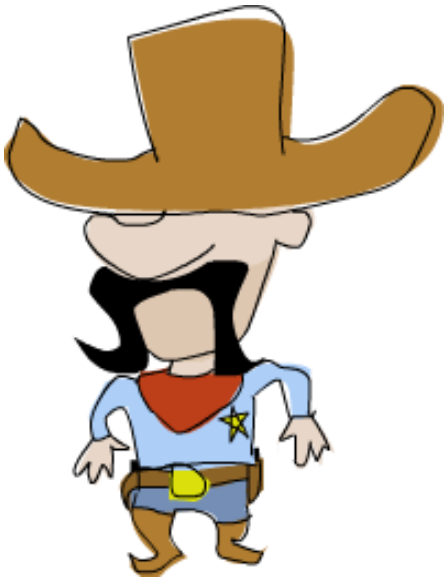
`elem` бере щось і список речей і каже, чи це щось є елементом списку. Зазвичай її викликають як інфіксну функцію, бо так код легше читається.

```
ghci> 4 `elem` [3,4,5,6]
True
ghci> 10 `elem` [3,4,5,6]
False
```

Ми розглянули кілька основних функцій, які працюють зі списками. Пізніше, в підрозділі 7.2, ми зустрінемося іще із декількома.

2.4 Техаські «рейнджі» або ж діапазони

А якщо ми захочемо мати список усіх чисел від 1 до 20? Звичайно, можна взяти й набрати всі ці числа вручну, але ж це заняття не для джентльменів, які вимагають, щоб їхні мови програмування були досконалими. Натомість ми скористаємося діапазонами. Діапазони в Хаскелі — це спосіб створення списку з елементів, які можна перелічити і для яких означено впорядкування. Наприклад — арифметичні прогресії. Числа можна перелічити і для них означено порядок. Один, два, три, чотири і так далі. Символи є теж впорядкованими. Абетка — це перелік символів від А до Я. Імена можна впорядкувати, але не можна перелічити. Що йде після «Івана»? Не знаю.



Щоб створити список, який містить усі натуральні числа від 1 до 20, просто напишіть `[1..20]`. Це еквівалентно `[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]`; тобто, між цими двома варіантами немає жодної різниці. Хоча ні: перелічувати купу чисел вручну — це тупо.

```
ghci> [1..20]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
ghci> ['a'..'z']
"abcdefghijklmnopqrstuvwxyz"
ghci> ['K'..'Z']
"KLMNOPQRSTUVWXYZ"
```

Діапазон — класна штука, бо для неї можна вказати крок. А раптом ви хочете всі парні числа у проміжку від 1 до 20? Або кожне третє число між 1 і 20?

```
ghci> [2,4..20]
[2,4,6,8,10,12,14,16,18,20]
ghci> [3,6..20]
[3,6,9,12,15,18]
```

Треба всього лиш відокремити перші два елементи комою, а тоді вказати верхню межу. Хоча діапазони з кроками доволі розумні, декому вони можуть видатися розумнішими ніж вони є насправді. Не можна написати `[1,2,4,8,16..100]` і сподіватися отримати всі степені двійки. Тому що: по-перше, можна вказати тільки один крок, а по-друге, — означення для неарифметичних послідовностей треба писати обережно, адже, зазвичай, означення, що містять лише декілька перших членів таких послідовностей, є неоднозначними.

Щоб створити список із усіма числами від 20 до 1, не можна написати `[20..1]`. Треба написати `[20,19..1]`.

Будьте уважні, коли використовуєте числа з плаваючою комою у діапазонах! Вони за означенням не зовсім точні, тому, якщо використовувати **плавомки** [floats] у діапазонах, можна отримати доволі цікаві результати.

```
ghci> [0.1, 0.3 .. 1]
[0.1,0.3,0.5,0.7,0.8999999999999999,1.0999999999999999]
```

Я б радив не використовувати їх у діапазонах списків.

За допомогою діапазонів можна створювати нескінченні списки. Треба всього лиш не встановлювати верхньої межі. Про нескінченні списки ми детальніше поговоримо згодом. А зараз погляньмо, як нам отримати перші 24 числа кратні 13. Звісно, ми можемо написати `[13,26..24*13]`. Але існує кращий спосіб: `take 24 [13,26..]`. Хаскел лінивий, тому він не буде пробувати

вирахувати увесь нескінченний список відразу, бо він ніколи не закінчиться. Хаскел чекатиме доки ви захочете щось дістати з того нескінченного списку, і, коли побачить, що ви хочете перші 24 елементи, охоче робить вам ласку.

Ось кілька функцій, які створюють нескінченні списки:

`cycle` бере список і зациклює його у нескінченний список. Якщо ви спробуєте відобразити результат, то це триватиме безконечно, тому десь його треба буде відрізати.

```
ghci> take 10 (cycle [1,2,3])
[1,2,3,1,2,3,1,2,3,1]
ghci> take 12 (cycle "LOL ")
"LOL LOL LOL "
```

`repeat` бере елемент і створює нескінченний список лише з цього елемента. Схоже на зациклення списку із одним-єдиним елементом.

```
ghci> take 10 (repeat 5)
[5,5,5,5,5,5,5,5,5,5]
```

Хоча, якщо вам потрібна певна кількість того ж самого елемента у списку, простіше скористатися функцією `replicate`. `replicate 3 10` повертає `[10,10,10]`.

2.5 Мене звати списковий характер

Якщо ви коли-небудь слухали курс математики, то мабуть стикалися зі *множинними характерами*. Їх зазвичай використовують для побудови більш специфічних множин з множин більш загальних. Простенький характер для множини, що містить перші десять парних натуральних чисел — це $S = \{2 \cdot x | x \in \mathbb{N}, x \leq 10\}$. Частина перед трубою (вертикальною рискою) називається функцією виводу. `x` — це змінна, `N` — вхідна множина, а `x <= 10` — предикат. Тобто, ця множина містить усі натуральні числа, що задовольняють умові, помножені на 2.

Якби ми хотіли написати це на Хаскел, ми б написали щось на зразок `take 10 [2,4..]`. А якщо ми хочемо не подвоювати перші 10 натуральних чисел, а застосовувати до них якусь більш складну функцію? Для цього існує списковий характер. Спискові характери дуже подібні до множинних характерів. Почнемо тренування на задачі отримання перших 10 парних чисел. Скористаймося, наприклад, списковим характером `[x*2 | x <- [1..10]]`. Ми витя-

гуємо `x` з `[1..10]`, подвоюємо і повертаємо. Для кожного елемента в `[1..10]` (який ми прив'язали до `x`), ми отримуємо цей елемент, тільки подвоєний. Ось цей списковий характер у дії.

```
ghci> [x*2 | x <- [1..10]]
[2,4,6,8,10,12,14,16,18,20]
```

Як бачите, ми отримали бажаний результат. Тепер додаймо до того характеру умову (або ж предикат). Предикати йдуть після зв'язок і відокремлюються від них комами. Скажімо, ми хочемо тільки ті елементи, які — якщо їх подвоїти — більші або дорівнюють 12.

```
ghci> [x*2 | x <- [1..10], x*2 >= 12]
[12,14,16,18,20]
```

Круто, все працює. А якщо ми хочемо всі числа від 50 до 100, остача яких, якщо їх поділити на 7, дорівнюватиме 3? Легко.

```
ghci> [ x | x <- [50..100], x `mod` 7 == 3]
[52,59,66,73,80,87,94]
```

Є! Зверніть увагу, що «прополювання» списків предикатами називається **фільтруванням**. Отож, ми щойно взяли список чисел і профільтрували їх предикатом. А тепер ще один приклад. Скажімо, нам потрібен характер, який заміняє кожне непарне число більше за 10 на `"BANG!"`, а кожне непарне число менше за 10 на `"BOOM!"`. Якщо число парне, ми викидаємо його з нашого списку. Для зручності помістимо цей характер всередину функції, щоб ми могли використати його повторно без зайвих клопотів.

```
boomBangs xs = [ if x < 10 then "BOOM!" else "BANG!" | x <- xs, odd x]
```

Остання частина характеру — це предикат. Функція `odd` повертає `True` для непарного числа і `False` для парного. Елемент входить до списку тільки якщо усі предикати після обчислення приймають значення `True`.

```
ghci> boomBangs [7..13]
["BOOM!","BOOM!","BANG!","BANG!"]
```

Можемо включити кілька предикатів. Якщо нам потрібні усі числа від 10 до 20, які не дорівнюють 13, 15 чи 19, ми напишемо:

```
ghci> [ x | x <- [10..20], x /= 13, x /= 15, x /= 19]
[10,11,12,14,16,17,18,20]
```

Ми не лише можемо мати кілька предикатів у списках характерах (щоб елемент потрапив до кінцевого списку, він мусить задовольнити усі предикати), а й брати елементи із кількох списків. У такому випадку характери видають усі комбінації із списків на вході, а тоді поєднують їх за допомогою заданої

функції виводу. Якщо характер отримує елементи із двох списків завдовжки 4 елементи кожний, то він створить список завдовжки 16 елементів (за умови, що немає фільтрації). Якщо ми маємо два списки, `[2,5,10]` і `[8,10,11]`, і хочемо отримати добутки усіх можливих комбінацій чисел у тих списках, ось що ми зробимо.

```
ghci> [ x*y | x <- [2,5,10], y <- [8,10,11]]
[16,20,22,40,50,55,80,100,110]
```

Як ми й очікували, довжина нового списку дорівнює 9. А якщо нам потрібні усі можливі добутки більше 50?

```
ghci> [ x*y | x <- [2,5,10], y <- [8,10,11], x*y > 50]
[55,80,100,110]
```

А як щодо спискового характеру, що поєднує список прикметників і список іменників… заради сміху.

```
ghci> let nouns = ["hobo","frog","pope"]
ghci> let adjectives = ["lazy","grouchy","scheming"]
ghci> [adjective ++ " " ++ noun | adjective <- adjectives, noun <- nouns]
["lazy hobo","lazy frog","lazy pope","grouchy hobo","grouchy frog",
"grouchy pope","scheming hobo","scheming frog","scheming pope"]
```

Я знаю! Напишімо свою власну версію `length`! Ми назвемо її `length'`.

```
length' xs = sum [1 | _ <- xs]
```

`_` означає, що нам байдуже, що ми візьмемо зі списку. Тому замість того, щоб написати ім'я змінної, яке ми ніколи не будемо використовувати, ми просто пишемо `_`. Ця функція замінює кожен елемент списку на `1`, а тоді додає. Тобто остаточна сума — це і буде довжина нашого списку.

Дружнє нагадування: оскільки рядки є списками, ми можемо використовувати спискові характери для обробки і побудови рядків. Ось ця функція, наприклад, бере рядок та вилучає з нього все, крім великих літер.

```
removeNonUppercase st = [ c | c <- st, c `elem` ['A'..'Z']]
```

Перевіряємо:

```
ghci> removeNonUppercase "Hahaha! Ahahaha!"
"HA"
ghci> removeNonUppercase "IdontLIKEFROGS"
"ILIKEFROGS"
```

Усю роботу тут робить предикат. Він каже, що символ увійде до нового списку тільки якщо він є елементом списку `['A'..'Z']`. Вкладені спискові хара-

ктери також можливі, якщо ви працюєте зі списками, що містять списки. Наприклад, хай наш список містить кілька списків чисел. Вилучімо всі непарні числа, не розморщивши список при цьому.

```
ghci> let xxs = [[1,3,5,2,3,1,2,4,5]
                ,[1,2,3,4,5,6,7,8,9]
                ,[1,2,4,2,1,6,3,1,3,2,3,6]]
ghci> [ [ x | x <- xs, even x ] | xs <- xxs]
[[2,2,4],[2,4,6,8],[2,4,2,6,2,6]]
```

Спискові характери можна записувати в кілька рядків тексту. Отож якщо ви не в GHCi, то краще розділити довгі спискові характери на кілька рядків тексту, особливо якщо вони містять інші спискові характери усередині.

2.6 Кортежі

До певної міри, кортежі схожі на списки — вони зберігають кілька значень як одне значення. Проте між ними є кілька істотних відмінностей. Список чисел — це список чисел, і це і є його тип. Не має значення, чи список містить тільки одне число чи нескінченну кількість чисел. Водночас, кортежі використовують тоді, коли точно знають, скільки значень треба поєднати, а тип кортежу залежить від того, скільки в нього компонентів, і якого ці компоненти типу. Кортежі позначаються круглими дужками, а їхні компоненти відокремлюються комами.

Ще одна важлива відмінність: кортежі не мусять бути однорідними. На відміну від списку, кортеж може містити комбінацію кількох (різних) типів.

Подумаймо, як відобразити двовимірний вектор у Хаскелі. Можемо використати список. Ніби те що треба. А якщо ми захочемо долучити кілька векторів до списку, щоб відобразити точки фігури на двовимірній площині? Можна написати щось на зразок `[[1,2],[8,11],[4,5]]`. Але проблема в тому, що ми також можемо написати `[[1,2],[8,11,5],[4,5]]`. Хаскел це проковтне, адже це все іще є списком списків із числами. З іншого боку ми знаємо, що це — нісенітниця в цьому контексті. Водночас, кортеж довжиною 2 (також відомий під назвою «пара») — це вже окремий тип, а це значить, що список не може містити у собі кілька пар і триплет (кортеж довжини 3). Тож краще використовуймо пари. Не будемо брати вектори в квадратні дужки, а використаємо круглі: `[(1,2),(8,11),(4,5)]`. А якщо спробувати задати отаку фігуру: `[(1,2),(8,11,5),(4,5)]`? Ну, тоді ми отримаємо отаку помилку:

```
Couldn't match expected type `(t, t1)'
against inferred type `(t2, t3, t4)'
In the expression: (8, 11, 5)
In the expression: [(1, 2), (8, 11, 5), (4, 5)]
In the definition of `it': it = [(1, 2), (8, 11, 5), (4, 5)]
```

Нам кажуть, що ми спробували використати пару і триплет в одному списку, а так не можна. Списку `[(1,2), ("One",2)]` створити також не можна, тому що перший елемент цього списку — це пара чисел, а другий елемент — пара, що складається зі рядка й числа. Кортежі використовуються для кодування широкого спектру даних. До прикладу, якщо ми хочемо відобразити чиїсь ім'я і вік на Хаскел, то можемо використати триплет: `("Christopher", "Walken", 55)`. Як бачимо з цього прикладу, триплети теж можуть містити списки.

Використовуйте кортежі, коли заздалегідь відомо, скільки компонентів матиме шматок даних. Кортежі доволі негнучкі, оскільки кожен окремий розмір кортежу — це окремий тип, тому не можна написати загальну функцію, яка додаватиме до кортежу елемент — вам доведеться написати одну функцію, яка додаватиме елемент до пари, ще одну функцію, яка додаватиме до триплетів, ще одну, яка додаватиме до квадруплетів, і так далі.

Одноелементні списки існують, а от одноелементний кортеж — ні. Та й це безглуздо, якщо добре подумати. Одноелементний кортеж — це просто елемент в обгортці, яка нічого цікавого не додає. Іншими словами, немає ніяких переваг використання одноелементного кортежу — краще використовувати власне елемент, безпосередньо.

Як і списки, кортежі можна порівнювати між собою, якщо їхні компоненти можна порівнювати. Але не можна порівняти два кортежі різного розміру, тоді як порівняти два списки різного розміру — цілком можливо. Ось дві корисні функції, що працюють із парами:

`fst` бере пару та повертає її перший компонент.

```
ghci> fst (8,11)
8
ghci> fst ("Wow", False)
"Wow"
```

`snd` бере пару та повертає її другий компонент. Ніколи б не здогадався!

```
ghci> snd (8,11)
11
ghci> snd ("Wow", False)
False
```

Примітка: Ці функції працюють тільки з парами. Вони не працюватимуть із триплетами, квадруплетами, квінтуплетами і так далі. Згодом я розповім про те, як витягувати дані з кортежів різними способами.

Класна функція, яка повертає список пар: `zip`. Вона бере два списки та «застібає» їх докупи в один список, поєднуючи відповідні елементи в пари. Простенька функція, але використовується дуже часто. Вона стане у пригоді, коли вам потрібно буде певним чином поєднати два списки або одночасно обробити два списки. Ось як це виглядає.

```
ghci> zip [1,2,3,4,5] [5,5,5,5,5]
[(1,5),(2,5),(3,5),(4,5),(5,5)]
ghci> zip [1 .. 5] ["one", "two", "three", "four", "five"]
[(1,"one"),(2,"two"),(3,"three"),(4,"four"),(5,"five")]
```

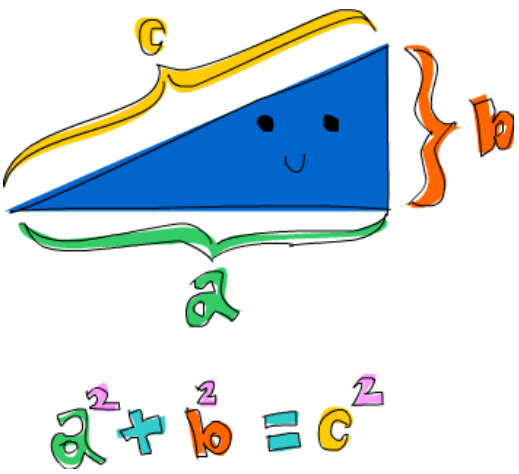
Як бачимо, функція `zip` об'єднує елементи в пари і створює новий список. Перший елемент іде в парі з першим, другий — із другим, і так далі. Зверніть увагу: оскільки пари мають в собі різні типи, `zip` бере два списки, що містять різні типи, і зліплює їх докупи. Що трапиться, якщо списки матимуть різну довжину?

```
ghci> zip [5,3,2,6,2,7,2,5,4,6,6] ["im","a","turtle"]
[(5,"im"),(3,"a"),(2,"turtle")]
```

Функція обрізає довший список, щоб він став завдовжки таким, як коротший список. Хаскел лінивий, тому скінченні списки можна застібати з нескінченними.

```
ghci> zip [1..] ["apple", "orange", "cherry", "mango"]
[(1,"apple"),(2,"orange"),(3,"cherry"),(4,"mango")]
```

Ось задача, яка поєднує кортежі та спискові характери: знайдемо прямокутний трикутник у якого довжина всіх сторін цілочисельна та менша або дорівнює 10, а периметр дорівнює 24. Спершу згенеруймо всі трикутники, сторони яких дорівнюють або менші за 10:




```
ghci> let triangles =
        [ (a,b,c) | c <- [1..10]
                  , b <- [1..10]
                  , a <- [1..10] ]
```

Ми беремо елементи із трьох списків, а наша функція виводу поєднує їх у триплет. Перевіримо результат, набравши в GHCi `triangles`. Ми отримаємо список усіх можливих трикутників, сторони яких дорівнюють або менші за 10. Далі додамо умову, що трикутник мусить бути прямокутним. Також змінимо цю функцію ще трохи — виключимо з розгляду повторення за допомогою умови: сторона b не більша за гіпотенузу, а сторона a не більша за сторону b.

```
ghci> let rightTriangles = [ (a,b,c) | c <- [1..10]
                                     , b <- [1..c]
                                     , a <- [1..b]
                                     , a^2 + b^2 == c^2]
```

Майже готово. Тепер ми востаннє змінимо нашу функцію — вкажемо, що нам потрібен трикутник, периметр якого дорівнює 24.

```
ghci> let rightTriangles' = [ (a,b,c) | c <- [1..10]
                                      , b <- [1..c]
                                      , a <- [1..b]
                                      , a^2 + b^2 == c^2
                                      , a + b + c == 24]
ghci> rightTriangles'
[(6,8,10)]
```

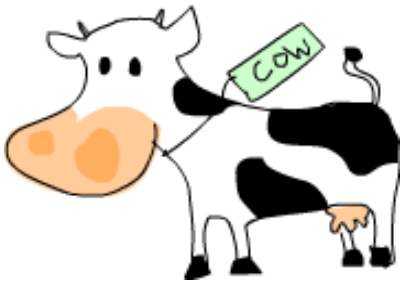
Ось і наша відповідь! Розв'язання задач у такий спосіб є поширеною ідіомою у функційному програмуванні: ви означуєте вихідну множину розв'язків, потім застосовуєте до них різного типу трансформації і фільтруєте їх, аж доки не отримаєте тільки бажані розв'язки.

Розділ 3

Типи і типокласи

Переклад українською Ганни Лелів

3.1 Повірте типові



Я вже згадував, що Хаскел має статичну систему типів. Тип кожного виразу є відомим під час компіляції, і тому код виходить надійніший. Якщо ви напишете програму, де ви спробуєте розділити булів тип із якимось числом, то вона навіть не скомпілюється. І це добре, адже краще вловити такі помилки під час компіляції, аніж мати аварійне завершення програми при виконанні. У Хаскелі все має тип, тому компілятор зробить чимало висновків про вашу програму ще до того, як її скомпілює.

На відміну від Java чи Pascal, Хаскел має виведення типів. Коли ми пишемо число, то не мусимо вказувати Хаскелу, що це є число. Він може самостійно це *вивести*, тому нам не треба явно розписувати типи функцій і виразів під час написання коду. В двох попередніх розділах ми зробили огляд головних ідей Хаскела, буквально у двох словах зачепивши типи. Проте, розуміння системи типів є дуже важливою частиною вивчення Хаскел.

Тип — це щось на кшталт ярлика, що його має кожен вираз. Він каже нам, якій категорії речей «підходить» цей вираз. Вираз `True` — це булів вираз, `”hello”` — це рядок, тощо.

А зараз скористаймося GHCi, щоб розглянути типи деяких виразів. У цьому нам допоможе команда `:t`, яка, якщо після неї подати коректний вираз, вкаже нам його тип. Ну що — поїхали!

```
ghci> :t 'a'
'a' :: Char
ghci> :t True
True :: Bool
ghci> :t "HELLO!"
"HELLO!" :: [Char]
ghci> :t (True, 'a')
(True, 'a') :: (Bool, Char)
ghci> :t 4 == 5
4 == 5 :: Bool
```

Застосування `:t` до виразу виводить той самий вираз, а після нього `::` і його тип. `::` читаємо як «має тип». Типи завжди пишуться з великої літери. Як бачимо, схоже на те, що `'a'` має тип `Char`. Не важко зрозуміти (якщо знати, як перекладається *character* з англійської!), що Char тут означає *символ*. `True` має тип `Bool`. Наразі все зрозуміло. Але що *це* таке?! Перевірка типу `"HELLO!"` вертає `[Char]`. Квадратні дужки позначають список. Отож, це читається *список символів*. На відміну від списків, кортежі різних довжин — різні типи. Тобто вираз `(True, 'a')` має тип `(Bool, Char)` тоді як вираз `('a','b','c')` матиме тип `(Char, Char, Char)`. Вираз `4 == 5` завжди вертатиме `False`, тому його тип є `Bool`.

Кожна функція теж має тип. Коли ми пишемо свої власні функції, то можемо, якщо захочемо, оголошувати їхні типи. Добрі програмісти завжди так і роблять, за винятком, можливо, випадків коли пишуть дуже короткі функції. Відтепер ми оголошуватимемо тип усіх функцій які ми писатимемо. Пригадуєте, як ми створили списковий характер, який фільтрує рядок так, щоб залишалися тільки великі літери? Ось як він виглядає разом із оголошенням типу.

```
removeNonUppercase :: [Char] -> [Char]
removeNonUppercase st = [ c | c <- st, c `elem` ['A'..'Z']]
```

`removeNonUppercase` має тип `[Char] -> [Char]`, тобто, ця функція перетворює рядок на рядок. Це тому що вона бере один рядок як параметр і повертає інший рядок як результат. Тип `[Char]` — ідентичний до `String`, тому зрозуміліше буде написати: `removeNonUppercase :: String -> String`. Ми не мусили оголосити тип цієї функції, тому що компілятор може самотужки вивести, що це функція, яка перетворює рядки на рядки. Але ми все-таки це зробили. А як написати тип функції, яка приймає кілька параметрів? Ось простенька

функція, яка бере три цілі числа та додає їх:

```
addThree :: Int -> Int -> Int -> Int
addThree x y z = x + y + z
```

Параметри відокремлюються `->`, а між параметрами і результатом обрахунку функції немає особливої різниці. Тип результату — це останній елемент в оголошенні, а перші три — це типи параметрів. Згодом я поясню, чому ми відокремлюємо і параметри і результат `->`, а не робимо якоїсь більш очевидної різниці між типом результату функції та типами параметрів на кшталт `Int, Int, Int -> Int` або ще якось.

Якщо ви хочете оголосити тип своєї функції, але не впевнені, яким саме він повинен бути, то просто напишіть функцію без оголошення, а тоді дізнайтеся її тип за допомогою `:t`. Функції — це вирази також, тож `:t` працюватиме з ними без жодних проблем.

Ось короткий огляд найпоширеніших типів.

Тип `Int` використовується для цілих чисел. Назва — скорочення англійського слова *integer*. `7` може бути `Int`, а `7.2` — ні. `Int` обмежений, тобто, він має мінімальне та максимальне значення. На 32-бітних машинах максимальне можливе значення `Int` зазвичай є 2147483647, а мінімальне — -2147483648.

Тип `Integer` — це теж цілі числа… [От лише назвою він ніяк не зв'язаний із *integer* з англійської… (жартую!)]. Різниця в тому, що цей тип необмежений, тобто він може представляти дійсно величезні числа. Просто гігантські. Водночас, `Int` — ефективніший.

```
factorial :: Integer -> Integer
factorial n = product [1..n]
```

```
ghci> factorial 50
30414093201713378043612608166064768844377641568960512000000000000
```

Тип `Float` — це дійсні числа, представлені як числа із плаваючою комою, одинарної точності.

```
circumference :: Float -> Float
circumference r = 2 * pi * r
```

```
ghci> circumference 4.0
25.132742
```

Тип `Double` — це дійсні числа також, представлені як числа із плаваючою комою, але подвійної точності!

```
circumference' :: Double -> Double
circumference' r = 2 * pi * r
```

```
ghci> circumference' 4.0
25.132741228718345
```

Тип `Bool` — це булів тип. Він може мати лише два значення: `True` і `False`.

Тип `Char` представляє символ. Символи беруться в ординарні лапки. Список символів — це рядок.

Кортежі — це типи, але вони залежать не тільки від типів своїх складових, а й від своєї довжини також. Тому, теоретично, існує нескінченна кількість типів кортежів, і її неможливо описати в цьому посібнику. Зверніть увагу, що порожній кортеж `()` — це тип також, і він може мати лише одне значення: `()`.

3.2 Змінні типу

Як ви гадаєте — який тип має функція `head`? `head` бере список з елементів будь-якого типу та повертає перший елемент. То якого вона типу? Перевірмо!

```
ghci> :t head
head :: [a] -> a
```

Гммм! Що таке `a`? Це тип? Пригадуєте, я вже казав, що типи пишуться з великої літери, отож це не може бути типом. Оскільки воно пишеться з малої літери — це **змінна типу**. А це значить, що `a` може мати будь-який тип. Схоже на узагальнені засоби в інших мовах програмування, але в Хаскелі така штука називається параметричним поліморфізмом, і в Хаскелі вона набагато могутніша, адже дає змогу легко написати дуже загальні функції (якщо вони не використовують ніякі спеціалізовані поведінки типів із якими оперують). Функції, в сиґнатурі яких є змінні типу, називаються **поліморфними функціями**. Оголошення типу функції `head` каже, що вона бере список з елементів будь-якого типу та повертає один елемент такого типу.

Імена змінних типу можуть містити понад один символ, але зазвичай їх називають a, b, c, d...

Пригадуєте функцію `fst`? Вона повертає перший компонент пари. Перевіримо її тип.

```
ghci> :t fst
fst :: (a, b) -> a
```

Як бачимо, `fst` бере кортеж, який містить два типи, та повертає елемент, який має такий самий тип, як перший компонент пари. Ось чому `fst` можна

застосувати для пари, що містить будь-які два типи. Зверніть увагу — `a` і `b` не мусять бути різного типу тільки тому, що це дві різні змінні типу. Тип функції всього лиш каже, що тип першого компонента і тип результату — одинакові.

3.3 Вступ до вступу до типокласів

Типоклас — це щось на кшталт інтерфейсу, який означує якусь поведінку. Якщо тип належить до певного типокласу, це означає, що він підтримує та реалізує поведінку, що її описує цей типоклас. Це збиває з пантелику чимало людей з шкіл об'єктно-орієнтованого програмування, тому що вони гадають, що це те саме, що класи у об'єктно-орієнтованих мовах. Але це не так. Типокласи нагадують Java інтерфейси. Тільки вони кращі.

Яка сиґнатура функції `==`?

```
ghci> :t (==)
(==) :: Eq a => a -> a -> Bool
```

Примітка: Оператор рівності `==` — це функція. Як і `+`, `*`, `-`, `/` та майже всі інші оператори. Якщо назва функції складається тільки зі спеціальних символів, функція за замовчуванням вважається інфіксною. Якщо ми хочемо перевірити її тип, передати її іншій функції чи викликати її як префіксну функцію, ми мусимо взяти її в дужки.

Цікаво. Ми побачили щось нове — символ `=>`. Усе, що передує символові `=>` називається **умовою типокласу**. Попереднє оголошення можна прочитати ось так: функція перевірки на рівність бере будь-які два значення однакового типу та повертає `Bool`. Тип обидвох значень мусить належати до типокласу `Eq` (це і була умова типокласу).

Типоклас `Eq` надає інтерфейс для перевірки на рівність. Будь-який тип, для двох значень якого має сенс перевірка на рівність, має належати до типокласу `Eq`. У Хаскелі усі стандартні типи — окрім IO (тип, що має справу з вводом і виводом) і функцій — належать до типокласу `Eq`.

Функція `elem` має тип `Eq a => a -> [a] -> Bool`, тому що вона використовує `==` для перевірки, чи є в списку потрібне нам значення.

Ось кілька основних типокласів:

Типоклас `Eq` використовують для типів, що підтримують перевірку на рівність. Його члени реалізують функції `==` і `/=`. Отож, якщо на змінну типу у функції накладено умову `Eq`, десь в означенні цієї функції використовується `==` або `/=`. Усі раніше згадані типи, окрім функцій, належать до `Eq`, тому значення таких типів можна перевіряти на рівність.

```
ghci> 5 == 5
True
ghci> 5 /= 5
False
ghci> 'a' == 'a'
True
ghci> "Ho Ho" == "Ho Ho"
True
ghci> 3.432 == 3.432
True
```

Типоклас `Ord` — для типів, що мають впорядковування.

```
ghci> :t (>)
(>) :: Ord a => a -> a -> Bool
```

Усі вищезгадані типи, окрім функцій, належать до `Ord`. `Ord` охоплює усі типові функції порівняння, як-от `>`, `<`, `>=` і `<=`. Функція `compare` бере два члени `Ord` однакового типу та повертає упорядкування. `Ordering` — це тип, що може мати три значення — `GT`, `LT` або `EQ`, що означає *більший ніж*, *менший ніж* і *рівний*, відповідно.

Щоб належати до `Ord`, тип мусить спершу бути членом престижного та ексклюзивного клубу `Eq`.

```
ghci> "Abrakadabra" < "Zebra"
True
ghci> "Abrakadabra" `compare` "Zebra"
LT
ghci> 5 >= 2
True
ghci> 5 `compare` 3
GT
```

Членів типокласу `Show` можна серіалізувати в рядок. Усі вищезгадані типи, окрім функцій, належать до `Show`. Найпоширеніша функція, що працює з типокласом `Show` — це `show`. Вона бере значення, тип якого належить до `Show`, переводить його в рядок і показує його.

```
ghci> show 3
"3"
ghci> show 5.334
"5.334"
ghci> show True
"True"
```

Типоклас `Read` — це по суті протилежний типоклас до `Show`. Функція `read` бере рядок та повертає тип, який належить до `Read`.

```
ghci> read "True" || False
True
ghci> read "8.2" + 3.8
12.0
ghci> read "5" - 2
3
ghci> read "[1,2,3,4]" ++ [3]
[1,2,3,4,3]
```

Наразі все ОК. Знову ж таки — усі вищезгадані типи належать до цього типокласу. А що трапиться, якщо виконати просто `read "4"`?

```
ghci> read "4"
<interactive>:1:0:
    Ambiguous type variable `a' in the constraint:
      `Read a' arising from a use of `read' at <interactive>:1:0-7
    Probable fix: add a type signature that fixes these type variable(s)
```

GHCi каже нам, що не знає, *що* ми хочемо отримати. Зверніть увагу: коли ми задіювали `read` раніше, ми потім якось використовували її результат. Завдяки цьому GHCi міг вивести, якого типу результат ми хотіли отримати від `read`. Якщо ми використовували його як булеве значення, то він повертав `Bool`. Але тепер, він знає, що ми хочемо якийсь тип, що належить до класу `Read`, але не знає, який саме. Погляньмо на типосигнатуру `read`.

```
ghci> :t read
read :: Read a => String -> a
```

Бачите? Він повернув тип, що належить до `Read`, але якщо ми не будемо його якось використовувати пізніше, немає можливості **визначити** [to determine] який саме тип треба. Ось для чого існують **анотації типу**. За допомогою анотацій типу можна явно вказати якого типу вираз має бути. Щоб створити таку анотацію треба додати `::` наприкінці виразу, а тоді вказати його тип. Дивіться:

```
ghci> read ”5” :: Int
5
ghci> read ”5” :: Float
5.0
ghci> (read ”5” :: Float) * 4
20.0
ghci> read ”[1,2,3,4]” :: [Int]
[1,2,3,4]
ghci> read ”(3, ’a’)” :: (Int, Char)
(3, ’a’)
```

Компілятор зазвичай може самостійно **вивести** [to infer] тип більшості виразів. Але часом компілятор не знає, чи повертати значення типу `Int` чи `Float` для виразу на кшталт `read ”5”`. Щоб **визначити** [to determine] тип, Хаскел мусить вирахувати `read ”5”`. Але ж Хаскел — це статично типізована мова, тому він мусить знати всі типи наперед, до компіляції (або, у випадку GHCi — до обрахунку). Тому мусимо сказати Хаскелові: «Гей, на випадок, якщо ти цього не знаєш, цей вираз повинен мати *оцей* тип!».

Члени типокласу `Enum` — це типи, втілення яких можна перелічити і впорядкувати. Основна перевага типокласу `Enum` полягає в тому, що його типи можна використовувати в діапазонах списків. Такі типи також мають означені наступні та попередні елементи, що їх можна отримати за допомогою функцій `succ` і `pred`, відповідно. Типи в цьому типокласі: `()`, `Bool`, `Char`, `Ordering`, `Int`, `Integer`, `Float` і `Double`.

```
ghci> [’a’..’e’]
”abcde”
ghci> [LT .. GT]
[LT,EQ,GT]
ghci> [3 .. 5]
[3,4,5]
ghci> succ ’B’
’C’
```

Члени типокласу `Bounded` мають верхню і нижню межу.

```
ghci> minBound :: Int
−2147483648
ghci> maxBound :: Char
’\1114111’
ghci> maxBound :: Bool
True
ghci> minBound :: Bool
```

```
False
```

`minBound` і `maxBound` варті уваги, тому що вони мають тип `Bounded a => a`. До певної міри, вони — поліморфні сталі.

Кортеж належить до `Bounded`, якщо до `Bounded` належить кожна з його компонент.

```
ghci> maxBound :: (Bool, Int, Char)
(True,2147483647,'\1114111')
```

Клас `Num` — це чисельний типоклас. Його елементи здатні діяти як числа. Розгляньмо тип числа:

```
ghci> :t 20
20 :: Num t => t
```

Здається, що цілі числа — це теж поліморфні сталі. Вони можуть поводитися як будь-який тип, що належить до типокласу `Num`.

```
ghci> 20 :: Int
20
ghci> 20 :: Integer
20
ghci> 20 :: Float
20.0
ghci> 20 :: Double
20.0
```

Це типи, що входять до типокласу `Num`. Коли ми перевіримо тип `*`, то побачимо, що він працює із усіма членами чисельного типокласу.

```
ghci> :t (*)
(*) :: Num a => a -> a -> a
```

Він бере два числа однакового типу та повертає число такого ж типу. Ось чому `(5 :: Int) * (6 :: Integer)` видасть помилку типу, тоді як `5 * (6 :: Integer)` спрацює і поверне `Integer`, адже `5` здатний поводитися як `Integer`, так і як `Int`.

Щоб приєднатися до `Num`, тип уже має приятелювати з `Show` і `Eq`.

Типоклас `Integral` — це також чисельний типоклас. `Num` включає всі числа — дійсні та цілі, а `Integral` — тільки цілі. До цього типокласу належать `Int` і `Integer`.

До типокласу `Floating` належать тільки числа з плаваючою комою, тобто `Float` and `Double`.

Функція `fromIntegral` — це вкрай корисна функція для роботи з числами. Її сигнатура — `fromIntegral :: (Num b, Integral a) => a -> b`, і з неї випливає, що ця функція бере ціле число з `Integral` та перетворює його у загальне число з `Num`. Вона стане вам у пригоді, коли ви захочете, щоб цілі числа і числа з плаваючою комою гарненько співпрацювали. До прикладу, сигнатура функції `length` є `length :: [a] -> Int`, а не більш загальна `length :: Num b => [a] -> b`. Напевно так склалося історично, хоча на мою думку — це тупо. Хай там як, якщо ми спробуємо отримати довжину списку, а тоді додати її до `3.2`, то отримаємо помилку, бо ми намагаємося додати докупи `Int` і число з плаваючою комою. Спробуймо це обійти — виконаймо `fromIntegral (length [1,2,3,4]) + 3.2` — і все супер.

Зверніть увагу, що `fromIntegral` має кілька обмежень класу у сигнатурі. Це цілком правильно. Як бачите, обмеження класу відокремлюються комами всередині дужок.

Розділ 4

Синтаксис у функціях

Переклад українською Ганни Лелів

4.1 Зіставлення із взірцем

Цей розділ розповість про кілька класних синтаксичних структур Хаскела, і ми розпочнемо із зіставлення із взірцем. Зіставлення із взірцем складається із означування взірців, яким повинні відповідати дані, перевірки чи дані дійсно відповідають цим взірцям, і, врешті, деконструювання даних відповідно до тих взірців.

Означуючи функції, можна означити різні тіла функцій для різних взірців. Це зазвичай веде до гарного — простого і читабельного — коду. Із взірцем можна зіставляти будь-які дані — числа, символи, списки, кортежі, тощо. Напишімо тривіальну функцію, яка перевіряє, чи число, яке ми надали, сімірка чи ні.

```
lucky :: Integral a => a -> String
lucky 7 = "LUCKY NUMBER SEVEN!"
lucky x = "Sorry, you're out of luck, pal!"
```

Коли ви викличете `lucky`, взірці перевірятимуться зверху до низу, і коли зіставлення буде успішним, буде використано відповідне тіло функції. Єдиний спосіб, в який число може зіставитись із першим взірцем, це бути 7. Якщо ж число не є 7, пошук перейде до другого взірця, який підходить до будь-чого і прив'яже те будь-що до імені `x`. Цю фун-

кцію також можна було б реалізувати, використавши інструкцію розгалуження. А що, якби ми захотіли мати функцію, яка впізнає числа від 1 до 5 і видає `"Not between 1 and 5"` для решти чисел? Якби не було ідіоми зіставлення із взірцем довелося б написати доволі заплутане дерево вибору «якщо-тоді-інакше». Однак, із нею, маємо:

```
sayMe :: Integral a => a -> String
sayMe 1 = "One!"
sayMe 2 = "Two!"
sayMe 3 = "Three!"
sayMe 4 = "Four!"
sayMe 5 = "Five!"
sayMe x = "Not between 1 and 5"
```

Зауважте, що якби ми поставили останній взірець (який усе ловить) першим, то завжди отримували б напис `"Not between 1 and 5"`, адже він ловив би всі числа, і в них не було б можливості «провалитися» до наступних взірців і пройти зіставлення з ними.

Пригадуєте функцію факторіалу, яку ми реалізували раніше? Ми означили факторіал числа `n` як `product [1..n]`. Ми також можемо означити функцію факторіалу *рекурсивно* — так, як її зазвичай означують у математиці. Спершу ми вказуємо, що факторіал 0 є 1. Тоді ми означуємо, що факторіал будь-якого додатного цілого числа є це число помножене на факторіал його попередника. Ось як це виглядає мовою Хаскела:

```
factorial :: Integral a => a -> a
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

Ми вперше означили функцію рекурсивно. У Хаскелі рекурсія дуже важлива, тому згодом ми розглянемо її детальніше. Але, у двох словах, ось що трапиться, коли ми спробуємо добути факторіал числа 3, наприклад. Розрахунки почнуться з `3 * factorial 2`. Факторіал 2 — `2 * factorial 1`, отож наразі маємо `3 * (2 * factorial 1)`. `factorial 1` дорівнює `1 * factorial 0`, отож ми отримуємо `3 * (2 * (1 * factorial 0))`. І ось ми дійшли до спритного трюку — ми означили факторіал 0 як 1, і оскільки ми натрапляємо на цей взірець першим, до універсального взірця справа не доходить, і тому повертається 1. Отож, остаточний результат є еквівалентним `3 * (2 * (1 * 1))`. Якби ми написали другий взірець вгорі над першим, він би ловив усі числа, 0 включно, і наші обчислення ніколи б не завершилися. Ось чому коли ми означуємо взірці, так важливо подавати означення у правильному порядку і означувати найбільш конкретні взірці першими, а більш загальні — пізніше.

Зіставлення із взірцем може закінчитися невдачею. Якщо ми означимо функцію ось так:

```
charName :: Char -> String
charName 'a' = "Albert"
charName 'b' = "Broseph"
charName 'c' = "Cecil"
```

а тоді спробуємо викликати її із вхідними даними, про які ми не подбали, ось що трапиться:

```
ghci> charName 'a'
"Albert"
ghci> charName 'b'
"Broseph"
ghci> charName 'h'
*** Exception: tut.hs:(53,0)-(55,21):
    Non-exhaustive patterns in function charName
```

Справедливі нарікання на невичерпуючі взірці. Створюючи групу взірців, ми завжди мусимо включити до неї універсальний взірець, щоб наша програма не завершилася аварійно через неочікувані вхідні дані.

Зіставлення із взірцем можна використати і для кортежів. А що якби ми захотіли створити функцію, яка отримує два вектори у 2D просторі (і вектор представлено у формі пари) і додає їх? Щоб додати два вектори, ми окремо додаємо їхні компоненти x, а тоді окремо їхні компоненти y. Ось як би ми зробили, якби ми нічого не знали про зіставлення із взірцями:

```
addVectors :: Num a => (a, a) -> (a, a) -> (a, a)
addVectors a b = (fst a + fst b, snd a + snd b)
```

Так, працює. Але існує кращий спосіб. Спробуймо змінити цю функцію так, щоб вона використовувала зіставлення із взірцем.

```
addVectors :: Num a => (a, a) -> (a, a) -> (a, a)
addVectors (x1, y1) (x2, y2) = (x1 + x2, y1 + y2)
```

Ось так! У сто разів краще. Зверніть увагу що це вже і є універсальний взірець. Тип `addVectors` (в обох випадках) — це `addVectors :: Num a => (a, a) -> (a, a) -> (a, a)`, отож ми точно отримаємо дві пари як параметри.

`fst` and `snd` виокремлюють компоненти пар. А як щодо трійок? Наразі ми не маємо жодних готових функцій для трійок, але можемо написати свої власні.

```
first :: (a, b, c) -> a
```

```
first (x, _, _) = x

second :: (a, b, c) -> b
second (_, y, _) = y

third :: (a, b, c) -> c
third (_, _, z) = z
```

`_` означає те ж саме, що й у спискових характерах: нам байдуже що воно таке та частина, і тому ми просто пишемо `_`.

А це нагадує мені, що зіставляти із взірцями можна і в спискових характерах. Ось, погляньте:

```
ghci> let xs = [(1,3), (4,3), (2,4), (5,3), (5,6), (3,1)]
ghci> [a + b | (a,b) <- xs]
[4,7,6,8,11,4]
```

Якщо зіставлення зі взірцем зазнає невдачі, то воно просто перейде до наступного елемента.

Списки також можна використати у зіставленні із взірцем. Можна зіставити з порожнім списком `[]` чи будь-яким взірцем, куди входить `:` і порожній список. Але оскільки `[1,2,3]` — це просто синтаксичний цукор для `1:2:3:[]`, ви можете використати вже відомий нам взірець. Такий взірець, як-от `x:xs`, прив'яже початок списку до `x`, а решту — до `xs`. Якщо у списку буде тільки один елемент, то `xs` буде зв'язано із порожнім списком.

Примітка: Взірець `x:xs` часто використовують, особливо з рекурсивними функціями. Але взірці, які містять `:`, можна зіставити тільки зі списками, що мають довжину 1 чи більше.

Якщо ви, скажімо, хочете прив'язати перші три елементи до змінних, а решту списку до іншої змінної, ви можете скористатися `x:y:z:zs`. Його можна зіставити тільки зі списками, які містять три і більше елементів.

Тепер ми вже знаємо, як зіставляти списки зі взірцями, тому спробуймо реалізувати нашу власну функцію `head`.

```
head' :: [a] -> a
head' []    = error "Can't call head on an empty list, dummy!"
head' (x:_) = x
```

Перевіримо, чи працює:

```
ghci> head' [4,5,6]
4
ghci> head' "Hello"
'H'
```

Супер! Зауважте, що, якщо ви хочете прив'язати до кілька змінних (навіть якщо одна з них — це просто підкреслення `_`, яке ні до чого не прив'язується), ми повинні взяти все в дужки. Зверніть також увагу на функцію `error`, яку ми використали. Вона бере рядок і генерує помилку виконання, використовуючи той рядок як інформацію про те, яка саме помилка трапилася. Через неї програма завершується аварійно, тому не радимо використовувати цю функцію занадто часто. Але викликати `head` із порожнім списком — безглуздо.

Напишімо тривіальну функцію, яка розповідає нам про кілька перших елементів списку у (не)зручній формі англійською.

```
tell :: Show a => [a] -> String
tell []       = "The list is empty"
tell (x:[])   = "The list has one element: " ++ show x
tell (x:y:[]) = "The list has two elements: " ++ show x ++ " and "
                ++ show y
tell (x:y:_)  = "This list is long. The first two elements are: "
                ++ show x ++ " and " ++ show y
```

Ця функція безпечна, оскільки вона працює з порожнім списком, одноелементним списком, списком із двома елементами і списком із більш ніж двома елементами. Зверніть увагу, що `(x:[])` і `(x:y:[])` можна переписати як `[x]` і `[x,y]`, відповідно (оскільки це синтаксичний цукор, тут нам дужки не потрібні). Ми не можемо переписати `(x:y:_)` із квадратними дужками, оскільки його можна зіставити з будь-яким списком, завдовжки 2 чи більше.

Ми вже реалізували нашу власну функцію `length` за допомогою спискових характерів. Зараз ми спробуємо зробити те саме, використовуючи зіставлення із взірцем і невеличку рекурсію:

```
length' :: Num b => [a] -> b
length' []     = 0
length' (_:xs) = 1 + length' xs
```

Це нагадує функцію факторіалу, яку ми написали раніше. Спочатку ми означили результат для відомих **вхідних даних** [input data] — порожнього списку. Цей взірець також відомий під назвою граничної умови. Тоді в другому взірці ми розбираємо список, поділивши його на голову і хвіст. Ми вказуємо, що довжина дорівнює 1 плюс довжина хвоста. `_` зіставляється з головою, бо нам, по

суті, байдуже, що то є. Зверніть увагу що ми подбали про всі можливі варіанти для списку: перший взірець зіставляється із порожнім списком, а другий — з усім, що не є порожнім списком.

Погляньмо, що трапиться, якщо ми викличемо `length'` по `"ham"`. Спершу воно перевірить, чи це не порожній список. Оскільки список не порожній, перехід буде здійснено до другого взірця. Далі відбувається зіставлення із другим взірцем, а він стверджує, що довжина дорівнює `1 + length' "am"`, оскільки ми поділили список на голову і хвіст і відкинули голову. Гаразд. `length'` від `"am"` дорівнює `1 + length' "m"`. Отож, ми маємо `1 + (1 + length' "m")`. `length' "m"` дорівнює `1 + length' ""` (можемо також написати `1 + length' []`). І ми означили `length' []` як `0`. Тому врешті-решт ми отримуємо `1 + (1 + (1 + 0))`.

Реалізуймо `sum`. Ми знаємо, що сума порожнього списку дорівнює 0. Ми записуємо це першим взірцем. Нам також відомо, що сума списку — це голова плюс сума решти списку. Якщо ми це все запишемо разом, отримаємо:

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

Є ще така штука як *взірці із ім'ям*. Це зручний спосіб розвалити щось на складові відповідно до взірця і поіменувати складові і, водночас, ще й надати ім'я для всієї купи. Це досягається написанням імені і `@` як префіксу до взірця. Наприклад, взірець `xs@(x:y:ys)`. Цей взірець зіставиться точнісінько з тим самим, що й `x:y:ys`, але ви також одним маєте змогу отримати цілий список за допомогою `xs`, і не треба буде повторюватися, знову набираючи `x:y:ys` у тілі функції. Ось простий як двері приклад:

```
capital :: String -> String
capital ""         = "Empty string, whoops!"
capital all@(x:xs) = "The first letter of " ++ all ++ " is " ++ [x]
```

```
ghci> capital "Dracula"
"The first letter of Dracula is D"
```

Зазвичай ми використовуємо взірці з іменами, щоб уникнути повторів, коли зіставляємо щось із взірцем, а у відповідному тому взірцеві тілі функції треба звертатись не тільки до того щось в цілому, а й також треба доступитися до його складових.

Ще одне — у зіставленні із взірцем не можна використовувати `++`. Якщо ви спробуєте зіставити `(xs ++ ys)` із взірцем, то що ж опиниться в першому, а що у другому списках? Це немає сенсу робити через неоднозначність. Але от має

сенс зіставити із `(xs ++ [x,y,z])` або просто `(xs ++ [x])`, але це неможливо зробити через однозв'язну природу списків в Хаскелі.

4.2 Варта, варта!

В той час як взірці перевіряють чи відповідає **вхід** [input] певній формі і деконструюють його згідно неї, вартові перевіряють, чи певна властивість (чи властивості) вхідних даних є правдивою чи хибною. Це дуже нагадує інструкцію розгалуження, і дійсно — вони дуже схожі. Справа в тому, що вартові легше читаються коли ми маємо кілька умов і вони особливо зручні при використанні в тандемі із взірцями.

Не буду пояснювати їхній синтаксис — краще спробуймо відразу написати функцію, використовуючи вартові. Напишемо просту функцію, яка вас похвалить чи, навпаки, насварить, залежно від того, який у вас індекс маси тіла (ІМТ). Щоб порахувати ІМТ, треба поділити свою вагу на зріст в квадраті. Якщо ваш ІМТ менший за 18.5, у вас недостатня маса тіла. Якщо він становить від 18.5 до 25, у вас нормальна вага. Від 25 до 30 — у вас надлишкова маса тіла, а понад 30 — ожиріння. Ось функція (ми не будемо зараз нічого обчислювати, ця функція просто отримує ІМТ і сварить вас):

```
bmiTell :: RealFloat a => a -> String
bmiTell bmi
    | bmi <= 18.5 = "You're underweight, you emo, you!"
    | bmi <= 25.0 = "You're supposedly normal. " ++
                    "Pffft, I bet you're ugly!"
    | bmi <= 30.0 = "You're fat! Lose some weight, fatty!"
    | otherwise   = "You're a whale, congratulations!"
```

Вартові позначаються вертикальними рисками (трубами), які ставляться після імені функції та її параметрів. Зазвичай їх трохи зсувають вправо і вишикують. Вартовий — це, по суті, булів вираз. Якщо результатом обрахунку вартового є `True`, тоді використовується відповідне тому вартовому тіло функції. Якщо `False` — контроль переходить до наступного вартового і так далі. Якщо подамо цій функції `24.3`, вона спершу перевірить, чи це число менше або дорівнює `18.5`. Оскільки це не так, перевірка переходить до наступного вартового. Другий вартовий перевіряє число, і оскільки `24.3` менше, ніж `25.0`, другий вартовий «пропускає» і повертається другий рядок.

Це дуже схоже на велике дерево «якщо-тоді-інакше» в імперативних мовах, тільки воно набагато краще і зручніше до відчитання. Великі дерева «якщо-

тоді-інакше» не бажано використовувати, але часом задача означена так, що без дерева ніяк не обійдешся. Вартові — чудова альтернатива таким деревам.

Дуже часто останній вартовий — це `otherwise`. `otherwise` означене просто як `otherwise = True` і він вловлює все. Схоже на взірці, але ті лише перевіряють, чи вхідні дані узгоджуються із ними, тоді як вартові здійснюють перевірку булевих умов. Якщо всі вартові функції після обрахування дорівнюватимуть `False` (і ми не дописали універсального вартового `otherwise`), оцінювання переходить до наступного **взірця**. Ось як взірці та вартові чудово співпрацюють. Якщо не знайдено відповідного вартового чи взірця, буде викинуто помилку.

Звісно, що вартових можна використовувати з функціями, які приймають стільки параметрів, скільки нам треба. Щоб не примушувати користувача самотужки рахувати свій ІМТ перед тим, як викликати функцію, спробуймо змінити цю функцію так, щоб вона брала зріст і вагу і сама виконувала всі обрахунки.

```
bmiTell :: RealFloat a => a -> a -> String
bmiTell weight height
    | weight / height ^ 2 <= 18.5 = "You're underweight, you emo, you!"
    | weight / height ^ 2 <= 25.0 = "You're supposedly normal. " ++
                                    "Pffft, I bet you're ugly!"
    | weight / height ^ 2 <= 30.0 = "You're fat! " ++
                                    "Lose some weight, fatty!"
    | otherwise                   = "You're a whale, congratulations!"
```

Подивимось, чи я товстий...

```
ghci> bmiTell 85 1.90
"You're supposedly normal. Pffft, I bet you're ugly!"
```

Ура! Я не товстий! Але Хаскел сказав, що я бридкий! Ну й хай собі!

Зверніть увагу, що між параметрами функції і першим вартовим не ставиться `=`. Чимало новачків отримують синтаксичні помилки бо вони туди його тулять.

Ще один простенький приклад: реалізуймо свою власну функцію `max`. Як ви пригадуєте, вона бере два значення, які можна порівнювати, і повертає більше з них.

```
max' :: Ord a => a -> a -> a
max' a b
    | a > b    = a
    | otherwise = b
```

Вартових можна писати одним рядком тексту також, але краще так не робити, бо тоді код важче читатиметься, навіть якщо функції коротенькі. Проте для

прикладу можемо записати `max'` ось так:

```
max' :: Ord a => a -> a -> a
max' a b | a > b = a | otherwise = b
```

Бррр! Прочитати взагалі неможливо! Йдемо далі: реалізуймо свою власну `compare`, використовуючи вартових.

```
myCompare :: Ord a => a -> a -> Ordering
a `myCompare` b
    | a > b     = GT
    | a == b    = EQ
    | otherwise  = LT
```

```
ghci> 3 `myCompare` 2
GT
```

Примітка: Ми можемо не тільки викликати функції інфіксно (за допомогою спадного наголосу), а й означувати їх інфіксно також. Такі означення часом легше читаються.

4.3 Де!?

У попередньому розділі ми означили функцію, яка рахує ІМТ і висловлює своє «фе», ось так:

```
bmiTell :: RealFloat a => a -> a -> String
bmiTell weight height
    | weight / height ^ 2 <= 18.5 = "You're underweight, you emo, you!"
    | weight / height ^ 2 <= 25.0 = "You're supposedly normal. " ++
                                    "Pffft, I bet you're ugly!"
    | weight / height ^ 2 <= 30.0 = "You're fat! " ++
                                    "Lose some weight, fatty!"
    | otherwise                   = "You're a whale, congratulations!"
```

Зверніть увагу, що ми маємо три повтори. Ми повторюємося тричі. У програмуванні повторитися (тричі) — це так само добре, як отримати по голові. Оскільки ми повторюємо той самий вираз тричі, було б чудово, якби ми могли порахувати його один раз, прив'язати до імені, а тоді використати ім'я замість виразу. Можемо змінити нашу функцію ось так:

```
bmiTell :: RealFloat a => a -> a -> String
bmiTell weight height
```

```
    | bmi <= 18.5 = "You're underweight, you emo, you!"
    | bmi <= 25.0 = "You're supposedly normal. " ++
                    "Pffft, I bet you're ugly!"
    | bmi <= 30.0 = "You're fat! Lose some weight, fatty!"
    | otherwise   = "You're a whale, congratulations!"
    where bmi = weight / height ^ 2
```

Ми пишемо ключове слово `where` після вартових (і перед ним найліпше давати такий самий відступ, як перед трубами), а тоді означуємо кілька змінних чи функцій. Ці імена видно усім вартовим, і ми більше не мусимо повторюватися. Якщо ми захочемо порахувати ІМТ по-іншому, нам треба внести зміни тільки один раз. По-друге, така річ як `weight / height ^ 2` тепер має ім'я `bmi`, і наш код стає легше читати. По-третє, наша програма працюватиме швидше (потенційно), адже `bmi` буде пораховано всього лиш один раз. А можемо трохи переборщити і записати цю функцію ось так:

```
bmiTell :: RealFloat a => a -> a -> String
bmiTell weight height
    | bmi <= skinny = "You're underweight, you emo, you!"
    | bmi <= normal = "You're supposedly normal. " ++
                      "Pffft, I bet you're ugly!"
    | bmi <= fat    = "You're fat! Lose some weight, fatty!"
    | otherwise     = "You're a whale, congratulations!"
    where bmi = weight / height ^ 2
          skinny = 18.5
          normal = 25.0
          fat = 30.0
```

Імена з блоку `where` видимі тільки у функції до якої цей блок належить, і тому вони не засмітять простору імен інших функцій. Зверніть увагу, що всі імена вирівняні в одну колонку. Якщо ми їх гарненько і правильно не вирівняємо, Хаскел заплутається, адже тоді він не знатиме, що всі вони належать до одного блоку.

Тіла функцій що відповідають різним взірцям не можуть звертатися до спільного блоку `where`. Якщо ви хочете, щоб кілька взірців однієї функції мали доступ до якогось спільного імені, його потрібно буде означити глобально.

В зв'язках з блоку `where` можна також **зіставляти із взірцем**! Ми б могли переписати блок `where` з нашої попередньої функції ось як:

```
...
    where bmi = weight / height ^ 2
          (skinny, normal, fat) = (18.5, 25.0, 30.0)
```

Напишімо ще одну досить тривіальну функцію, де ми надаємо ім'я і прізвище людини і отримуємо у відповідь її ініціали.

```
initials :: String -> String -> String
initials firstname lastname = [f] ++ ". " ++ [l] ++ "."
    where (f:_) = firstname
          (l:_) = lastname
```

Ми б могли зіставити безпосередньо у параметрах функції (і, власне, то було б коротше і зрозуміліше), але я просто хочу показати, що це можна зробити і в зв'язках з `where`.

Ми означували сталі в блоках `where`, але там можна означувати також і функції. Не зраджуючи нашому здоровому способу програмування, створімо функцію, яка бере список пар вага-зріст і повертає список з кількох ІМТ.

```
calcBmis :: RealFloat a => [(a, a)] -> [a]
calcBmis xs = [bmi w h | (w, h) <- xs]
    where bmi weight height = weight / height ^ 2
```

І це все! У цьому прикладі ми означили функцію `bmi` тому що нам її треба буде викликати багато разів для кожної пари зі списку що подано як параметр. Ми мусимо опрацювати список що надходить, а там для кожної пари — різний ІМТ.

Зв'язки `where` бувають і вкладеними. Це поширена ідіома — означувати допоміжну функцію в секції `where` іншої функції. Ну а далі і в допоміжній функції можна означувати нові функції, кожна з яких може мати свою секцію `where`.

4.4 Let it be[†]

Зв'язки `let` дуже схожі на зв'язки `where`. Зв'язки `where` — це синтаксична конструкція, що дає вам змогу зробити прив'язку до змінних у кінці функції, і змінні буде видно в усьому тілі тієї функції, у тому числі і в вартових. А от зв'язки `let` дозволяють робити прив'язку до змінних будь-де. До того ж, вони самі є виразами (а не інструкціями). Оскільки зв'язки `let` є дуже локальними означення з `let` «не перебігають» з вартового до вартового. Як і будь-які конструкції в Хаскелі, які прив'язують значення до імен, зв'язки `let` добре працюють із зіставленням із взірцем. Перевіримо `let` на практиці! Ось як можна означити функцію, що вираховує площу поверхні циліндра з його висоти та радіуса:

[†] **«Хай буде так» — назва останнього студійного альбому гурту The Beatles.**

```
cylinder :: RealFloat a => a -> a -> a
cylinder r h =
    let sideArea = 2 * pi * r * h
        topArea  = pi * r ^ 2
    in  sideArea + 2 * topArea
```

Загальна форма отака: `let` <зв'язки> `in` <вираз>. Імена, що означені у частині що йде після `let`, доступні в виразі після `in`. Як бачите, ми могли б це означити і за допомогою блоку `where`. Зверніть увагу, що імена також вирівняні в одну колонку. Отож, у чому різниця між цими двома підходами? Наразі здається, що `let` спочатку подає зв'язки, а пізніше — вираз у якому вони використовуються, тоді як `where` робить навпаки.

Різниця полягає в тому, що зв'язки `let` є виразами. Зв'язки `where` є всього лиш синтаксичними конструкціями (інструкціями). Пригадуєте, коли ми говорили про інструкцію розгалуження, я пояснював, що інструкція «якщо-тоді-інакше» є також виразом, і тому її можна запхнути куди завгодно?[†]

```
ghci> [if 5>3 then "Woo" else "Boo", if 'a'>'b' then "Foo" else "Bar"]
["Woo", "Bar"]
ghci> 4 * (if 10 > 5 then 10 else 0) + 2
42
```

Те ж саме можна зробити і зі зв'язками `let`.

```
ghci> 4 * (let a = 9 in a + 1) + 2
42
```

`let` також використовують для означення функцій локально:

```
ghci> [let square x = x * x in (square 5, square 3, square 2)]
[(25,9,4)]
```

Якщо ми хочемо означити кілька змінних в одному рядку тексту, ми, звичайно ж, не можемо вишикувати їх у колонку. Але ми можемо відокремлювати їх крапкою з комою.

```
ghci> (let a = 100; b = 200; c = 300 in a*b*c, let foo="Hey ";
bar = "there!" in foo ++ bar)
```

[†]Інструкції не повертають результати, а вирази — так.

```
(6000000,"Hey there!")
```

Після останньої зв'язки не обов'язково ставити крапку з комою, але якщо хочете — будь ласка. Як я вже казав, в зв'язках `let` можна зіставляти із взірцем. Це допоможе швидко розвалити кортеж на складові і прив'язати їх до імен. Отаке.

```
ghci> (let (a,b,c) = (1,2,3) in a+b+c) * 100
600
```

Зв'язки `let` можна вставляти всередину спискових характерів. Спробуймо переписати наш попередній приклад де обробляється список пар вага-зріст, використовуючи `let` всередині спискового виразу (а не означуючи допоміжну функцію за допомогою `where`, як це було раніше).

```
calcBmis :: RealFloat a => [(a, a)] -> [a]
calcBmis xs = [bmi | (w, h) <- xs, let bmi = w / h ^ 2]
```

Ми вставляємо `let` всередину спискового характеру так само, як предикат, але він не фільтрує список, а лише прив'язує до імен. Імена, означені в `let` всередині спискового виразу є видимими у функції виводу (частина перед `|`), а також для всіх предикатів і секцій, що йдуть після цієї зв'язки. То ж ми могли б змусити нашу функцію повертати лише ІМТ товстунів:

```
calcBmis :: RealFloat a => [(a, a)] -> [a]
calcBmis xs = [bmi | (w, h) <- xs, let bmi = w / h ^ 2, bmi >= 25.0]
```

Ми не можемо використати ім'я `bmi` у частині `(w, h) <- xs`, оскільки вона була означена перед зв'язкою `let`.

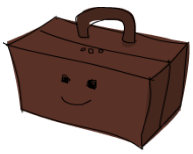
Ми не писали `in` у зв'язці `let`, коли використали її у списковому характері, оскільки в характерах видимість імен уже означено. Проте ми могли б використати зв'язку `let` із `in` у предикаті, і тоді означені там імена були б видимими тільки для того предиката. Означуючи функції та константи безпосередньо в GHCi, ми теж можемо вилучити частину `in`. У такому випадку імена будуть видимими під час усієї інтерактивної сесії.

```
ghci> let zoot x y z = x * y + z
ghci> zoot 3 9 2
29
ghci> let boot x y z = x * y + z in boot 3 4 2
14
ghci> boot
<interactive>:1:0: Not in scope: `boot'
```

Ви запитаєте: якщо зв'язки `let` такі круті, то чому б їх не використовувати постійно замість `where`? Справа в тім, що зв'язки `let` є виразами і тому доволі

локальні. Зокрема, вони не поширюються на вартових. Дехто надає перевагу зв'язкам `where`, тому що імена йдуть після функції, у якій вони використовуються. Тоді тіло функції ближче до її імені та оголошення її типу (в просторі коду), а такий код декому легше читати.

4.5 Вирази вибору



Чимало імперативних мов (C, C++, Java і так далі) мають конструкцію case, і якщо ви колись програмували такими мовами, то напевно знаєте, про що йдеться — про те, щоб взяти змінну, і виконувати різні блоки коду для різних її конкретних значень, із можливістю долучення універсального блоку коду на випадок, якщо змінна приймає якесь значення, для якого ми не налаштували спеціального блоку коду для обробки.

Хаскел запозичує це поняття і вдосконалює його. Як можемо здогадатися із назви, вирази вибору — це, нууууу, вирази, майже такі ж як вирази «якщо-тоді-інакше» і зв'язки `let`. Ми не тільки можемо обчислювати вирази, залежно від того, яке значення набуває змінна, а й зіставляти із взірцем. Хммм, взяти змінну, зіставити її із взірцем, порахувати щось залежно від її значення — де ми чули про це раніше? Ага! Зіставлення із взірцем в означеннях функцій! Насправді, такі означення є всього лише синтаксичним цукром для виразів вибору. Ці два шматки коду роблять те ж саме і є взаємозамінними:

```
head' :: [a] -> a
head' []    = error "No head for empty lists!"
head' (x:_) = x
```

```
head' :: [a] -> a
head' xs = case xs of []    -> error "No head for empty lists!"
                      (x:_) -> x
```

Бачите — вирази вибору мають простенький синтаксис:

```
case expression of pattern -> result
                   pattern -> result
                   pattern -> result
                   ...
```

`expression` зіставляється із взірцем. Зіставляння із взірцем відбувається як завжди: повертається `result` для першого взірця який зіставляється з `expression`. Якщо провалюється повз увесь вираз вибору і відповідного взірця (і зіставлення) немає — отримуємо помилку виконання.

В той час як параметри функцій можна зіставити із взірцем лише під час означування функцій, вирази вибору можна використовувати де завгодно. До прикладу:

```
describeList :: [a] -> String
describeList xs = "The list is " ++ case xs of
    []  -> "empty."
    [x] -> "a singleton list."
    xs  -> "a longer list."
```

Вирази вибору стануть в пригоді, коли ми зіставлятимемо щось із взірцем посередині виразу. Повторюючись, наголосимо, що зіставлення із взірцем в означеннях функцій — це синтаксичний цукор для виразів вибору, тому цю функцію можна було б означити і так:

```
describeList :: [a] -> String
describeList xs = "The list is " ++ what xs
    where what []  = "empty."
          what [x] = "a singleton list."
          what xs  = "a longer list."
```

Розділ 5

Рекурсія

Переклад українською Марини Стрельчук

5.1 Привіт, рекурсіє!

Ми познайомилися із рекурсією у попередньому розділі, а у цьому ми її розглянемо детальніше. Зокрема, поговоримо, про те, що таке рекурсія, чому вона важлива в Хаскелі і як можна отримувати компактні та елегантні розв'язки задач, якщо думати про них рекурсивно.

Якщо ви все ще не знаєте, що таке рекурсія, прочитайте це речення. Ха-ха! Жартую! Насправді, рекурсія — це спосіб означування функцій, де функція, яку ми означуємо, використовується всередині свого ж означення. В математиці означення часто даються рекурсивно. Наприклад, послідовність Фібоначчі означується рекурсивно. Спочатку ми означуємо два перших числа Фібоначчі нерекурсивно. Ми говоримо, що $F(0) = 0$ і $F(1) = 1$; це означає, що 0-ве і 1-ше числа Фібоначчі дорівнюють 0 і 1 відповідно. Для будь-якого іншого натурального числа, відповідне число Фібоначчі є сумою двох попередніх чисел Фібоначчі. Таким чином, $F(n) = F(n-1) + F(n-2)$. Отже, $F(3)$ рівне $F(2) + F(1)$, що дорівнює $(F(1) + F(0)) + F(1)$. Оскільки кінцевий вираз містить лише нерекурсивно означені числа Фібоначчі, ми можемо з упевненістю сказати, що $F(3)$ дорівнює 2. В означенні рекурсії наявність одного або двох елементів, які задані нерекурсивно (як от $F(0)$ та $F(1)$ з нашого прикладу), на-

зивають **граничною умовою**. Граничні умови є важливими і їх треба задавати, якщо ви хочете, щоб ваша рекурсивна функція мала змогу завершити свою роботу. Якби ми не означили $F(0)$ та $F(1)$ нерекурсивно, ми б ніколи не отримали відповіді, незалежно від того, яке б значення не подавали: досягнувши 0, ми б тоді просто перейшли до від'ємних чисел. Бац, і раптом вже вираховуємо $F(-2000)$, що є $F(-2001) + F(-2002)$, і, отже, мусимо продовжувати далі — а кінця розрахунків наразі не видно!

Рекурсія є важливою в Хаскелі, тому що, на відміну від імперативних мов, в Хаскелі ми робимо обчислення в спосіб задання *означення* того, **що** ми хочемо отримати, замість *описання*, як його треба отримувати. Саме тому в Хаскелі і немає циклів while та for, а замість них нам частенько доводиться використовувати рекурсію.

5.2 Максимум крутизни

Функція `maximum` бере список об'єктів, які можуть бути впорядковані (втілення типокласу `Ord`), і повертає найбільший із них. Подумайте, як ви б реалізували це на імперативний манер. Ви, напевно, створили б змінну, в якій зберігали б поточне максимальне значення, і після того в циклі порівнювали б його з кожним елементом списку. Якщо поточне максимальне значення менше за значення елементу, ви б замінили його на цей елемент. Максимальне значення, яке залишається в кінці, і є результатом. Хух! Доволі багато слів пішло на описання такого простого алгоритму!

Тепер розгляньмо рекурсивне означення. Для початку створімо граничну умову, де скажемо, що максимум одноелементного списку дорівнює значенню єдиного елементу цього списку. Далі кажемо, що максимум більш довгого списку дорівнює голові, якщо вона більша ніж максимум хвоста даного списку. Якщо ж максимальне значення хвоста більше за голову, тоді це значення і є максимумом списку. Ось і все! Тепер реалізуймо це на Хаскелі.

```
maximum' :: Ord a => [a] -> a
maximum' [] = error "maximum of empty list"
maximum' [x] = x
maximum' (x:xs)
    | x > maxTail = x
    | otherwise = maxTail
    where maxTail = maximum' xs
```

Як бачимо, зіставлення із взірцем чудово поєднується із рекурсією! Більшість імперативних мов не використовують зіставлення із взірцем, тому необхідно вживати інструкції розгалуження для перевірки граничних умов. Ну а

ми просто використовуємо тут взірці. Таким чином, перша гранична умова каже: якщо список пустий, завершуємося аварійно! Таке твердження є логічним, адже що є максимумом порожнього списку? Уявлення не маю. Другий взірець є реалізацією другої граничної умови. Якщо це одноелементний список, ми просто повертаємо єдиний елемент цього списку.

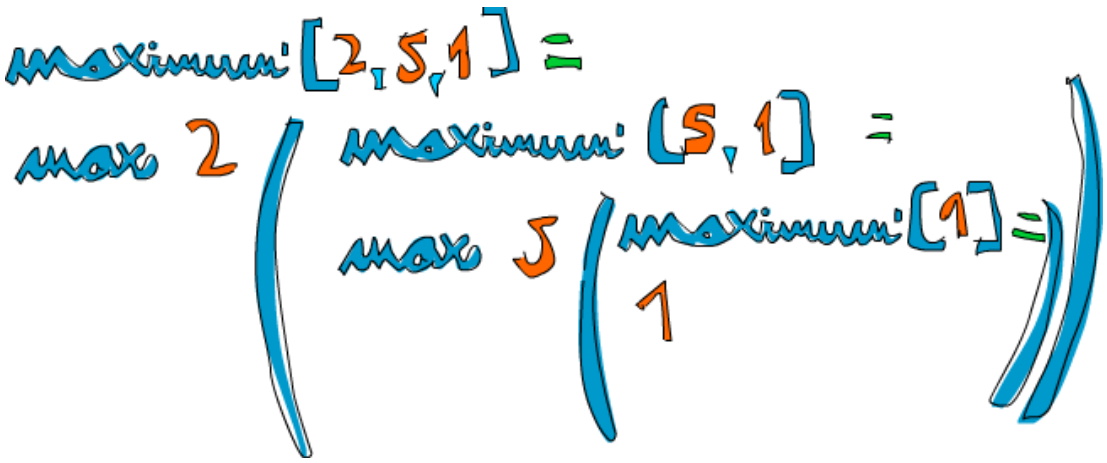
Третє означення виконує всю роботу. Ми зіставляємо із взірцем для розвалення списку на голову та хвіст. В рекурсивній обробці списків — це дуже поширена ідіома, тому звикайте до неї. Ми використовуємо зв'язку `where` для означування `maxTail` — покладаємо його рівним максимуму хвоста. Потім перевіряємо, чи значення голови більше ніж максимум хвоста. Якщо так, повертаємо голову списку. В іншому випадку — повертаємо максимум хвоста.

До прикладу, розгляньмо список чисел `[2,5,1]` і перевіримо роботу нашої функції. Якщо ми застосуємо `maximum'` до цього списку, перші два взірці не зіставляться. А от третій — так, і тому наш список розділиться на `2` і `[5,1]`. `where` захоче знайти максимум `[5,1]`, і тому виконання піде цим шляхом. `[5,1]` знову зіставиться лише із третім взірцем, і `[5,1]` буде поділено на `5` та `[1]`. Знову ж таки, зв'язка `where` захоче знайти максимум `[1]`. Тут вперше «вистрілює» гранична умова, і як максимум повертається `1`. Нарешті! Тепер ми робимо один крок нагору і порівнюємо `5` та максимум `[1]` (тобто, `1`), і ми звичайно ж отримуємо `5`. Отже, тепер ми знаємо, що максимум `[5,1]` дорівнює `5`. Йдемо вгору ще на один крок, де ми працювали із `2` і `[5,1]`. Порівнюючи `2` із максимумом `[5,1]` (який, як виявилося є `5`), ми вибираємо `5`.

А тепер іще більш зрозумілий варіант написання цієї функції — із застосуванням `max`. Якщо ви пам'ятаєте, `max` — це функція, яка приймає два значення і повертає більше з них. Ось як ми можемо переписати `maximum'` із використанням `max`:

```
maximum' :: Ord a => [a] -> a
maximum' []     = error "maximum of empty list"
maximum' [x]    = x
maximum' (x:xs) = max x (maximum' xs)
```

Елегантно, чи не так?! По суті, максимум списку — це `max` першого елементу і `maximum'` хвоста.



5.3 Ще трохи рекурсивних функцій

Тепер, коли ми знаємо, як думати рекурсивно, напишімо декілька функцій із використанням рекурсії. Перш за все, ми реалізуємо `replicate`. `replicate` приймає `Int` і деякий елемент, та повертає список, що містить кілька повторень того елементу. Наприклад, `replicate 3 5` повертає `[5,5,5]`. Подумаймо про граничні умови. Гадаю, що гранична умова то є нуль або якесь значення менше за нуль. Якщо ми спробуємо повторити щось нуль разів, ми маємо отримати порожній список. Це виконується також і для від'ємних чисел, тому що копіювати елемент від'ємну кількість разів не має сенсу.

```
replicate' :: (Num i, Ord i) => i -> a -> [a]
replicate' n x
    | n <= 0    = []
    | otherwise = x:replicate' (n-1) x
```

Тут ми використовуємо вартових замість взірців, оскільки ми перевіряємо булеву умову. Якщо `n` менше або рівне 0, повертаємо порожній список. В іншому випадку повертаємо список, де `x` стоїть в голові, а хвостом є список, де `x` повторене n-1 разів. Зрештою, `(n-1)` частина зробить виклик, в якому спрацює гранична умова.

Примітка: `Num` не є підкласом `Ord`. Це означає, що сутність, яка є числом, не обов'язково повинна мати впорядкування. Ось чому ми маємо накладати дві умови типокласів, — `Num` та `Ord`, коли додаємо чи віднімаємо і, водночас, порівнюємо.

Йдемо далі — реалізуємо `take`. `take` бере якусь кількість елементів з початку списку. Наприклад, `take 3 [5,4,3,2,1]` поверне `[5,4,3]`. Якщо ми спробуємо взяти 0 або меншу кількість елементів зі списку, отримаємо порожній список. Крім того, якщо ми спробуємо взяти будь-що з порожнього списку, ми отримаємо порожній список. Зверніть увагу — маємо дві граничні умови, навіть не напружившись. Розпишімо їх:

```
take' :: (Num i, Ord i) => i -> [a] -> [a]
take' n _
    | n <= 0   = []
take' _ []     = []
take' n (x:xs) = x : take' (n-1) xs
```

Перший взірець стверджує: якщо ми спробуємо взяти 0 або від'ємне число елементів, отримаємо порожній список. Зверніть увагу, що ми використовуємо `_` для зіставлення зі списком — тому що нам байдуже *що* то є. Також зауважте, що ми використовуємо вартового, але без `otherwise` частини. Це означає, що коли `n` є більшим за 0, зіставляння «провалиться» до наступного взірця. Другий взірець каже: при спробі взяти щось із порожнього списку ми отримаємо порожній список. Третій взірець розбиває список на голову і хвіст. Тоді ми означуємо, що `n` перших елементів зі списку `(x:xs)` то є список, де `x` є головою, а хвіст складається з результату взяття `n-1` елементів з хвоста списку на вході — себто з `xs`. Спробуйте розписати на аркуші паперу, як виглядатиме процес обчислення результату, якщо ми спробуємо взяти, скажімо, 3 з `[4,3,2,1]`.

`reverse` просто розвертає список. Подумайте про граничні умови. Які во-

ни? Що там думати — це порожній список! Розвернений порожній список дорівнює порожньому спискові (самому собі). Гаразд. А як щодо решти варіантів? Ну, ви можете сказати, що, розділивши список на голову і хвіст, розвернення списку дорівнюватиме розверненню хвоста із додаванням голови в кінці.

```
reverse' :: [a] -> [a]
reverse' [] = []
reverse' (x:xs) = reverse' xs ++ [x]
```

Ось так!

Оскільки Хаскел підтримує нескінченні списки, наша рекурсія не обов'язково повинна мати граничну умову. Проте, якщо граничних умов немає, рекурсія буде або ковбасити щось нескінченно довго, або побудує нескінченну структуру даних, як, наприклад, нескінченний список. Однак, приємність роботи із нескінченними списками полягає в тому, що ми можемо обрізати їх там, де забажаємо. `repeat` приймає елемент і повертає нескінченний список, який містить тільки цей елемент. Рекурсивна реалізація є доволі простою, — дивіться:

```
repeat' :: a -> [a]
repeat' x = x:repeat' x
```

Виклик `repeat 3` поверне список, який починається з `3` та містить нескінченну кількість трійок у хвості. Тому виклик `repeat 3` буде пораховано як `3:repeat 3`, що є `3:(3:repeat 3)`, що, в свою чергу, є `3:(3:(3:repeat 3))`, і так далі. `repeat 3` ніколи не завершить роботу, проте — `take 5 (repeat 3)` поверне список з п'яти трійок. По суті, це те саме що і `replicate 5 3`.

`zip` бере два списки і «застібає» їх разом отак: `zip [1,2,3] [2,3]` повертає `[(1,2),(2,3)]`. Якщо довжина списків на вході різна, `zip` обрізає довший список до довжини коротшого. Що ми отримаємо, `zip`-нувши щось із порожнім списком? Порожній список. Це і буде нашою граничною умовою. Проте `zip` отримує як параметри два списки, тому насправді існує дві граничні умови:

```
zip' :: [a] -> [b] -> [(a,b)]
zip' _ [] = []
zip' [] _ = []
zip' (x:xs) (y:ys) = (x,y):zip' xs ys
```

Перші два взірці кажуть, що, якщо перший або другий список порожні, ми отримаємо порожній список. Згідно з третім взірцем, два списки дорівнюють паруванню голів цих списків та застібанню їх хвостів. Застібання `[1,2,3]` та `['a','b']` в якийсь момент спробує поєднати `[3]` з `[]`. Тоді спрацю-

ють граничні умови і тому ми отримаємо `(1,'a'):(2,'b'):[]`, що є тотожнім `[(1,'a'),(2,'b')]`.

Реалізуймо ще одну стандартну функцію — `elem`. Ця функція приймає деякий елемент і список та перевіряє, чи містить даний список цей елемент. Граничною умовою, як і в більшості випадків при роботі зі списками, є порожній список. Ми знаємо, що порожній список (чисел, звірів, непрочитаних книжок і так далі.) не містить елементів взагалі, тому ми гарантовано не знайдемо там нічого, *що* б ми не шукали.

```
elem' :: Eq a => a -> [a] -> Bool
elem' a [] = False
elem' a (x:xs)
    | a == x    = True
    | otherwise = a `elem'` xs
```

Досить просто і очікувано. Якщо голова списку не дорівнює нашому елементові, то ми перевіряємо хвіст. Якщо ми досягнемо порожнього списку, то результатом буде `False`.

5.4 Швидко, відсортуйсь!

У нас є список з елементів, які можуть бути відсортовані. Їх тип є втіленням типокласу `Ord`. І тепер, ми хочемо відсортувати їх! Існує дуже класний алгоритм сортування, який називається швидким сортуванням. Це доволі нетривіальний спосіб сортування елементів. Імперативні реалізації зазвичай потребують більше десяти рядків коду, а от на Хаскелі реалізація є елегантною та набагато коротшою. Алгоритм швидкого сортування став своєрідною візитною карткою Хаскела. Тому реалізуймо цей алгоритм і тут, хоча, варто зазначити, що це заняття є дещо показовим, а реалізація стала певним кліше, адже всі її використовують для демонстрації елегантності Хаскела.

Отже, почнімо з сиґнатури типу — маємо `quicksort :: Ord a => [a] -> [a]`. Жодних сюрпризів! Граничні умови? Як і очікувалося — порожній список. Відсортований порожній список — це порожній список. «На пальцях» алгоритм виглядає отак: **відсортований список є списком, де спочатку ідуть елементи менші (або рівні) за голову списку на вході (і ці елементи є відсортованими), потім власне голова, а потім — елементи, які за значенням більші за голову (ці елементи теж є відсортовани-**

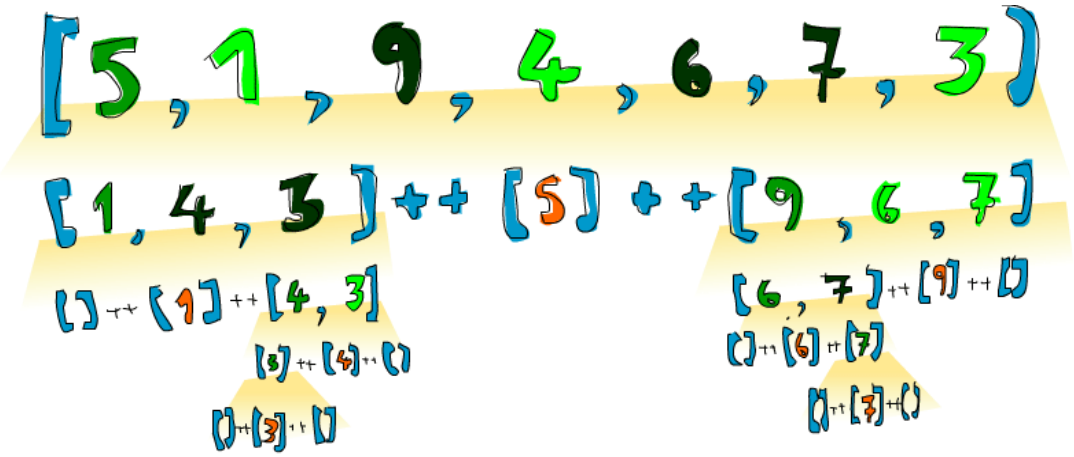
ми). Зверніть увагу, ми використали «є відсортованими» двічі в цьому означенні, тому швидше за все доведеться рекурсивно викликати себе двічі! Зверніть також увагу, що у означенні нашого алгоритму ми використали дієслово *є* замість імперативної мантри *виконайте спочатку це, потім оце, і згодом зробіть оте...* В цьому і є краса функційного програмування! Як будемо фільтрувати список, щоб отримати лише елементи менші за голову, та елементи, які є більші за голову? Спискові характери. Отож, до справи — ось як виглядатиме означення цієї функції:

```
quicksort :: Ord a => [a] -> [a]
quicksort [] = []
quicksort (x:xs) =
    let smallerSorted = quicksort [a | a <- xs, a <= x]
        biggerSorted  = quicksort [a | a <- xs, a > x]
    in  smallerSorted ++ [x] ++ biggerSorted
```

Трохи потестуймо наш алгоритм, щоб переконатися в його коректності.

```
ghci> quicksort [10,2,5,3,1,6,7,4,2,3,4,8,9]
[1,2,2,3,3,4,4,5,6,7,8,9,10]
ghci> quicksort "the quick brown fox jumps over the lazy dog"
"        abcdeeefghhijklmnoooopqrrsttuuvwxyz"
```

Нарешті! Те що треба! Скажімо ми хочемо відсортувати `[5,1,9,4,6,7,3]`. Цей алгоритм спочатку візьме голову, яка є `5`, і розташує її всередині двох списків. До першого потрапляють елементи менші за голову (або такі самі як голова), а до другого — більші. Отже, у певний момент матимемо `[1,4,3] ++ [5] ++ [9,6,7]`. Тепер відомо, що у повністю відсортованому списку `5` стоятиме на четвертому місці, оскільки ми маємо 3 числа менші за 5 та 3 числа більші за 5. Далі, якщо ми відсортуємо `[1,4,3]` і `[9,6,7]`, ми отримаємо повністю відсортований список! Ми сортуємо ці два підсписки, використовуючи цю ж саму функцію. Таким чином, ми розбиватимемо підсписки на дедалі коротші підпідсписки аж доки не отримаємо порожні списки, а порожні списки вже й так відсортовані. Ось ілюстрація:



Елементи, які потрапили на свої місця і не будуть змінювати позиції, позначено **помаранчевим кольором**. Якщо ви прочитаєте їх зліва направо, ви побачите відсортований список. Хоча ми вирішили порівнювати елементи з хвостів із головами, ми могли б використовувати будь-який елемент замість голови у порівнянні. В швидкому сортуванні, елемент, із яким порівнюються решта елементів, називається стрижнем. Стрижні тут позначено **зеленим**. Ми вибрали голову, тому що її легко отримати, використовуючи зіставлення із взірцем. Елементи, які є менші за стрижень — **світло-зелені**, а елементи більші за стрижень — **темно-зелені**. Градієнтні жовтуватощі тут позначають застосування швидкого сортування.

5.5 Думаючи рекурсивно

Ми доволі добре попрацювали із рекурсією, і як ви, напевно, помітили, тут є певна ідіома. Зазвичай ви спочатку означуєте граничний випадок, а потім — функцію, яка щось робить з якимось елементом та результатом застосування цієї функції до решти елементів. Не має значення, *що* це — список, дерево або будь-яка інша структура даних. Сума то є перший елемент списку плюс сума решти списку. Добуток елементів списку — перший елемент списку, помножений на добуток решти елементів списку. Довжина списку то є одиниця плюс довжина хвоста списку. І так далі, і так далі...

Звісно, приклади, наведені вище, всі мають якісь граничні випадки. Зазвичай, граничні випадки — це деякі сценарії, в яких застосування рекурсії не має сенсу. При роботі із списками граничним випадком найчастіше є порожній список. Якщо ви маєте справу з деревами, граничний випа-

док — це зазвичай вузол, який не має дітей.

Схожа ситуація і при рекурсивній роботі із числами. Як правило, йдеться про деяке число і функцію, аргументом якої є це число, але трохи модифіковане. Ми працювали із функцію факторіалу, яку було означено як добуток числа та факторіалу цього числа мінус один. Таке рекурсивне означення не має сенсу, якщо число дорівнює нулю, оскільки факторіал існує лише для додатних чисел. Досить часто значенням з граничного випадку виявляється нейтральний елемент. Нейтральним елементом для операції множення є 1, оскільки, помноживши будь-що на 1, ви отримаєте те будь-що без змін. Також, коли ми обчислюємо суму списків, ми означуємо суму порожнього списку як 0, і 0 тут — це нейтральний елемент операції додавання. В алгоритмі швидкого сортування граничним випадком є порожній список й нейтральним елементом також є порожній список, тому що при додаванні порожнього списку до будь-якого списку, ви просто отримаєте назад початковий список без змін.

Підсумовуючи: аби знайти рекурсивний розв'язок задачі, подумайте спочатку про випадки, де рекурсія не має сенсу, і спробуйте використати їх як граничні умови; потім розгляньте нейтральні елементи; наостанок, подумайте, як краще розбити параметри функції (наприклад, списки-параметри зазвичай розбиваються на голову і хвіст за допомогою зіставлення із взірцем) і до якої частини того розбиття ви будете застосовувати рекурсивний виклик.

Розділ 6

Функції вищого порядку

Переклад українською Ганни Лелів

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

6.1 Карійовані функції

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

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

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

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

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

Погляньте на цю сміховинно просту функцію:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

6.2 Із вищим порядком усе в порядку

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

6.3 Відображення і фільтри

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

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

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

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

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

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

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

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

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

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

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

Пам'ятаєте функцію швидкого сортування з розділу 5? Ми скористалися списковими характерами, щоб відфільтрувати елементи списку менші (чи рівні) за значенням або більші за стрижень. Те ж саме можна реалізувати і за допомогою `filter`, і результат буде набагато читабельніший:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

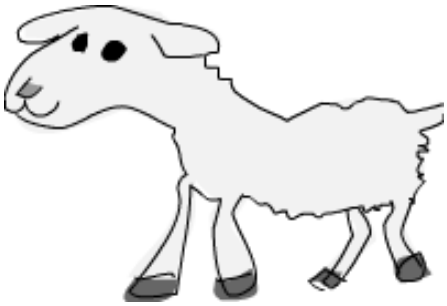
6.4 Лямбди

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

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

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

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



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

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

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

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

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

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

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

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

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

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

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

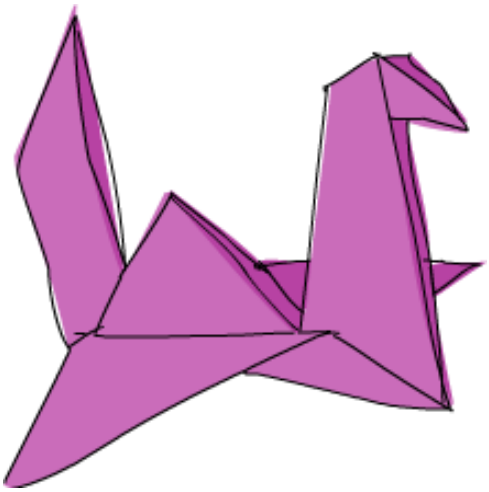
```
flip' :: (a -> b -> c) -> b -> a -> c
flip' f = \x y -> f y x
```

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

6.5 Згортком і батька легше бити

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

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



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

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

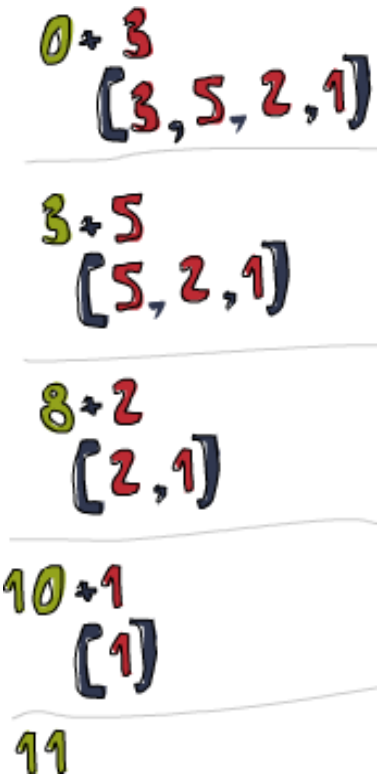
```
sum' :: Num a => [a] -> a
```

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

Тестуємо, один-два-три:

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

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



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

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

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

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

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

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

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

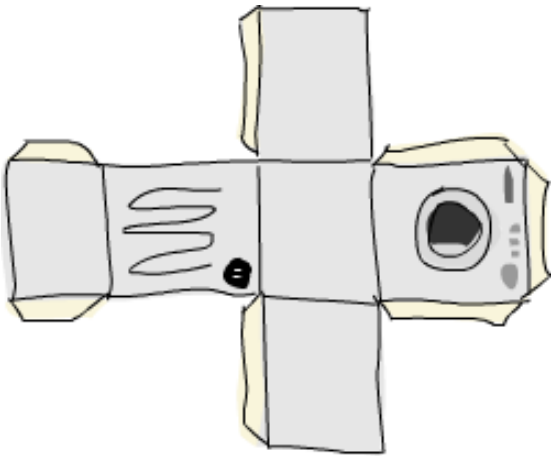
Правий згорток, `foldr`, працює подібно до лівого згортка, тільки накопичувач поїдає значення з правого боку. На додачу, бінарна функція лівого згортка бере накопичувач як перший параметр, а поточне значення — як другий параметр (отож `\acc x -> ...`), а бінарна функція правого згортка має поточне значення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

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

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

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

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



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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

6.6 Доларове застосування функцій

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

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

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

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

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

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

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

6.7 Композиція функцій

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

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

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

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

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

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

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

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

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

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

на

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

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

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

можна записати як

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

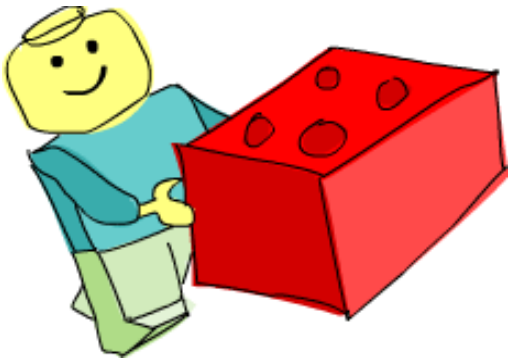
[†] «Гольф-програмування» — програмування для розваги, де метрикою успіху, окрім коректності, є довжина програми. Метою гри є написання найкоротшої програми, яка втілює заданий алгоритм.

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»).

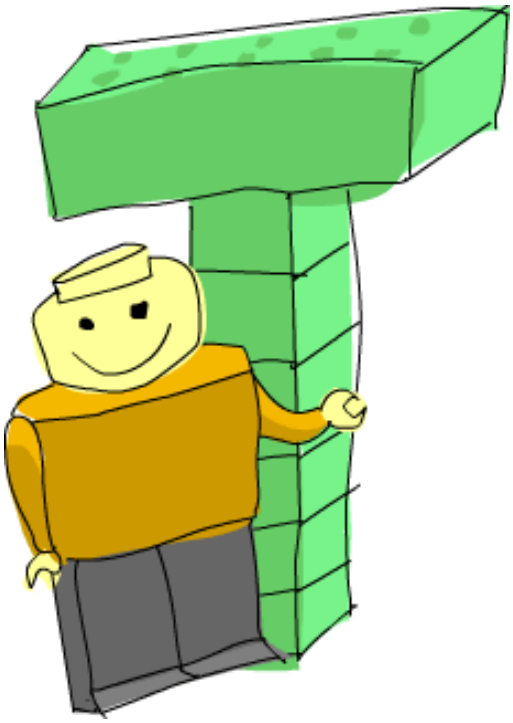
Списки теж можна порівнювати, і таке порівняння відбувається лексикографічно. А що ж робити, коли треба відсортувати список списків, але не за

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.

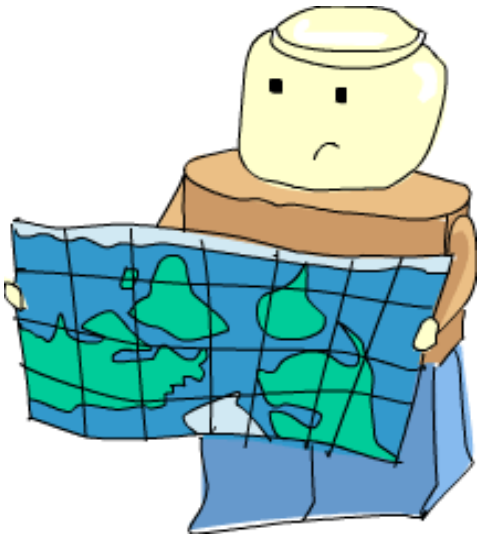
Це стандартна рекурсивна функція, що працює із списком. Крайовий випадок, розбиття списку на голову та хвіст, рекурсивні виклики — всі тут родичі

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.

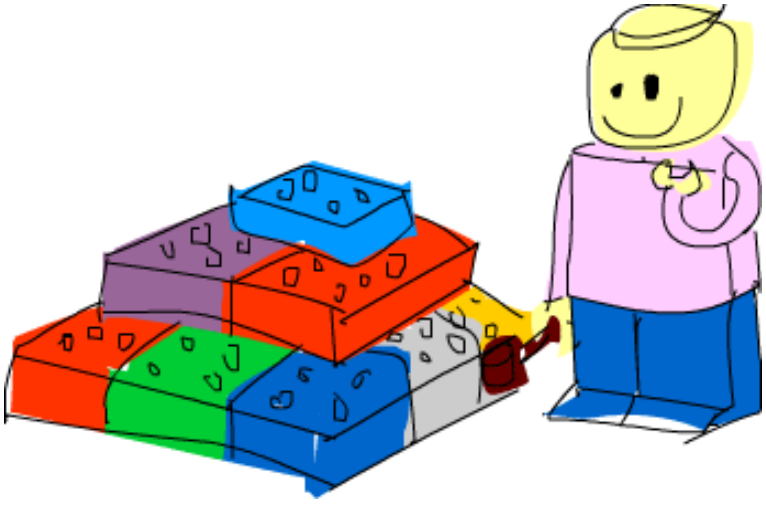
```
ghci> Set.filter odd $ Set.fromList [3,4,5,6,7,2,3,4]
fromList [3,5,7]
ghci> Set.map (+1) $ Set.fromList [3,4,5,6,7,2,3,4]
fromList [3,4,5,6,7,8]
```

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.

Розділ 8

Побудова власних типів і типокласів

Переклад українською Богдана Пеньковського

В попередніх розділах ми розглянули деякі вже наявні в мові Хаскел типи і типокласи, а в цьому ми навчимося створювати нові типи і типокласи та залучати їх до роботи!

8.1 Вступ до алгебраїчних типів даних

Нам вже траплялося багацько різних типів даних — `Bool`, `Int`, `Char`, `Maybe` тощо. Але як створити новий тип? Що ж, один спосіб — це використати ключове слово **data** для означення типу. Погляньмо, як тип `Bool` означено в стандартній бібліотеці.

```
data Bool = False | True
```

`data` означає, що ми означуємо новий тип даних. Частина, що передує `=` — це ім'я нового типу, і тут ім'я є `Bool`. Речі, які подано після `=`, називаються **конструкторами значень**, і кожен конструктор відповідає значенню, яке цей тип може мати. Знак `|` читається як *або*. Таким чином, ми можемо прочитати це означення отак: тип `Bool` може мати значення `True` або значення `False`. Ім'я типу і конструктори значень повинні починатися із великої літери.

Подібно до означення `Bool`, можна собі уявити, що `Int` означено отак:

```
data Int =
    -2147483648 | -2147483647 | ... | -1 | 0 | 1 | 2 | ... | 2147483647
```

Перший і останній конструктори значень є мінімальним і максимальним значеннями типу `Int`, відповідно. Насправді `Int` означено по-іншому, а три

крапки тут є тому, що ми викинули з означення цілу купу чисел. Це означення — не реальний код, а лише ілюстрація.

А зараз подумаймо про те, яким чином можна **представити** [to represent] геометричну фігуру мовою Хаскел. Наприклад, можна було б використати кортеж. Коло можна представити як `(43.1, 55.0, 10.4)`, де перше і друге числа є координатами центра кола, а третє — радіусом. Наче непогано, але ці числа можна переінтерпретувати як завгодно (наприклад, як тривимірний вектор). Кращий підхід — створити новий тип даних для представлення фігури. Хай фігура може бути або колом, або прямокутником. Маємо:

```
data Shape = Circle Float Float Float | Rectangle Float Float Float Float
```

Що це таке? Міркуватимемо так. Конструктор значень `Circle` має три **поля** [fields], кожне з яких бере плавомку (`Float`). Отже, означуючи конструктор значень, ми можемо, за бажанням, додати декілька типів після імені того конструктора, і ці типи означуватимуть значення, які «міститиме» цей конструктор. Тут перші два поля є координатами центру, а третє — радіусом. Конструктор значень прямокутника, `Rectangle`, має чотири поля, кожне з яких також приймає плавомку. Перші два є координатами лівого верхнього кута, а два останні — правого нижнього.

Коли я кажу «поля», я насправді маю на увазі параметри. Конструктори значень для якогось типу є звичайними функціями, які повертають значення такого типу. Розгляньмо сиґнатури типів цих двох конструкторів значень.

```
ghci> :t Circle
Circle :: Float -> Float -> Float -> Shape
ghci> :t Rectangle
Rectangle :: Float -> Float -> Float -> Float -> Shape
```

Чудово, конструктори значень є функціями, як і все інше. Хто б, скажи, подумав? А тепер запишемо функцію, яка приймає фігуру й повертає площу її

поверхні.

```
surface :: Shape -> Float
surface (Circle _ _ r)          = pi * r ^ 2
surface (Rectangle x1 y1 x2 y2) = (abs $ x2 - x1) * (abs $ y2 - y1)
```

Першим гідним уваги тут є оголошення типу. Воно каже: функція отримує фігуру і повертає плавомку. Ми не можемо записати оголошення типу як `Circle -> Float`, тому що `Circle` не є типом — ним є `Shape`. Так само, як ми не можемо написати функцію з оголошеннями типу `True -> Int`. Варто також звернути увагу на таке: ми можемо використовувати конструктори у взірцях. Ми стикалися з цим і раніше (весь час, якщо чесно), коли записували `[]`, або `False`, або `5`, однак ці конструктори не мали жодних полів. Отже, беремо і просто записуємо ім'я конструктора, а потім зв'язуємо його поля з іменами. Оскільки нас цікавить радіус, ми не дуже переймаємося першими двома полями, які нам кажуть, де знаходиться коло.

```
ghci> surface $ Circle 10 20 10
314.15927
ghci> surface $ Rectangle 0 0 100 100
10000.0
```

Отак — працює! Але якщо ми спробуємо просто надрукувати `Circle 10 20 5` в командному рядку, отримаємо помилку. Це тому, що Хаскел не знає, як серіалізувати в рядок наш новий тип даних (поки що). Пам'ятайте: коли ми намагаємося надрукувати значення в командному рядку, Хаскел спочатку запускає функцію `show`, щоб отримати рядкове представлення цього значення, а вже потім виводить його в терміналі. Щоб зробити наш тип `Shape` членом типокласу `Show`, ми переозначимо його ось як:

```
data Shape = Circle Float Float Float
           | Rectangle Float Float Float Float deriving (Show)
```

Поки що ми не будемо особливо турбуватися про кінцівку цього означення. Просто скажемо що, якщо додати `deriving (Show)` в кінець означення *data*, Хаскел автомагічно зробить цей тип членом типокласу `Show`. І тепер ми можемо виконати таке:

```
ghci> Circle 10 20 5
Circle 10.0 20.0 5.0
ghci> Rectangle 50 230 60 90
Rectangle 50.0 230.0 60.0 90.0
```

Конструктори значень є функціями, тому ми можемо відображати за

їхньою допомогою, частково їх застосовувати, і взагалі робити все що завгодно. Знадобився список концентричних кіл із різними радіусами — легко:

```
ghci> map (Circle 10 20) [4,5,6,7]
[Circle 10.0 20.0 4.0
,Circle 10.0 20.0 5.0
,Circle 10.0 20.0 6.0
,Circle 10.0 20.0 7.0]
```

Наш тип даних хороший, але міг би бути й ще кращим. Запишімо проміжний тип даних, який описує точку у двовимірному просторі. Тоді ми зможемо використати його і зробити наші представлення фігур більш зрозумілими.

```
data Point = Point Float Float deriving (Show)
data Shape = Circle Point Float | Rectangle Point Point deriving (Show)
```

Зауважимо: означаючи точку, ми назвали тип даних та конструктор значень однаково — `Point`. Тут немає ніякого прихованого змісту, просто, як правило, конструктору надають ім'я таке ж, як і для типу, коли є лише один конструктор значень. Тепер `Circle` має два поля, одне з яких має тип `Point`, а інше — тип `Float`. Це полегшує розуміння в плані «що є що». Те ж стосується і прямокутника. Тепер нам потрібно змінити нашу функцію `surface`, щоб вона могла працювати із новими типами.

```
surface :: Shape -> Float
surface (Circle _ r) = pi * r ^ 2
surface (Rectangle (Point x1 y1) (Point x2 y2)) =
    (abs $ x2 - x1) * (abs $ y2 - y1)
```

Єдине, що нам довелося змінити, це — взірці. Ми повністю знехтували точкою у взірці для кола. У взірці для прямокутника ми просто застосували вкладене зіставлення із взірцем, щоб дістати поля точок. Якби нам було треба також поіменувати точки в цілому, можна було б також застосувати взірці з іменами.

```
ghci> surface (Rectangle (Point 0 0) (Point 100 100))
10000.0
ghci> surface (Circle (Point 0 0) 24)
1809.5574
```

Як щодо функції, яка соватиме фігуру? Вона приймає фігуру, величину, на яку її треба зсунути по осі абсцис, величину зсуву по осі ординат, і повертає нову фігуру із тими ж самими розмірами, от тільки розташовану десь в іншому місці.

```
nudge :: Shape -> Float -> Float -> Shape
nudge (Circle (Point x y) r) a b = Circle (Point (x+a) (y+b)) r
```

```
nudge (Rectangle (Point x1 y1) (Point x2 y2)) a b =
    Rectangle (Point (x1+a) (y1+b)) (Point (x2+a) (y2+b))
```

Досить просто. Для кожного типу фігури ми додаємо зсув до точки, яка описує її місце розташування.

```
ghci> nudge (Circle (Point 34 34) 10) 5 10
Circle (Point 39.0 44.0) 10.0
```

Якщо ми не хочемо мати справу безпосередньо з точками, ми можемо написати кілька допоміжних функцій, які спочатку створюватимуть фігури якогось розміру в початку координат, а вже потім зсуватимуть їх куди треба.

```
baseCircle :: Float -> Shape
baseCircle r = Circle (Point 0 0) r

baseRect :: Float -> Float -> Shape
baseRect width height = Rectangle (Point 0 0) (Point width height)
```

```
ghci> nudge (baseRect 40 100) 60 23
Rectangle (Point 60.0 23.0) (Point 100.0 123.0)
```

Звісно, у модулях ви можете експортувати не тільки функції, а й типи даних. Для цього достатньо записати ваш тип разом із іменами функцій, які ви експортуєте, а потім додати дужки. В дужках треба вказати розділені комами конструктори значень, які ви хочете експортувати. Якщо ви хочете експортувати всі конструктори значень якогось типу, просто пишіть в дужках дві крапки (себто, `..`).

Якщо б ми хотіли експортувати функції і типи, які ми означили в даному розділі, ми могли б почати з цього:

```
module Shapes
( Point(..)
, Shape(..)
, surface
, nudge
, baseCircle
, baseRect
) where
```

За допомогою `Shape(..)` ми експортували всі конструктори значень для типу `Shape`. Отже, кожен, хто імпортує наш модуль, може будувати фігури, використовуючи конструктори значень `Rectangle` та `Circle`. Це те ж саме, що і `Shape (Rectangle, Circle)`.

Також ми могли б і не експортувати жодних конструкторів значень для `Shape` взагалі, просто записавши `Shape` в інструкції експорту. У такий спосіб, той, хто імпортує наш модуль, зможе створювати фігури тільки за допомогою допоміжних функцій `baseCircle` та `baseRect`. Модуль `Data.Map` використовує цей підхід. Ви не можете створити мапу, виконавши `Map.Map [(1,2),(3,4)]`, тому що відповідний конструктор значень не експортований. Проте ви можете скористатись однією із допоміжних функцій на кшталт `Map.fromList`. Пам'ятаймо, конструктори значень є просто функціями, які отримують поля як параметри й повертають значення відповідного типу (як-от `Shape`) як результат. Тому, коли ми вирішуємо їх не експортувати, ми запобігаємо використанню цих функцій тими, хто імпортує наш модуль. Якщо конструктори значень не експортуються, але експортуються якісь інші функції, які повертають значення потрібного типу, ми можемо використати ці функції для побудови значень.

Типи даних, для яких конструктори не експортовано, є більш абстрактними, бо їхню реалізацію приховано. До того ж, використання конструкторів значень у зіставленнях із взірцем стає неможливим.

8.2 Синтаксис для записів

Нехай нам дали завдання створити тип даних, який описуватиме особу. Інформація, яку б ми хотіли зберігати, отака: ім'я, прізвище, вік, зріст, номер телефону і улюблений вид морозива. Не знаю як вам, але це все, що мені треба знати про людину. Поїхали!

```
data Person = Person String String Int Float String String deriving (Show)
```

Гаразд. Першим полем є ім'я, наступним — прізвище, третім йде вік і так далі. Для прикладу, створімо особу:

```
ghci> let guy = Person "Buddy" "Finklestein" 43 184.2 "526-2928" "Chocolate"
ghci> guy
Person "Buddy" "Finklestein" 43 184.2 "526-2928" "Chocolate"
```

Непогано, але читається нелегко. Що, якби ми хотіли записати функцію, яка б повертала лише якусь окрему частину інформації про особу? Функцію, яка повертає ім'я, функцію, яка повертає прізвище, і так далі. Що ж, тоді б нам довелося означити їх приблизно так:

```
firstName :: Person -> String
firstName (Person firstname _ _ _ _ _) = firstname

lastName :: Person -> String
lastName (Person _ lastname _ _ _ _) = lastname

age :: Person -> Int
age (Person _ _ age _ _ _) = age

height :: Person -> Float
height (Person _ _ _ height _ _) = height

phoneNumber :: Person -> String
phoneNumber (Person _ _ _ _ number _) = number

flavor :: Person -> String
flavor (Person _ _ _ _ _ flavor) = flavor
```

Фух. Написав, але без жодного задоволення! Втім, незважаючи на незграбність цього розв'язку і НУДЬГУ, яку відчуваєш, коли його записуєш, він працює.

```
ghci> let guy = Person "Buddy" "Finklestein" 43 184.2 "526-2928" "Chocolate"
ghci> firstName guy
"Buddy"
ghci> height guy
184.2
ghci> flavor guy
"Chocolate"
```

Ви скажете, що має бути кращий спосіб. Ні — його не існує, вибачте.

Жартую, є. Ха-ха-ха! Творці Хаскела були дуже розумними й передбачили такий хід подій. Вони надали альтернативний спосіб означування таких типів даних. Ось як ми могли б отримати еквівалентний **функціонал** [functionality] за допомогою записів.

```
data Person = Person { firstName :: String
                     , lastName :: String
                     , age :: Int
                     , height :: Float
                     , phoneNumber :: String
                     , flavor :: String
                     } deriving (Show)
```

Тому замість простого оголошення типів полів, одного за одним, розділених пробілами, ми користуємося фігурними дужками, а усередині, для кожного поля, спочатку пишемо ім'я поля, наприклад, `firstName`, потім ставимо подвійну двокрапку `::` (вона також зветься Paamayim Nekudotayim[†], хе-хе), а потім вказуємо тип. В результаті маємо тип даних, повністю аналогічний попередньому. Головною перевагою цього підходу є те, що функції для отримання значень полів створюються автоматично. Через те, що ми використали запис, Хаскел нам автоматично створив отакі функції: `firstName`, `lastName`, `age`, `height`, `phoneNumber` та `flavor`.

```
ghci> :t flavor
flavor :: Person -> String
ghci> :t firstName
firstName :: Person -> String
```

Існує ще одна перевага у використанні записів. Коли **ми автовтілюємо** `Show` [we derive `Show` instance], тип серіалізуватиметься в рядок по-іншому, якщо ми означимо його за допомогою синтаксису для записів. Хай в нас є тип, який **представляє** [represents] автомобіль. Ми хочемо бути в курсі, яка компанія його виготовила, яка назва моделі та який рік виробництва. Дивіться:

```
data Car = Car String String Int deriving (Show)
```

```
ghci> Car "Ford" "Mustang" 1967
Car "Ford" "Mustang" 1967
```

Якщо ми переозначимо цей тип, використовуючи запис, ми можемо побудувати авто ось так:

```
data Car = Car {company :: String, model :: String, year :: Int} deriving (Show)
```

```
ghci> Car {company="Ford", model="Mustang", year=1967}
Car {company = "Ford", model = "Mustang", year = 1967}
```

Створюючи нове авто, не обов'язково подавати поля в правильному порядку — головне, щоб вони всі були подані. Але якщо ми не використовуємо синтаксис для записів в означенні типу, при створенні значення цього типу значення-поля мають бути подані в тому порядку, в якому вони подані в означенні цього типу.

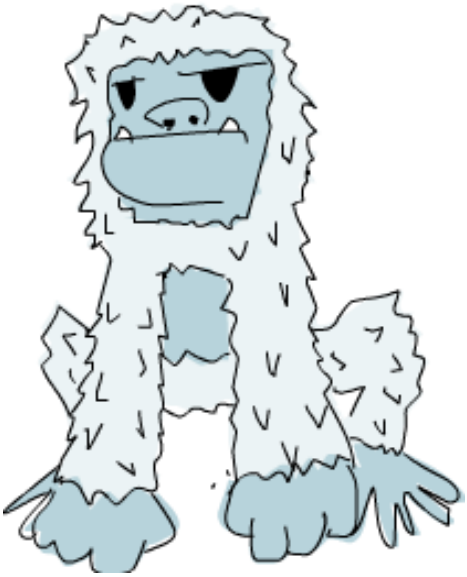
[†] З івриту: נקודתיים פעמיים, вимовляється як [paʔaˈmajim nəkudoˈtajim], і означає «подвійна двокрапка». Ця назва ніде не використовується, окрім як в Zend Engine 0.5 з PHP 3, який розроблено в Ізраїлі і де можна зустріти наповнені змістом повідомлення про помилку на кшталт «Parse error: syntax error, unexpected T_PAAMAYIM_NEKUDOTAYIM» :-). В PHP подвійна двокрапка є оператором визначення зони видимості.

Використовуйте записи, коли конструктор має кілька полів і не є очевидним «що є що». Якщо ми означимо тип даних для тривимірних векторів як `data Vector3D = Vector Int Int Int`, з цього означення легко здогадатися, що поля швидше за все є елементами вектора. Проте в наших типах `Person` та `Car` семантичне навантаження полів сяє не настільки яскраво, і ми значно виграємо від використання синтаксису для записів в цих означеннях.

8.3 Параметри типів

Конструктор значень може брати декілька значень-параметрів і повертає нове значення. Наприклад, конструктор `Car` приймає три значення і будує значення типу `Car`. За аналогією, **конструктори типів** також можуть приймати інші типи як параметри і будувати нові типи. Це може спочатку звучати занадто «мета», але насправді це не складно. Якщо вам знайомі **шаблони C++** [templates in C++], то ви помітите, що вони схожі на конструктори типів. Для того, щоб розібратися як працюють параметри типів, розгляньмо реалізацію одного з типів, який нам вже зустрічався:

```
data Maybe a = Nothing | Just a
```



Тут `a` є параметром типу. І через те, що параметр типу є, ми називаємо `Maybe` конструктором типу. Залежно від того, *що* ми хочемо щоб цей тип містив (коли він не є `Nothing`), цей конструктор типів може побудувати нам типи `Maybe Int`, `Maybe Car`, `Maybe String` і так далі. Значення не може мати тип просто `Maybe`, бо це не є тип — це є конструктор типів. Для того, щоб це був справжній тип (себто, можна створити значення, що матиме такий тип), всі його параметри повинні бути заповнені.

Таким чином, якщо ми передаємо `Char` як параметр до `Maybe`, ми отримуємо тип `Maybe Char`. До прикладу, значення `Just 'a'` має тип `Maybe Char`.

Ви могли цього й не знати, але ми вже користувалися типом, що має параметр, ще до того як ми познайомилися із `Maybe`. Цим типом є список! Хоча в грі задіяно певний синтаксичний цукор, тип-список приймає параметр аби утворити конкретний тип. Значення можуть мати тип `[Int]`, або `[Char]`, або `[[String]]`, але не може бути значення, яке має просто тип `[]`.

Повозімося в пісочниці із типом `Maybe`:

```
ghci> Just "Haha"
Just "Haha"
ghci> Just 84
Just 84
ghci> :t Just "Haha"
Just "Haha" :: Maybe [Char]
ghci> :t Just 84
Just 84 :: Num t => Maybe t
ghci> :t Nothing
Nothing :: Maybe a
ghci> Just 10 :: Maybe Double
Just 10.0
```

Параметри типів є корисними, бо вони уможливлюють створення різних типів залежно від того, *що* ми бажаємо розмістити «усередині» тих типів. Коли ми пишемо `:t Just "Haha"`, система виведення типів з'ясовує, що це має бути тип `Maybe [Char]`, адже `a` в `Just a` є рядком, тому `a` в `Maybe a` також має бути рядком.

Зауважте, що `Nothing` має тип `Maybe a`, і цей тип є поліморфним. Якщо деяка функція бере `Maybe Int` як параметр, ми можемо передати їй `Nothing`, бо `Nothing` не містить значення взагалі, тому не важливо, який у цього «відсутнього» значення тип. Тип `Maybe a` може поводитись як `Maybe Int` якщо треба, так само як `5` може поводитись як `Int` або як `Double`. Аналогічно, порожній список має тип `[a]` і може поводитись як порожній список чого завгодно. Саме тому ми можемо обчислювати `[1,2,3] ++ []` та `["ha","ha","ha"] ++ []`.

Параметризація типів — річ корисна, але тільки тоді, коли вона доречна. Зазвичай вона використовується, коли наш тип даних працюватиме незалежно від типу значення, яке він тримає в собі — подібно до `Maybe a`. Якщо наш тип поводиться як «коробка для чогось», то варто цей тип параметризувати. Ми могли б змінити наш тип даних `Car` із цього:

```
data Car = Car { company :: String
               , model :: String
               , year :: Int
               } deriving (Show)
```

на цей:

```
data Car a b c = Car { company :: a
                     , model :: b
                     , year :: c
                     } deriving (Show)
```

Але чи виграли б ми насправді? Відповідь: ймовірно, що ні, бо ми тільки-но означили функції, які працюють лише з типом `Car String String Int`. Наприклад, маючи наше перше означення для `Car`, ми могли б записати функцію, яка виводить на екран характеристики автомобіля гарненьким текстиком.

```
tellCar :: Car -> String
tellCar (Car {company = c, model = m, year = y}) =
    "This " ++ c ++ " " ++ m ++ " was made in " ++ show y
```

```
ghci> let stang = Car {company="Ford", model="Mustang", year=1967}
ghci> tellCar stang
"This Ford Mustang was made in 1967"
```

Симпатична невеличка функція! Оголошення типу — симпатичне, функція добре працює. А що, коли `Car` був би `Car a b c`?

```
tellCar :: Show a => Car String String a -> String
tellCar (Car {company = c, model = m, year = y}) =
    "This " ++ c ++ " " ++ m ++ " was made in " ++ show y
```

Ми змусили цю функцію приймати `Car` ось такого типу: `Show a => Car String String a`. Як бачите, сиґнатура типу є складнішою ніж попередня, а єдина користь, яку ми отримуємо у винагороду — це те, що тепер `c` може бути будь-яким типом, який втілює типоклас `Show`.

```
ghci> tellCar (Car "Ford" "Mustang" 1967)
"This Ford Mustang was made in 1967"
ghci> tellCar (Car "Ford" "Mustang" "nineteen sixty seven")
"This Ford Mustang was made in \"nineteen sixty seven\""
ghci> :t Car "Ford" "Mustang" 1967
Car "Ford" "Mustang" 1967 :: Num t => Car [Char] [Char] t
ghci> :t Car "Ford" "Mustang" "nineteen sixty seven"
Car "Ford" "Mustang" "nineteen sixty seven" :: Car [Char] [Char] [Char]
```

Але в реальності ми швидше за все користуватимемося `Car String String Int` у більшості випадків, тому схоже, що параметризувати тип `Car` не варто. Зазвичай ми використовуємо параметри типів, коли «внутрішні» типи, що їх містять у собі різні конструктори значень «зовнішнього» типу, не є важливими для роботи того «зовнішнього» типу. Список «чогось» є списком чогось і не має значення, яким є тип того чогось — список все одно може працювати. Якщо ми хочемо обчислити суму списку чисел, ми можемо, згодом, у функції сумування, вказати, що ми хочемо саме список *чисел*. Те ж стосується `Maybe`. Тип `Maybe` описує ситуацію, коли є можливість або не мати нічого (наприклад: немає результату обрахунку), або мати щось одне якогось типу (наприклад: є одна-єдина відповідь). Для роботи `Maybe` не має значення, яким є тип того «чогось».

Іншим прикладом параметризованого типу, який нам вже траплявся, є `Map k v` із модуля `Data.Map`. `k` є типом ключів у мапі і `v` є типом значень. Це є хороший приклад структури даних, де параметри типів є дуже корисними. Параметризовані мапи дозволяють нам описувати відображення з будь-якого типу в будь-який інший, за умови, що тип ключа належить до типокласу `Ord`. Якби ми взялися написати означення мапи, то в нас могло б виникнути бажання включити в означення даних цю умову типокласу, на кшталт:

```
data Ord k => Map k v = ...
```

Проте ми маємо дуже серйозну домовленість в Хаскелі: **ніколи не вказувати умови типокласів в означеннях даних**. Чому? А тому, що ми не отримуємо особливої користі від цього, а роботи це додає — змушує писати більше класових обмежень, навіть коли вони не потрібні. Є умова типокласу `Ord k` в *data*-означенні для `Map k v` чи немає — все одно доведеться писати цю умову в сиґнатурах типу функцій, яким потрібно, щоб ключі у мапі можна було впорядкувати. Зате, якщо ми не вкажемо цю умову в означенні, нам не потрібно буде писати `Ord k =>` в оголошеннях типу тих функцій, для яких можливість впорядкування ключів не є необхідністю. Прикладом такої функції є `toList`, яка лише бере мапу і перетворює її в асоціативний список. Її сиґнатурою типу є `toList :: Map k a -> [(k, a)]`. Якщо б `Map k v` мало б умову типокласу в *data*-означенні, тип `toList` мав би бути `toList :: Ord k => Map k a -> [(k, a)]`, хоча ця функція не робить порівнянь ключів взагалі.

Тому не варто записувати умови в *data*-означеннях, навіть коли здається, що в цьому є сенс, оскільки вам все одно доведеться їх записувати в оголоше-

ннях типу функцій.

Реалізуймо тривимірний вектор і деякі операції, що працюють із ним. Скористаймося параметризованим типом: хоча вектор зазвичай містить значення лише чисельних типів, чисельних типів є декілька, і тому параметризація доречна.

```
data Vector a = Vector a a a deriving (Show)

vplus :: Num t => Vector t -> Vector t -> Vector t
(Vector i j k) `vplus` (Vector l m n) = Vector (i+l) (j+m) (k+n)

vectMult :: Num t => Vector t -> t -> Vector t
(Vector i j k) `vectMult` m = Vector (i*m) (j*m) (k*m)

scalarMult :: Num t => Vector t -> Vector t -> t
(Vector i j k) `scalarMult` (Vector l m n) = i*l + j*m + k*n
```

`vplus` додає два вектори і це реалізовано простим додаванням відповідних компонент. Функція `scalarMult` — для обрахунку скалярного добутку двох векторів, а `vectMult` — для множення вектора на скаляр. Ці функції можуть оперувати `Vector Int`, `Vector Integer`, `Vector Float` і так далі, допоки виконується умова, що `a` із `Vector a` належить типокласу `Num`. Також, перевіривши оголошення типу для цих функцій, ви побачите, що вони можуть оперувати тільки векторами одного типу, а інші додаткові числа, що є залучені в деяких операціях, повинні бути того ж типу, що і числа, які містяться у векторах. Зверніть увагу — ми не додавали умову `Num` в *data*-означення, бо її нам все одно додавати в оголошення типу функцій.

Знову ж таки, дуже важливо розрізняти конструктори типів і конструктори значень. В означенні типу даних, все, що стоїть перед `=`, є конструктором типів, а конструктори значень йдуть після (відокремлені знаком `|`, якщо їх декілька). Тому, наприклад, `Vector t t t -> Vector t t t -> t` — то є нісенітниця через те, що в означенні типу мають бути задіяні типи, а конструктор **типу** вектора приймає лише один параметр, тоді як конструктор **значень** приймає три. Нумо досліджувати наші вектори!

```
ghci> Vector 3 5 8 `vplus` Vector 9 2 8
Vector 12 7 16
ghci> Vector 3 5 8 `vplus` Vector 9 2 8 `vplus` Vector 0 2 3
Vector 12 9 19
ghci> Vector 3 9 7 `vectMult` 10
Vector 30 90 70
ghci> Vector 4 9 5 `scalarMult` Vector 9.0 2.0 4.0
```

```
74.0
ghci> Vector 2 9 3 `vectMult` (Vector 4 9 5 `scalarMult` Vector 9 2 4)
Vector 148 666 222
```

8.4 Автоматичні втілення

У підрозділі 3.3 ми пояснили основні принципи дії типокласів. Ми дізналися, що типоклас є схожим на інтерфейс, який означує деяку поведінку. Тип може **втілити** типоклас, якщо він підтримує таку поведінку. Приклад: тип `Int` є втіленням типокласу `Eq`, тому що типоклас `Eq` означує поведінку для речей, які можна порівнювати. А оскільки цілі числа можна порівнювати, `Int` є членом типокласу `Eq`. Справжня користь приходить разом із функціями, які працюють як інтерфейс для `Eq`, а саме — `==` та `/=`. Якщо тип є членом типокласу `Eq`, ми можемо **перевіряти на рівність** [check for equality] значення цього типу за допомогою функції `==`. Ось чому вирази `4 == 4` та `"foo" /= "bar"` **успішно проходять перевірку системою типів** [typecheck] (іншими словами — не містять помилок типу).

Ми також згадували, що типокласи часто плутають із класами з імперативних мов, таких як Java, Python, C++ і таке інше. Це збиває з пантелику багатьох людей. В щойно перелічених мовах класи є «кресленнями», згідно яких конструюються об'єкти, які інкапсулюють в собі стан та можуть виконувати певні дії. Типокласи ж більш за все схожі на інтерфейси. Ми не будуємо об'єкти з типокласів. Натомість ми спочатку створюємо наш тип даних, а вже потім ми думаємо про те, як він може поводитись: якщо як щось, що можна порівнювати, ми робимо його втіленням типокласу `Eq`; якщо він може поводитись як щось, що може бути впорядковане, ми робимо його втіленням типокласу `Ord`; і так далі.

В наступному підрозділі ми розглянемо, як ми можемо вручну робити власноруч створені типи втіленнями типокласів — втілювати типоклас, як ми згодом побачимо, це є те саме, що й означувати функції, означування яких вимагає цей типоклас. Але зараз гляньмо, як Хаскел може втілити отакі типокласи для нас автоматично: `Eq`, `Ord`, `Enum`, `Bounded`, `Show` і `Read`. Наказати Хаскелові автомагічно втілити певну поведінку у наших типах можна за допомогою ключового слова *deriving*, і це ключове слово треба вжити в означенні нашого типу даних.

Розгляньмо цей тип даних:

```
data Person = Person { firstName :: String
                     , lastName :: String
                     , age :: Int
                     }
```

Він описує особу. Будемо вважати, що немає двох людей із однаковим ім'ям, прізвищем та віком. Тепер, наприклад, хай в нас є два записи для двох людей. Чи має сенс перевірка чи відповідають ці записи одній і тій самій особі? Звісно, що так. Ми можемо спробувати порівняти їх і подивитися, чи однакові вони, чи ні. Ось чому має сенс зробити цей тип членом типокласу `Eq`. На цей раз втілимо `Eq` в `Person` автоматично за допомогою *deriving*:

```
data Person = Person { firstName :: String
                     , lastName :: String
                     , age :: Int
                     } deriving (Eq)
```

Якщо ми означимо `Person` в такий спосіб, а потім спробуємо порівняти два значення типу `Person` за допомогою `==` чи `/=`, Хаскел спочатку перевірить, чи збігаються конструктори значень (хоча в цьому випадку є тільки один конструктор значень), а потім перевірить, чи однакові всі поля. Тестування кожної пари полів також відбувається за допомогою `==`. Але є одне але: щоб це спрацювало, типи всіх полів також повинні бути членами типокласу `Eq`. Оскільки і `String`, і `Int` вже втілюють `Eq`, проблем не виникає. Тестуймо наше новеньке втілення:

```
ghci> let mikeD = Person { firstName = "Michael"
                         , lastName = "Diamond"
                         , age = 43 }
ghci> let adRock = Person { firstName = "Adam"
                          , lastName = "Horovitz"
                          , age = 41 }
ghci> let mca = Person { firstName = "Adam", lastName = "Yauch", age = 44 }
ghci> mca == adRock
False
ghci> mikeD == adRock
False
ghci> mikeD == mikeD
True
ghci> mikeD == Person { firstName = "Michael"
                      , lastName = "Diamond"
                      , age = 43 }
True
```

Звісно, оскільки тепер `Person` — в `Eq`, ми можемо використати його замість `a` в усіх функціях, які мають умову типокласу `Eq a` в своїй сиґнатурі типу, як-от, наприклад, `elem`.

```
ghci> let beastieBoys = [mca, adRock, mikeD]
ghci> mikeD `elem` beastieBoys
True
```

Типокласи `Show` та `Read` — для речей, які можуть бути перетворені на рядки або відновлені з рядків відповідно. Як і з `Eq`, якщо ми хочемо, щоб якийсь тип втілював `Show` чи `Read`, а конструктор того типу має поля, тоді типи полів також мають бути членами `Show` чи `Read`. Зробімо наш тип даних `Person` на додачу ще й членом типокласів `Show` і `Read`.

```
data Person = Person { firstName :: String
                     , lastName :: String
                     , age :: Int
                     } deriving (Eq, Show, Read)
```

Тепер можна видрукувати особу в терміналі.

```
ghci> let mikeD = Person { firstName = "Michael"
                         , lastName = "Diamond"
                         , age = 43 }
ghci> mikeD
Person {firstName = "Michael", lastName = "Diamond", age = 43}
ghci> "mikeD is: " ++ show mikeD
"mikeD is: Person {firstName = \"Michael\", lastName = \"Diamond\", age = 43
}"
```

Якщо б ми спробували надрукувати особу в терміналі до того, як тип даних `Person` отримав членство в `Show`, Хаскел би поскаржився на нас, заявляючи, що йому не відомо, як можна представити особу рядком. Але оскільки ми автоматично втілили `Show`, йому це тепер відомо.

`Read` є по суті оберненим типокласом до `Show`. Типоклас `Show` існує для перетворення значень на рядки, а `Read` — для перетворення рядків на значення. Проте пам'ятайте: коли ми використовуємо функцію `read`, ми повинні явно анотувати тип, щоб повідомити Хаскелу, значення якого типу ми хочемо мати в результаті. Якщо ми цього не зробимо в явному вигляді, Хаскел не знатиме про який тип йдеться.

```
ghci> let s =
```

```
    "Person {firstName =\"Michael\", lastName =\"Diamond\", age = 43}"
ghci> read s :: Person
Person {firstName = "Michael", lastName = "Diamond", age = 43}
```

Якщо ми використаємо результат `read` так, що Хаскел зможе **вивести** [infer], що він має перетворити рядок на `Person`, тоді не потрібно буде анотувати тип явно.

```
ghci> let s =
    "Person {firstName =\"Michael\", lastName =\"Diamond\", age = 43}"
ghci> read s == mikeD
True
```

Ми також можемо читати параметризовані типи, але тоді необхідно заповнити параметри типу. Тому ми не можемо записати `read "Just 't'" :: Maybe a`, але можемо записати `read "Just 't'" :: Maybe Char`.

Також можна автоматично втілити й `Ord`. Цей типоклас — для типів, значення яких можна впорядкувати. А якщо порівняти два значення одного типу, але такі, що були створені за допомогою різних конструкторів? У цьому випадку вважається меншим значення, яке було побудовано за допомогою конструктора, який було означено першим. Наприклад, розгляньмо тип `Bool`, який може мати значення `False` або `True`. З метою ілюстрації його поведінки при порівнянні значень будемо вважати, що цей тип є реалізований отак:

```
data Bool = False | True deriving (Ord)
```

Оскільки конструктор значень `False` подано першим, а `True` вказано після нього, `True` вважатиметься більшим за `False`.

```
ghci> True `compare` False
GT
ghci> True > False
True
ghci> True < False
False
```

У типі даних `Maybe a` конструктор значень `Nothing` вказаний перед конструктором значень `Just`, тому значення `Nothing` є завше меншим за значення `Just <<something>>`, навіть коли це `<<something>>` є мінус один мільярд трильйонів. Але якщо ми порівняємо два значення `Just`, то порівнюватимуться речі усередині цих двох `Just`:

```
ghci> Nothing < Just 100
```

```
True
ghci> Nothing > Just (-49999)
False
ghci> Just 3 `compare` Just 2
GT
ghci> Just 100 > Just 50
True
```

Але от `Just (*3) > Just (*2)` не спрацює, бо `(*3)` та `(*2)` є функціями, а функції не втілюють `Ord`.

Ми можемо запросто використати алгебраїчні типи даних[†] для побудови перелічень — типокласи `Enum` та `Bounded` допоможуть нам в цьому. Розгляньмо оцей тип даних:

```
data Day = Monday
         | Tuesday
         | Wednesday
         | Thursday
         | Friday
         | Saturday
         | Sunday
```

Оскільки тут всі конструктори значень є **нульарними** [nullary] (не приймають параметрів, тобто полів), ми можемо зробити цей тип даних членом типокласу `Enum`. Типоклас `Enum` об'єднує разом всі речі, в яких є **попередники** [predecessors] та **наступники** [successors]. Ми можемо також зробити його членом типокласу `Bounded`, який об'єднує усі речі, які мають мінімум і максимум. І оскільки ми маємо таку хорошу нагоду, зробімо так, щоб наш тип на додачу втілював усі типокласи, які можна втілити автоматично, і погляньмо, що тепер з цим типом можна робити.

```
data Day = Monday
         | Tuesday
         | Wednesday
         | Thursday
         | Friday
         | Saturday
```

[†]Алгебраїчні типи даних (АТД) називаються алгебраїчними, бо будувати їх дозволено операціями, які разом утворюють таку собі невеличку «алгебру». Алгебра АТД в Хаскелі складається з «сум» (наприклад, в `data Pair = I Int | B Bool`, маємо розгалуження — структура даних `Pair` є або `Int` із конструктором `I` або `Bool` із конструктором `B`) і «добутків» (наприклад, в `data Pair = P Int Bool`, маємо поєднання — структура даних `Pair` є поєднанням `Int` і `Bool` за допомогою конструктора `P`).

```
        | Sunday
          deriving (Eq, Ord, Show, Read, Bounded, Enum)
```

Завдяки членству в `Show` та `Read`, ми можемо перетворити значення цього типу на рядки — і навпаки.

```
ghci> Wednesday
Wednesday
ghci> show Wednesday
"Wednesday"
ghci> read "Saturday" :: Day
Saturday
```

Дякуючи типокласам `Eq` та `Ord`, ми можемо порівнювати та впорядковувати дні.

```
ghci> Saturday == Sunday
False
ghci> Saturday == Saturday
True
ghci> Saturday > Friday
True
ghci> Monday `compare` Wednesday
LT
```

Через те, що тип є членом `Bounded`, ми можемо знайти найменший день і день найбільший.

```
ghci> minBound :: Day
Monday
ghci> maxBound :: Day
Sunday
```

Не забуваймо про `Enum`. Ми можемо знаходити попередників та наступників днів та використовувати дні в побудові діапазонів днів!

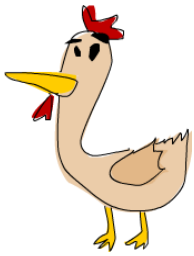
```
ghci> succ Monday
Tuesday
ghci> pred Saturday
Friday
ghci> [Thursday .. Sunday]
[Thursday,Friday,Saturday,Sunday]
ghci> [minBound .. maxBound] :: [Day]
[Monday,Tuesday,Wednesday,Thursday,Friday,Saturday,Sunday]
```

Круто!

8.5 Типи-синоніми

Ми вже згадували, що `[Char]` та `String` є еквівалентними та взаємозамінними. Це реалізовано за допомогою **синонімії типів**. Типи-синоніми по суті нічого не роблять: синонімія існує для того, щоб можна було давати типам різні імена, для поліпшення читабельності коду і документації. Ось як стандартна бібліотека означує тип-синонім `String`, який є синонімом типу `[Char]`.

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



Ми ввели нове ключове слово *type*. Воно може збити декого із пантелику, бо ми взагалі-то не створюємо нічого нового (ми це робили за допомогою ключового слова *data*). Натомість ми просто створюємо синонім для типу, який вже існує.

Якщо ми придумаємо функцію, яка переганяє рядок у верхній регістр і назвемо її `toUpperString`, або ще щось, то ми можемо оголосити її як `toUpperString :: [Char] -> [Char]` або ж як `toUpperString :: String -> String`. Обидва варіанти є по суті ідентичними, але останній приємніше читати.

Коли ми мали справу із модулем `Data.Map`, ми спочатку представили телефонну книгу асоціативним списком, а потім поміняли представлення на мапу. Як ми уже з'ясували, асоціативний список — то є список пар ключ-значення. Тож погляньмо на телефонну книгу, яка у нас була.

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

Ми бачимо, що тип `phoneBook` є `[(String,String)]`. Це нам каже, що ми маємо справу із асоціативним списком, який асоціює рядки із рядками, і більш нічого. Створімо синонім цього типу, щоб надати ще трохи додаткової інформації в оголошенні типу.

```
type PhoneBook = [(String,String)]
```

Тепер оголошенням типу для нашої телефонної книги може бути `phoneBook :: PhoneBook`. Так само створімо синоніми для типу `String`.

```
type PhoneNumber = String
```

```
type Name = String
type PhoneBook = [(Name,PhoneNumber)]
```

Типи-синоніми до `String` Хаскел-програмісти зазвичай створюють, коли хочуть надати більше інформації про семантичне навантаження цих рядків, і про те, як ці рядки мають використовуватись у функціях.

А тепер, означуючи функцію, яка приймає пару — ім'я та число — і відповідає, чи є така пара у нашій телефонній книзі, ми можемо почати з гарного і інформативного оголошення типу.

```
inPhoneBook :: Name -> PhoneNumber -> PhoneBook -> Bool
inPhoneBook name pnumber pbook = (name,pnumber) `elem` pbook
```

Якщо б ми вирішили не користуватися тут синонімами, ця функція мала б тип `String -> String -> [(String,String)] -> Bool`. Оголошення із синонімами, як бачимо, легше читати і розуміти. Але з синонімами можна як недоборщити, так і переборщити! Синоніми зазвичай означують для того, щоб описати, що саме представляють «стандартні» типи у наших функціях (і таким чином оголошення типів стають краще задокументованими); також синоніми стають в пригоді там, де в оголошенні функцій є «порівняно довгі» типи (на кшталт `[(String,String)]`), що повторюються багато разів і мають якесь специфічне семантичне навантаження в контексті цих функцій.

Синонімічні типи можна параметризувати. Хай нам потрібен тип, який представлятиме асоціативний список і ми хочемо, щоб він був достатньо загальним. Щоб можна було використовувати які завгодно типи в ролі ключів і значень, можемо зробити так:

```
type AssocList k v = [(k,v)]
```

Тепер для функції, яка бере ключ і шукає відповідне тому ключе́ві значення в асоціативному списку, можна записати сиґнатуру `Eq k => k -> AssocList k v -> Maybe v`. Конструктор типу `AssocList` приймає два типи і повертає конкретний тип — такий як, наприклад, `AssocList Int String`.

«Конкретний тип» каже: Увага! Коли я кажу *конкретні типи*, я маю на увазі або «повністю застосовані» конструктори типів на кшталт `Map Int String`, або ж мова іде про поліморфні функції — фіговини на кшталт `[a]` або `Ord a => Maybe a` і всяке таке. А ще, деколи я та інші непересічні типи вживаємо такі «розмиті» фрази як «тип `Maybe`», але насправді ми не маємо на увазі «тип», бо кожен чайник знає, що `Maybe` є конструктором типів. Коли я подам

тип `String` до `Maybe`, і отримаю `Maybe String` — ось тоді я матиму конкретний тип! Значення мають тип, і цей тип може бути лише типом конкретним. Отож, підсумовуючи, живи швидко, кохай палко і тримай порох сухим!

Ми можемо будувати нові функції частковим застосуванням інших функцій. За аналогією, частково застосовуючи конструктори типів, ми отримуємо нові конструктори типів. Так само, як виклик функції із недостатньою кількістю параметрів повертає нову функцію, не додаючи параметрів конструкторові типів, ми отримуємо частково застосований конструктор типів. Якщо нам треба тип, який представлятиме усі мапи `Data.Map` із цілих чисел у що завгодно, ми можемо або записати його так:

```
type IntMap v = Map Int v
```

Або ж так:

```
type IntMap = Map Int
```

В обох реалізаціях, конструктор типу `IntMap` приймає один тип-параметр, і цей тип параметризуватиме те, у що саме ця мапа відображатиме цілі числа.

Мало не забув! Якщо ви збираєтеся це реалізувати, скоріш за все ви зробите імпорт `Data.Map` в підпростір імен. Якщо ви імпортували в підпростір, ім'я підпростору має передувати й іменам конструкторів типів. Отже, якщо ім'я підпростору є Map (буде таке за замовчуванням, якщо ви виконаєте `import qualified Data.Map`), треба буде переозначити наш `IntMap` отак: `type IntMap = Map.Map Int`.

Впевніться, що ви дійсно розумієте різницю між конструкторами типів і конструкторами значень. Ми створили типи-синоніми `IntMap` чи `AssocList`, які насправді є конструкторами типів (синонімами конструкторів типів!), і їх також не варто плутати із конструкторами значень. Тому речі такі, як, наприклад, `AssocList [(1,2),(4,5),(7,9)]`, є нісенітницями. Все, що дозволяє нам синонімія, — це вживати інше ім'я для типу чи конструктору типів в в сиґнатурі типу функції чи в анотації типу виразу. Ми можемо записати `[(1,2),(3,5),(8,9)] :: AssocList Int Int`, і це означатиме, що числа всередині пар асоціативного списку матимуть тип `Int`, але ми все ще можемо використовувати такий список як звичайний список пар цілих чисел. Типи-синоніми (і типи загалом) можуть лише використовуватися в тих місцях в хаскельному

коді, де йдеться про типи. А саме — в означеннях нових типів (тобто, в означеннях за допомогою ключових слів *data* та *type*) або коли ми знаходимося після `::` (тобто, в типосиґнатурах функцій та в анотаціях типів).

Інший чудовий двопараметровий конструктор типів — `Either a b`. Орієнтовно він є означений ось так:

```
data Either a b = Left a | Right b deriving (Eq, Ord, Read, Show)
```

Він має два конструктори значень. Конструктор `Left` містить щось типу `a`, а `Right` — щось типу `b`. Тому можна використовувати цей тип для інкапсуляції значення якогось одного типу або значення якогось іншого типу. Коли ми отримуємо значення типу `Either a b`, ми зазвичай зіставляємо із взірцем по кожному з конструкторів і далі робимо різні речі, в залежності від того, зіставився конструктор `Left` чи `Right`.

```
ghci> Right 20
Right 20
ghci> Left "w00t"
Left "w00t"
ghci> :t Right 'a'
Right 'a' :: Either a Char
ghci> :t Left True
Left True :: Either Bool b
```

Ми бачили, що `Maybe a` використовувався в основному для представлення результатів обрахунків, які могли або успішно, або неуспішно завершитися. Але іноді `Maybe a` недостатньо, оскільки `Nothing` не передає достатньо інформації про природу того неуспіху (все, що передається — це те, що обрахунок був неуспішним). Це підходить, наприклад, для функцій, які можуть «неуспішно завершитися» тільки в один спосіб. Або ж для випадків коли нас не цікавить причина неуспіху. До прикладу: пошук, який виконується `Data.Map.lookup`, не завершиться поверненням значення лише якщо мапа не містить ключа, за яким ведеться пошук, а отже ми знаємо точно що сталося. Однак якщо нас цікавить, чому якась функція завалилася, або ж в який спосіб з декількох можливих вона завалилася, то ми зазвичай повертаємо з такої функції `Either a b`, де `a` є типом, що звітує нам про можливу причину неуспіху, а `b` є типом значення-результату, який (значення-результат) повертається у випадку успішного завершення. Отже, помилки використовують конструктор значень `Left`, тоді як результати використовують `Right`.

Приклад: в школі є шафки для того, щоб школярі мали куди покласти свої плакати гурту Guns'n'Roses. Кожна шафка має замок, а кожен замок має **код** [code combination]. Коли школяр хоче отримати шафку, він каже вчителеві її номер, а той видає код замка для неї. Однак, якщо шафка вже використовується

кимось іншим, вчитель не може видати її код, і тоді потрібно обрати якусь іншу. Скористаємося `Data.Map`, щоб представити шафки. Номер шафки буде відображатися в пару значень — стан шафки (зайнята чи вільна) і код замка.

```
import qualified Data.Map as Map

data LockerState = Taken | Free deriving (Show, Eq)

type Code = String

type LockerMap = Map.Map Int (LockerState, Code)
```

Прості речі. Ми означили новий тип даних, що описує, чи є вільною шафка чи ні, і створили тип-синонім для коду замка. Ми також означили мапу шафок як тип-синонім до типу, що відображає цілі числа в пари `(LockerState, Code)`. А тепер ми напишемо функцію, яка шукає код у мапі шафок. Ми скористаємося типом `Either String Code` для представлення нашого результату, бо ми можемо і не знайти код. Причини дві — шафка може бути зайнятою і в цьому випадку ми не маємо права показати код, або ж шафки із таким номером взагалі не існує. Якщо пошук завершився неуспішно, ми просто скористаємося `String` для повідомлення, що саме трапилося.

```
lockerLookup :: Int -> LockerMap -> Either String Code
lockerLookup lockerNumber map =
    case Map.lookup lockerNumber map of
        Nothing -> Left $ "Locker number " ++ show lockerNumber
                                           ++ " doesn't exist!"
        Just (state, code) -> if state /= Taken
                              then Right code
                              else Left $ "Locker " ++ show lockerNumber
                                                    ++ " is already taken!"
```

Ми виконуємо звичайний пошук в мапі. Якщо ми отримуємо `Nothing`, то повертаємо значення типу `Left String`, яке каже, що шафка взагалі не існує. Якщо ж ми знаходимо таку шафку, тоді виконуємо додаткову перевірку, щоб дізнатися чи зайнята ця шафка. Якщо так, то повертаємо `Left` з повідомленням про те, що вона вже зайнята. Якщо не зайнята, тоді повертаємо значення типу `Right Code`, в якому ми даємо учневі правильний код замка. Насправді типом результату є `Right String`, але ми ввели синонім до нього, щоб надати трохи більше інформації про цей тип в сиґнатурах типу. Ось приклад мапи:

```
lockers :: LockerMap
lockers = Map.fromList
    [(100,(Taken,"ZD39I"))
```

```
    ,(101,(Free,"JAH3I"))
    ,(103,(Free,"IQSA9"))
    ,(105,(Free,"QOTSA"))
    ,(109,(Taken,"893JJ"))
    ,(110,(Taken,"99292"))
    ]
```

Тепер спробуймо пошукати там якісь коди замків.

```
ghci> lockerLookup 101 lockers
Right "JAH3I"
ghci> lockerLookup 100 lockers
Left "Locker 100 is already taken!"
ghci> lockerLookup 102 lockers
Left "Locker number 102 doesn't exist!"
ghci> lockerLookup 110 lockers
Left "Locker 110 is already taken!"
ghci> lockerLookup 105 lockers
Right "QOTSA"
```

Ми могли б скористатися `Maybe a` для представлення результату, але тоді не змогли б дізнатися у випадку неуспішного пошуку, чому не отримали код. А в теперішній реалізації маємо інформацію про причину неуспіху — її вбудовано в тип-результат.

8.6 Рекурсивні структури даних

Як ми вже бачили, конструктор в алгебраїчному типі даних може мати кілька полів (а може й не мати взагалі) і кожне поле повинно бути якогось конкретного типу. Цікаво, що ми також можемо створювати типи, в яких є конструктори, що мають поля такого ж типу, як і тип, що ми його створюємо! Іншими словами, ми можемо означувати рекурсивні типи, де одне значення якогось типу містить значення того ж типу, яке, в свою чергу, містить ще одне або декілька значень того ж самого типу і так далі.

Поміркуймо над цим списком: `[5]`. Це просто синтаксичний цукор для `5:[]`. Ліворуч від `:` стоїть значення, а праворуч — список, і в цьому випадку він порожній. Тепер, а як щодо списку `[4,5]`? Що ж, це розцукровується в `4:(5:[])`. Дивимося на перший оператор `:` і бачимо, що праворуч стоїть елемент, а ліворуч — список `5:[]`. Так само можна «розібрати

по запчастинах» список `3:(4:(5:6:[]))` — його можна записати як `3:4:5:6:[]` (бо оператор `:` є правоасоціативним) або ж як `[3,4,5,6]`.

Можна сказати, що список може бути порожнім списком або він може бути елементом, з'єднаним за допомогою `:` із іншим списком (який може бути порожнім списком або не бути).

Тож скористаймося алгебраїчними типами даних для створення нашого власного списку!

```
data List a = Empty | Cons a (List a) deriving (Show, Read, Eq, Ord)
```

Це читається так само, як і наше означення списків у одному із попередніх абзаців. Список є або порожнім списком, або є поєднанням значення і списку. Якщо це збиває з пантелику, можливо буде легше зрозуміти це означення, якщо його переписати із використанням синтаксису для записів?

```
data List a = Empty |
    Cons { listHead :: a, listTail :: List a} deriving (Show, Read, Eq, Ord)
```

Можливо вас також підбентежує[*] конструктор `Cons`? *cons* — це словосинонім до `:`. Річ у тім, що в списках `:` є насправді конструктором, який приймає значення та інший список і повертає список. Ми вже можемо тут скористатися нашим новим типом для списків! Іншими словами, конструктор `:` має два поля. Одне поле типу `a`, а інше — типу `[a]`.

```
ghci> Empty
Empty
ghci> 5 `Cons` Empty
Cons 5 Empty
ghci> 4 `Cons` (5 `Cons` Empty)
Cons 4 (Cons 5 Empty)
ghci> 3 `Cons` (4 `Cons` (5 `Cons` Empty))
Cons 3 (Cons 4 (Cons 5 Empty))
```

Ми викликали наш конструктор `Cons` в інфіксний спосіб, щоб можна було легше бачити його подібність до `:`. Конструктор `Empty` — схожий на `[]`, а вираз `4 `Cons` (5 `Cons` Empty)` — подібний до `4:(5:[])`.

Якщо назва функції складається лише з спецсимволів (на кшталт `>`, `:`, `$`, `+` і таке подібне), ця функція стає інфіксною автоматично[‡]. Так само із конструкторами[§]: вони ж бо є звичайними функціями, які повертають значення.

[*] Це слово є ©Олег Науменко.

[‡] Можливо, треба буде увімкнути LANGUAGE-директиву `TypeOperators`. Також, ім'я інфіксної функції не може починатися з `:`.

[§] Але є обмеження: ім'я конструктора значень має починатися з `:`.

Дивіться!

```
infixr 5 :-:
data List a = Empty | a :-: (List a) deriving (Show, Read, Eq, Ord)
```

Перш за все, зверніть увагу на нову синтаксичну конструкцію для **оголошення асоціативності** [fixity declaration]. Коли ми означуємо функції як оператори, ми можемо вказати для них асоціативність (але це не обов'язково). Асоціативність описує, яким є оператор — право- чи **лівоасоціативним** [left-associative] і наскільки «сильно». Наприклад, оголошення асоціативності для `*` є `infixl 7 *`, а для `+` — `infixl 6 +`. Це означає, що вони обидва є лівоасоціативними (тобто, `4 * 3 * 2` — це є `(4 * 3) * 2`), але `*` **зв'язує сильніше** [binds tighter], ніж `+`, бо має більше значення асоціативності (7 в `*` проти 6-ти в `+`), і тому `5 + 4 * 3` — є це `5 + (4 * 3)`.

Інше, що ми змінили — ми просто записали `a :-: (List a)` замість `Cons a (List a)`. Тепер ми можемо складати списки нашого найновішого спискового типу ось так:

```
ghci> 3 :-: 4 :-: 5 :-: Empty
(:-:) 3 ((:-:) 4 ((:-:) 5 Empty))
ghci> let a = 3 :-: 4 :-: 5 :-: Empty
ghci> 100 :-: a
(:-:) 100 ((:-:) 3 ((:-:) 4 ((:-:) 5 Empty)))
```

Оскільки ми автоматично втілили `Show` для нашого типу, Хаскел покаже його, але інтерпретуватиме наш конструктор як префіксну функцію — ось звідки дужки навколо оператора (пам'ятайте, що `4 + 3` — це є `(+) 4 3`).

Побудуймо функцію, яка додає два списки в один. Ось як означений оператор `++` для звичайних списків.

```
infixr 5  ++
(++) :: [a] -> [a] -> [a]
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

То ми просто поцупимо це означення для нашого власного списку. Назвемо функцію `.++`.

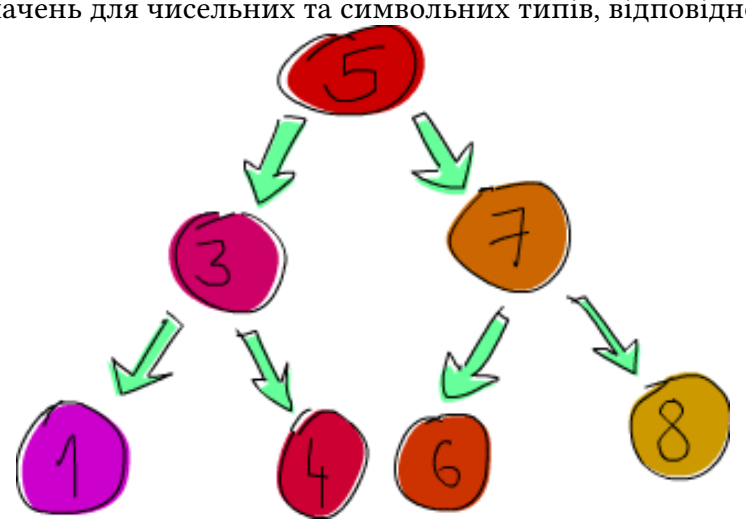
```
infixr 5  .++
(.++) :: List a -> List a -> List a
Empty .++ ys      = ys
(x :-: xs) .++ ys = x :-: (xs .++ ys)
```

І перевірмо, чи працює...

```
ghci> let a = 3 :-: 4 :-: 5 :-: Empty
ghci> let b = 6 :-: 7 :-: Empty
ghci> a .++ b
(:-:) 3 ((:-:) 4 ((:-:) 5 ((:-:) 6 ((:-:) 7 Empty))))
```

Гарно. Є гарно. Якщо б захотіли, ми могли б втілити всі функції, що працюють із списками, для нашого власного спискового типу.

Зверніть увагу, як ми зіставили із взірцем `(x :-: xs)`. Зіставлення із взірцем в Хаскелі насправді зіставляє із взірцем конструктори. Ми можемо зіставити `:-:` тому, що це є конструктор значень для нашого власного списку, і можемо також зіставити `:`, бо це є конструктор значень для стандартного списку. Те ж стосується й `[]`. Оскільки зіставлення з взірцем «іде» (лише) по конструкторах, ми здатні зіставляти всякі такі штукенції — звичайні там префіксні конструктори, чи то речі такі як `8` або `'a'`, які є, по суті, конструкторами значень для чисельних та символьних типів, відповідно.



Зараз ми запрограмуємо **бінарне дерево пошуку [binary search tree]**. Якщо ви не знайомі із бінарними деревами (пошуку) з мов таких як C, ось, коротко, про цю структуру даних: дерево складається з елементів (або вузлів); зазвичай кожен елемент дерева містить в собі значення і, на додачу, ще й вказує на два інші елементи (так звані *елементи-діти*); дітей розрізняють — є ліве дитинча, і є дитинча праве. Значення в лівій дитині є меншим за значення елемента-батька, а в правій — більшим (а якщо коротко — лівий елемент є меншим за батьківський, а правий — більшим). Кожен із елементів-дітей також може вказувати ще на два елементи. В загальному випадку дітей може й не бути, або може бути лише одна дитина. В результаті, кожен елемент може мати не більше двох піддерев. Найцікавішою властивістю, що випливає з такої побудови, є те, що відомо наперед, що всі елементи в лівому піддереві елемента, в якого, скажімо, значення є п'ятірка, будуть меншими за п'ять. А елементи в правому піддереві будуть більшими.

Якщо нам потрібно перевірити, чи 8 є в нашому дереві (див. малюнок), ми починаємо з 5, і, оскільки 8 більше за 5, йдемо праворуч. Тепер ми опинилися біля елементу із сімкою, і, оскільки 8 більше за 7, ми знову ідемо праворуч. Ура — ми знайшли наше число, за три кроки! Якби ж це був звичайний список (або дерево, але не збалансоване), нам би довелося здійснити сім кроків замість трьох, щоб перевірити чи є там вісімка.

Множини і мапи із `Data.Set` та `Data.Map` реалізовані на базі дерев, тільки замість звичайних бінарних дерев пошуку вони використовують збалансовані бінарні дерева, які завжди себе підтримують в хорошій (збалансованій) формі. Але зараз ми будемо реалізовувати звичайні бінарні дерева пошуку.

А тепер проговорюються отакі слова[†]: дерево є або порожнім, або ж воно є елементом, який містить якесь значення і два піддерева. Схоже, в пригоді стане алгебраїчний тип даних!

```
data Tree a = EmptyTree
            | Node a (Tree a) (Tree a) deriving (Show, Read, Eq)
```

Гаразд, добре, то є добре. Замість того, щоб вручну будувати дерево, ми напишемо функцію, яка бере дерево та значення і вбудовує у те дерево вузол із тим значенням. Досягається це ось як: порівнюємо значення, яке хочемо вставити, із значенням у кореневому вузлі, і, якщо воно менше за корінь, йдемо наліво, а якщо більше — направо. Ми виконуємо цю процедуру для кожного наступного вузла, допоки досягнемо порожнього дерева. Як тільки його досягли, просто замінюємо те порожнє дерево на вузол із нашим значенням.

В мовах на кшталт C ця процедура була б реалізована за допомогою змін **вказівників** [pointers] та змін значень всередині вузлів дерева. В Хаскелі ж, насправді, ми не можемо змінити дерево, тому нам потрібно створювати нове піддерево щоразу, коли було прийняте рішення про те, куди треба йти (вліво або вправо). **Функція вставки** [insert function] повертає зовсім[‡] нове дерево, оскільки в Хаскелі немає концепції вказівника — Хаскел знає лише про **значення** [values]. Таким чином, тип нашої функції вставки буде схожим на `a -> Tree a -> Tree a`. Функція бере значення і дерево та повертає нове дерево, в яке вбудовано новий елемент із цим значенням. Цей підхід може здатися **неефективним** [inefficient], але він є таким, бо **чистофункційні структури даних** [purely functional data structures] є **незміненними** [immutable], а це дозволяє реалізації, де стара (до модифікації) і нова (після) версії структури даних мають (через так званий **поділ** [sharing]) багато спільних частин[†].

[†]Фраза «А тепер проговорюються отакі слова» є ©Анісімов Ігор Олексійович. Використано без дозволу.

[‡]Семантично «зовсім нове». Але реалізація зазвичай така, що воно матиме спільні частини із попереднім деревом.

[†]Легше перепросити, ніж переписати це речення. Отже — перепрошую! :) NB: Тепер зро-

Отже, ось дві функції. Одна є **допоміжною функцією** [utility function] для створення дерева із одним-єдиним вузлом — так званого **однодерева** [singleton tree], а інша — для вставки значення в дерево.

```
singleton :: a -> Tree a
singleton x = Node x EmptyTree EmptyTree

treeInsert :: Ord a => a -> Tree a -> Tree a
treeInsert x EmptyTree = singleton x
treeInsert x (Node a left right)
    | x == a = Node x left right
    | x < a  = Node a (treeInsert x left) right
    | x > a  = Node a left (treeInsert x right)
```

Функція `singleton` є допоміжною функцією для створення нашвидкоруч вузла, який має усередині якесь значення і два порожні піддерева. У функції вставки ми спочатку перевіряємо взірцем крайову умову. Якщо ми досягли порожнього піддерева, значить ми є там, де ми хочемо бути, і тоді ми замінюємо те порожнє дерево на однодерево із нашим значенням. Якщо ми вставляємо в **непорожнє дерево** [non-empty tree], ми повинні дещо перевірити. Перш за все, якщо значення, яке ми вставляємо, дорівнює кореневому, просто повертаємо те дерево без змін. Якщо менше, повертаємо дерево, що є схожим на дерево на вході: значення в корені без змін, праве піддерево без змін, але замість лівого піддерева ми вбудовуємо в нього нове піддерево, в яке (буде) вставлено наше значення. Те ж саме (з точністю до заміни ліво на право) відбувається, якщо наше значення є більшим за значення з **кореневого елемента** [root node].

Далі ми створимо функцію, яка перевіряє, чи містить дерево певний елемент. По-перше, означмо **крайову умову** [edge condition]. Якщо ми шукаємо елемент у порожньому дереві, то його там явно немає. Гаразд. Зауважте, як це схоже на крайову умову пошуку елементів у списках. Шукати елемент у порожньому списку теж не варто, бо його там немає (і не тільки його — там немає багатьох інших елементів!). Порожнеча… Але годі з цим, рухаємося далі: якщо ми шукаємо елемент в непорожньому дереві, тоді нам треба дещо перевірити. Якщо значення елемента в кореневому вузлі дорівнює значенню, що ми шукаємо — ура! А якщо ні — що тоді? Тоді ми можемо використати той факт, що всі «ліві елементи» є менші за кореневий. Тому, якщо елемент, що ми шукаємо, є менший за кореневий, тоді ми продовжуємо пошуки у лівому піддереві. Якщо ж він більший — будемо шукати його у правому.

```
treeElem :: Ord a => a -> Tree a -> Bool
treeElem x EmptyTree = False
```

зуміло, чому цього речення бракує в оригіналі.

```
treeElem x (Node a left right)
    | x == a = True
    | x < a  = treeElem x left
    | x > a  = treeElem x right
```

Як бачимо, параграф слів втілився в п'ять рядків хаскелівського коду. Пограймося-но із нашими деревами! Замість того, щоб будувати дерево вручну (хоча ми б могли), ми скористаємося згортком і будуватимемо дерево зі списку. Пам'ятайте: багато алгоритмів, в яких список обробляється елемент за елементом і які повертають якесь значення, можна реалізувати за допомогою згортка! Ми почнемо із порожнього дерева і згортатимемо список з правого боку, вставляючи елемент за елементом в наше дерево-накопичувач.

```
ghci> let nums = [8,6,4,1,7,3,5]
ghci> let numsTree = foldr treeInsert EmptyTree nums
ghci> numsTree
Node 5 (Node 3 (Node 1 EmptyTree EmptyTree)
 (Node 4 EmptyTree EmptyTree))
  (Node 7 (Node 6 EmptyTree EmptyTree)
   (Node 8 EmptyTree EmptyTree))
```

`treeInsert` у виклику `foldr` була **згортаючою функцією** [folding function] (вона бере дерево та елемент зі списку та повертає нове дерево), а `EmptyTree` був стартовим накопичувачем. `nums`, звісно, був списком, який ми згортали.

Коли ми друкуємо те дерево в консолі, воно не дуже легко читається, але якщо постараємося, можемо розгледіти його топологію. Бачимо, що кореневий елемент містить 5 і має два піддерева, із значеннями 3 та 7 в кореневих елементах цих двох піддерев, і так далі.

```
ghci> 8 `treeElem` numsTree
True
ghci> 100 `treeElem` numsTree
False
ghci> 1 `treeElem` numsTree
True
ghci> 10 `treeElem` numsTree
False
```

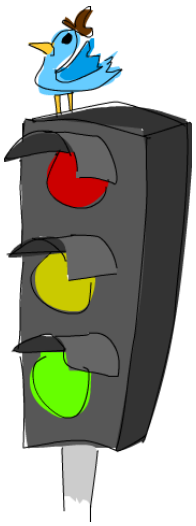
Перевірка на наявність [membership check] елемента також гарненько працює. Чудово.

Отже, як ми побачили, алгебраїчні структури даних є дійсно крутими й потужними концепціями у Хаскелі. Ми можемо використовувати їх в побудові будь-чого, починаючи від булевих значень та **перелічення днів тижня**

[weekday enumeration] і не закінчуючи бінарними деревами пошуку і багато чим іще!

8.7 Вступ до типокласів

Наразі ми вивчили деякі стандартні типокласи Хаскела і розглянули, які до них належать типи. Ми також навчилися автоматично створювати втілення стандартних типокласів для наших власних типів — себто, не писали втілення самі, а просили, щоб Хаскел вивів їх для нас. Ну а у цьому розділі ми навчимося створювати нові, власні типокласи і втілювати їх вручну.



Коротко нагадаємо про типокласи: типокласи схожі на інтерфейси. Типоклас означує деяку поведінку (як-от перевірку на рівність, порівняння для впорядкування, можливість перелічення), а потім для типів, які можуть поводитися таким чином, пишуться відповідні втілення. Втілити типоклас — це те саме, що означити функції, означування яких той типоклас «вимагає». Тому, коли ми кажемо, що «наш тип є втіленням типокласу» або «наш тип втілює типоклас», то маємо на увазі, що ми можемо використовувати функції, типосиґнатури яких той типоклас означує[†], в роботі із цим нашим типом.

Типокласи не мають практично нічого спільного із класами в мовах таких як Java або Python. Це заплутує багатьох людей, тому я б хотів, щоб ви прямо зараз забули все, що знали про класи в імперативних мовах.

Наприклад, типоклас `Eq` є для речей, які можна перевірити на рівність — і він вимагає означення функцій `==` та `/=`. Якщо ми маємо якийсь тип (скажімо `Car`), а перевірка двох машин на рівність є цілком природною (за допомогою функції `==`), тоді є цілком природно зробити тип `Car` членом `Eq`, написавши відповідне втілення.

Ось як типоклас `Eq` є означено у стандартному модулі `Prelude`:

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

Чекайте, чекайте, чекайте! Якийсь новий дивний синтаксис та нові ключові слова! Не хвилюйтеся, за секунду все стане зрозумілим. Спершу, коли ми пишемо `class Eq a where`, це означає, що ми означуємо новий типоклас, який зветься `Eq`. `a` є параметром типу і це означає, що `a` гратиме роль типу, який втілюватиме `Eq`. Він не обов'язково повинен називатися `a`, ім'я не обов'язково

[†]І, в такий спосіб, вимагає означення цих функцій від членів. Але типоклас може також пропонувати й означення для цих функцій.

має складатися із однієї літери — головне, щоб те ім'я починалося з малої літери. Тоді ми означаємо декілька функцій. Надання означень не є обов'язковим — вимагається лише надати типосиґнатури для функцій (себто, тип функції і її ім'я).

Примітка: Декому могло бути зрозумілішим, якби ми замість `a` вжили б щось більш інформативне, як от, наприклад, `equatable` (з англійської, «equatable» — «той, кого можна перевірити на рівність»), і записали `class Eq equatable where`, а потім оголосили б тип отак: `(==) :: equatable -> equatable -> Bool`.

А втім, ми *таки записали* тіла для функцій, яких вимагає `Eq`, тільки от ми означили їх взаєморекурсивно. Ми сказали, що два значення типокласу `Eq` є рівними, якщо вони не різні, і що вони різні, якщо вони не дорівнюють одне одному. Насправді, це робити було зовсім не обов'язково, але ми зробили — і скоро побачимо, як нам це стане в пригоді.

Примітка: Якщо ми напишемо `class Eq a where`, а потім подамо оголошення типу «усередині» цього типокласу на кшталт `(==) :: a -> a -> Bool`, то в майбутньому, якщо перевіримо тип такої функції, він буде `Eq a => a -> a -> Bool`.

Тепер в нас є типоклас — і що ми можемо з ним робити? Що ж, зовсім небагато, насправді. Але тільки-но почнемо будувати втілення того типокласу, почнемо отримувати гарний **функціонал** [functionality]. Тож розгляньмо наступний тип:

```
data TrafficLight = Red | Yellow | Green
```

Він означує стани світлофора. Зауважте, що цього разу ми не втілювали ніякі типокласи автоматично. Це тому, що ми плануємо написати декілька втілень вручну, хоча типокласи такі як `Eq` та `Show` ми могли б втілити автоматично. Ось як `TrafficLight` втілюватиме `Eq`:

```
instance Eq TrafficLight where
    Red == Red       = True
    Green == Green   = True
    Yellow == Yellow = True
    _ == _           = False
```

Тут ми скористалися **ключовим словом** [keyword] *instance*. Таким чином, ключове слово *class* призначене для означування нових типокласів, а *instance* — для створення втілень типокласів для наших типів. Коли ми означували `Eq`, то записали `class Eq a where` і сказали, що `a` є тип, втілення якого буде створено пізніше, і під `a` розумівся якийсь (будь-який) тип. Можемо легко тут це бачити, бо, коли створюємо втілення, пишемо `instance Eq TrafficLight where` — замінюємо `a` на тип, для якого пишемо втілення, — себто, на `TrafficLight`.

Оскільки `==` було означено в типокласі `Eq` **через** [in terms of] `/=` та навпаки, тепер нам треба було лише **замістити** [to override] якесь одне із двох у втіленні. Це зветься **мінімальним повним означенням** [minimal complete definition] для типокласу, себто, — мінімальним набором функцій, які потрібно реалізувати для типу, щоб він міг поводитися так, як зазначає типоклас. Мінімальне повне означення для `Eq` — це або заміщення `==`, або заміщення `/=`. Якщо б `Eq` було означено так:

```
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
```

нам би довелося писати тіла для обох цих функцій у кожному втіленні, бо Хаскел не знає, що ці функції пов'язані. Тоді мінімальним повним означенням був би боєкомплект з `==` та `/=`.

Як бачите, ми реалізували `==` просто за допомогою зіставлення із взірцем. Оскільки випадків, де два сигнали світлофора є нерівними, є набагато більше, ми вказали лише ті, коли сигнали однакові, а потім просто зловили решту у фінальний **універсальний взірець** [catch-all pattern] — якщо це не одна із попередніх комбінацій, то сигнали не однакові.

Втілимо також вручну типоклас `Show`. Щоб задовольнити вимоги мінімального повного означення для `Show`, потрібно просто реалізувати його функцію `show`, яка приймає значення і перетворює його на рядок.

```
instance Show TrafficLight where
    show Red    = "Red light"
    show Yellow = "Yellow light"
    show Green  = "Green light"
```

Знову ж таки, ми скористалися зіставленням із взірцем для досягнення мети. Погляньмо на це в дії:

```
ghci> Red == Red
True
ghci> Red == Yellow
```

```
False
ghci> Red `elem` [Red, Yellow, Green]
True
ghci> [Red, Yellow, Green]
[Red light,Yellow light,Green light]
```

Симпатично. Ми могли б автовтілити `Eq` і це було б повним еквівалентом цієї нашої «ручної роботи» (але цього разу ми реалізували вручну з педагогічних міркувань). Однак, автовтілення `Show` перетворювало б конструктори значень на рядки безпосередньо. А тут нам закортіло, щоб сигнали серіалізувалися в рядки шаленої інформативності, як от `"Red light"`, і тому нам довелося втілювати вручну.

Можна також створювати типокласи, які є **підкласами** [subclasses] інших типокласів. Означення типокласу `Num` є доволі довгим, але ось як виглядає його початок:

```
class Eq a => Num a where
   ...
```

Як ми вже згадували, є багато місць, куди можемо **увіпхнути** [cram in] умови типокласів. І тут — одне з таких місць: `class Eq a => Num a where` — це є те ж саме що й `class Num a where`, тільки ми вимагаємо, що наш тип `a` повинен втілювати `Eq`. Ми, по суті, кажемо, що потрібно втілити `Eq` до того, як можна починати втілювати `Num`. Перед тим, як якийсь тип зможе вважатися числом, цілком природно вимагати можливості перевірки на рівність для значень цього типу. От і власне все, що треба сказати про створення **підкласів** [subclass], — це є просто додавання **умови типокласу** [class constraint] в означення типокласу. І тепер, коли ми означуємо тіла функцій у *class*-означенні чи то коли ми означуємо їх у *instance*-означенні, ми спираємося на те, що `a` є членом `Eq` і, таким чином, можна користуватися `==` і перевіряти на рівність значення, що мають тип `a`.

А як же тип `Maybe` та списковий тип — чи можуть вони втілювати типокласи? Те, що відрізняє `Maybe` від, скажімо, `TrafficLight` — це те, що `Maybe` не є конкретним типом, а конструктором типу, який приймає один тип-параметр (наприклад, `Char` чи щось інше) і повертає **конкретний тип** [concrete type] (наприклад, `Maybe Char`). Погляньмо на типоклас `Eq` знову:

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

З оголошень типу бачимо, що `a` використовується як конкретний тип, бо всі типи у функціях повинні бути конкретними (пам'ятайте, ви не можете написати функцію, що має тип `a -> Maybe`, але можете написати функцію, що має тип `a -> Maybe a` чи тип `Maybe Int -> Maybe String`). Ось чому ми не можемо записати щось на кшталт

```
instance Eq Maybe where
    ...
```

бо, повторюючись, `a` має бути конкретним типом, але `Maybe` не є конкретним типом — це конструктор типу, який приймає один параметр та повертає конкретний тип. Але було б дуже нудно писати `instance Eq (Maybe Int) where`, `instance Eq (Maybe Char) where` і так далі для всіх існуючих в світі типів! Замість цього можна записати це ось як:

```
instance Eq (Maybe m) where
    Just x == Just y   = x == y
    Nothing == Nothing = True
    _ == _             = False
```

Це є те ж саме, що сказати, що ми хочемо зробити всі типи «на зразок» `Maybe <<something>>` членами `Eq`. Насправді ми могли б так і назвати той параметр **«something»** [«щось»], і записати `(Maybe <<something>>)`, але зазвичай, коли ми іменуємо параметри типів, ми надаємо перевагу ім'ям з однієї літери, залишаючись вірними стилю Хаскела. `(Maybe m)` грає роль `a` в `class Eq a where`. У той час як `Maybe` не є конкретним типом, `Maybe m` ним є. Вказавши параметр типу (себто, `m`, мала літера), ми сказали, що хочемо, щоб всі типи на зразок `Maybe m`, де `m` є будь-яким типом, були членами типокласу `Eq`.

Але тут є одна проблемка. Ви бачите її? Ми застосовуємо `==` до «вмісту» `Maybe`, не маючи жодної гарантії, що той вміст є членом `Eq`! Ось чому нам потрібно змінити наше *instance*-означення таким от чином:

```
instance Eq m => Eq (Maybe m) where
    Just x == Just y   = x == y
    Nothing == Nothing = True
    _ == _             = False
```

Нам потрібно було додати умову типокласу! У цьому *instance*-означенні ми кажемо: ми хочемо, щоб всі типи зразка `Maybe m` були членами типокласу `Eq`, але тільки ті типи, в яких `m` (тобто, вміст `Maybe`) є членом `Eq`. Автоматичне втілення виглядало б точнісінько так.

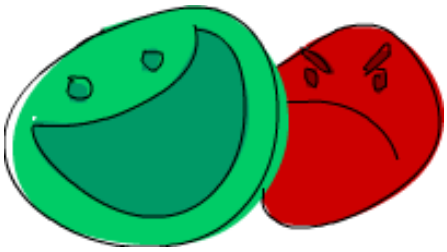
У більшості випадків умови типокласу в *class*-означеннях використовуються для того, щоб зробити типоклас підкласом іншого типокласу, а умови типокласу в *instance*-означеннях використовуюся для того, щоб викласти вимоги до типів, що параметризують тип, для якого пишеться втілення. Наприклад, тут ми вимагали, щоб вміст `Maybe` був членом типокласу `Eq`.

Якщо, втілюючи типоклас, ви бачите, що якийсь тип використовується в оголошенні як конкретний (як, наприклад, `a` в `a -> a -> Bool`), ви маєте подати всі параметри типу, щоб отримався конкретний тип, і взяти усе це діло в дужки.

Примітка: Зверніть увагу, що тип, для якого ви пишете втілення якогось типокласу, замінить параметр у *class*-означенні того типокласу. `a` із `class Eq a where` буде замінено справжнім типом, коли писатиметься втілення, тому намагайтеся подумки розмістити ваш тип також і в оголошеннях типів функцій з відповідного типокласу. `(==) :: Maybe -> Maybe -> Bool` — нісенітниця, тоді як `(==) :: Eq m => Maybe m -> Maybe m -> Bool` можна зрозуміти. Але це — лише інформація для роздумів, не більше, бо `==` завжди матиме тип `(==) :: Eq a => a -> a -> Bool`, незалежно від того, скільки втілень `Eq` ми напишемо.

О, ще одне! Якщо ви хочете побачити всі втілення якогось типокласу, просто спитайте `:info YourTypeClass` у GHCi. Наприклад, `:info Num` спочатку покаже список функцій, яких вимагає типоклас `Num`, а після того — список усіх членів цього типокласу! `:info` також працює із типами і конструкторами типів. Якщо ви запитаєте `:info Maybe`, вам буде показано всі типокласи, які `Maybe` втілює. А ще `:info` хрумає функції і показує їх оголошення типу. Я вважаю, що це — суперкруто.

8.8 Типоклас «так-ні»



У JavaScript-і та деяких інших **слабкотипізованих мовах** [weakly typed languages] ви можете записати практично будь-що у **вираз розгалуження** [if expression]. Наприклад, всі оці речі є дозволеними:

- `if (0) alert("YEAH!") else alert("NO!")`,
- `if ("") alert("YEAH!") else alert("NO!")`,
- `if (false) alert("YEAH!") else alert("NO!")`,

тощо. І в кожному з цих випадків буде викинуто віконце-попередження із `NO!`. Якщо ви виконаєте `if ("WHAT") alert ("YEAH!") else alert("NO!")`, буде викинуто попередження `"YEAH!"`, бо в JavaScript-і непорожні рядки вважаються ближчими до **істинних булевих значень** [boolean values of true] ніж до хибних.

Хоча в Хаскелі краще використовувати в **булевій семантиці** [boolean semantics] виключно булеві значення, спробуймо все одно реалізувати таку JavaScript-ову поведінку. Заради забави! Розпочнімо із *class*-означення.

```
class YesNo a where
    yesno :: a -> Bool
```

Доволі просто. Типоклас `YesNo` оголошує одну функцію. Ця функція приймає одне значення. Потрібно, щоб тип того значення можна було якось пов'язати із поняттям «істинності». І ця функція каже нам вже точно, чи є її аргумент істинним чи хибним. Зверніть увагу, що з того, як ми використовуємо `a` в оголошенні, випливає, що `a` має бути конкретним типом.

Далі означмо деякі втілення. Для чисел ми вважатимемо, що будь-яке число, що не є нулем, є істинним, а нуль — хибним (як і в JavaScript-і).

```
instance YesNo Int where
    yesno 0 = False
    yesno _ = True
```

Порожні списки (а тому і порожні рядки, звичайно ж) є більш хибними ніж істинними, тоді як непорожні списки — більш істинними ніж хибними.

```
instance YesNo [a] where
    yesno [] = False
    yesno _  = True
```

Зверніть увагу на те, як ми «просто, лєгко і нєпрінуждьонно» вставили в список параметр `a`, щоб зробити його конкретним типом, навіть якщо ми не робили ніяких припущень щодо природи типу `a`. Що іще, гм... О! Ідея! Є іще, власне, `Bool` і там і так ясно, яке значення є більш істинним, а яке — менше.

```
instance YesNo Bool where
    yesno = id
```

Га? Що в біса таке те `id`? Це стандартна бібліотечна функція, яка бере параметр і повертає його ж, — себто, вона є якраз тим, що нам тут і потрібно було записати.

Напишімо втілення й для `Maybe a`!

```
instance YesNo (Maybe a) where
    yesno (Just _) = True
    yesno Nothing  = False
```

Знову ж таки, нам не потрібна була тут умова типокласу, бо ми не робили ніяких припущень щодо вмісту `Maybe`. Ми лише сказали, що значення є близьким до істини, якщо воно є `Just`-значення, а близьким до хиби — якщо `Nothing`. Ми все ж повинні були записати `(Maybe a)` замість просто `Maybe`, бо, якщо подумати над цим, функція `Maybe -> Bool` не може існувати (тому що `Maybe` не є конкретним типом), тоді як із `Maybe a -> Bool` все тіп-топ. А найкрутішим наслідком цього є те, що тепер будь-який тип зразка `Maybe <<something>>` є членом типокласу `YesNo` і немає значення, чим є те `<<something>>`.

Раніше ми означили тип `Tree a`, який був реалізацією бінарного дерева пошуку. Ми можемо покласти, що порожнє дерево є хибним значенням, а непорожнє — істинним.

```
instance YesNo (Tree a) where
    yesno EmptyTree = False
    yesno _         = True
```

Чи можна відобразити значення світлофора в множину значень «так-ні»? Звісно. На червоне світло ви зупиняєтеся. На зелене — рухаєтеся. Якщо жовте? Ех, я зазвичай іду на жовте, бо живу задля адреналіну.

```
instance YesNo TrafficLight where
    yesno Red = False
    yesno _   = True
```

Круто, ми понавтілювали собі трохи втілень, — а тепер ходімо пограймося!

```
ghci> yesno $ length []
False
ghci> yesno "haha"
True
ghci> yesno ""
False
ghci> yesno $ Just 0
True
```

```
ghci> yesno True
True
ghci> yesno EmptyTree
False
ghci> yesno []
False
ghci> yesno [0,0,0]
True
ghci> :t yesno
yesno :: YesNo a => a -> Bool
```

Добре, все працює! Створімо функцію, яка імітує **інструкцію розгалуження** [if statement], але працює зі значеннями типу `YesNo`.

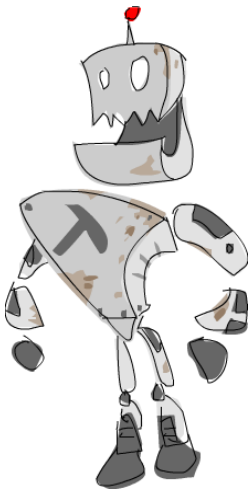
```
yesnoIf :: YesNo y => y -> a -> a -> a
yesnoIf yesnoVal yesResult noResult =
    if yesno yesnoVal then yesResult else noResult
```

Доволі просто. Функція бере **ніби-«так-ні»-якесь значення** [yes-no-ish value] та дві інші речі. Якщо ніби-«так-ні»-якесь значення є схоже на «так», функція повертає першу із двох речей, інакше повертає другу річ.

```
ghci> yesnoIf [] "YEAH!" "NO!"
"NO!"
ghci> yesnoIf [2,3,4] "YEAH!" "NO!"
"YEAH!"
ghci> yesnoIf True "YEAH!" "NO!"
"YEAH!"
ghci> yesnoIf (Just 500) "YEAH!" "NO!"
"YEAH!"
ghci> yesnoIf Nothing "YEAH!" "NO!"
"NO!"
```

8.9 Типоклас Functor

Досі нам траплялося багато типокласів зі стандартної бібліотеки. Ми гралися із `Ord`, який призначений для речей, які можуть бути впорядкованими. Ми потоваришували із `Eq`, який є для речей, які можна перевіряти на рівність. Ми побачили `Show`, який є інтерфейсом для типів, значення яких можуть бути перетворені на рядки. Наш хороший друг, `Read`, завше поруч, коли нам потрібно пере-



творити рядок на значення деякого типу. А тепер ми поглянемо на типоклас `Functor`, який, по суті, є призначений для речей, які можна **відображати** [map over]. Ймовірно, ви зараз думаєте про списки, оскільки **відображення списків** [mapping over lists] є **провідною ідіомою** [dominant idiom] в Хаскелі. І ви праві — **тип списків** [list type] є членом типокласу `Functor`.

Чи є кращий спосіб запізнатися із типокласом `Functor`, ніж подивитися на його реалізацію? Підгляньмо!

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

Гаразд. Ми бачимо, що він означує одну функцію, `fmap`, і не надає їй **реалізації за замовчуванням** [default implementation]. Тип `fmap` є цікавим. Дотепер у означеннях типокласів змінна типу, яка грала роль типу, для якого означується поведінка, іменувала **конкретний тип** [concrete type], як-от `a` у `(==) :: Eq a => a -> a -> Bool`. Але тепер `f` не є конкретним типом (тобто, не є типом, для якого ми можемо побудувати значення, як-от, наприклад, `Int`, `Bool` чи `Maybe String`), а є конструктором типу, який бере один тип-параметр. Короткий приклад-освіжувач пам'яті: `Maybe Int` є конкретним типом, але `Maybe` є конструктором типу, який приймає один тип як параметр. В усякому разі, ми бачимо, що `fmap` приймає як перший параметр функцію, що перетворює один тип на інший, як другий — функтор, який є застосовано до одного типу, а повертає як результат функтор, застосований до іншого типу.

Якщо це звучить спантеличуюче — не хвилюйтеся. Все стане на свої місця незабаром, коли ми розглянемо декілька прикладів. Хмм, це оголошення типу для `fmap` щось мені нагадує. Якщо ви не знаєте, яка сиґнатура типу в функції `map`, ось вона: `map :: (a -> b) -> [a] -> [b]`.

О, як цікаво! `map` приймає функцію, яка перетворює один тип в інший, та список елементів одного типу і повертає список елементів іншого типу. Друзі, здається ми маємо собі функтор! Справді, `map` є просто `fmap`, яка працює лише зі списками. Ось як список втілює типоклас `Functor`.

```
instance Functor [] where
    fmap = map
```

Ось і все! Зауважте, ми не написали `instance Functor [a] where`, бо із `fmap :: (a -> b) -> f a -> f b` бачимо, що `f` має бути конструктором типу, який приймає один тип-параметр. `[a]` уже є конкретним типом (список із елементів типу `a`) тоді, як `[]` є конструктором типів, який приймає один тип і

може будувати нам такі типи як `[Int]`, `[String]` чи навіть `[[String]]`.

Оскільки для списків `fmap`-ом є просто функція `map`, ми отримуємо однакові результати в обох випадках:

```
map :: (a -> b) -> [a] -> [b]
ghci> fmap (*2) [1..3]
[2,4,6]
ghci> map (*2) [1..3]
[2,4,6]
```

Що ж трапиться, якщо ми застосуємо `map` або `fmap` до порожнього списку? Ну звісно ми отримаємо порожній список. Вони просто перетворюють порожній список типу `[a]` в порожній список типу `[b]`.

Типи, які можуть поводитися як **коробки** [boxes], можуть бути функторами. Уявіть собі, що список є коробкою, яка має нескінченну кількість маленьких відділень, всі з яких можуть бути порожніми, або одне-єдине може бути зайнятим, а всі інші порожніми, або ж, якась кількість їх може бути зайнятою, а решта — ні. То що ще може поводитися як коробка? Ну, наприклад, тип `Maybe a`. У певному розумінні, це наче коробка, котра може або не містити нічого, і в такому разі вона має значення `Nothing`, або ж вона може містити лише один елемент, як-от `"HAHA"`, і в цьому випадку вона має значення `Just "HAHA"`. Ось яким чином `Maybe` є функтором.

```
instance Functor Maybe where
    fmap f (Just x) = Just (f x)
    fmap f Nothing  = Nothing
```

Знову ж таки, зауважте, що ми записали `instance Functor Maybe where` замість `instance Functor (Maybe m) where` так само, як ми робили, коли мали справу із `Maybe` та `YesNo`. `Functor` вимагає не конкретний тип, а конструктор типу, який приймає один тип-параметр. Якщо ви подумки замініте `f`-ки на `Maybe`, то подумки побачите, що у цьому конкретному випадку `fmap` діятиме як `(a -> b) -> Maybe a -> Maybe b`, і ця типосиґнатура є цілком «нормальною». Але якщо ви замініте `f`-ки на `(Maybe m)`, то здаватиметься, що функція `fmap` діятиме як `(a -> b) -> Maybe m a -> Maybe m b`, що не має, чорт забирай, анінайменшого сенсу, бо `Maybe` приймає лише один тип-параметр!

В усякому разі, реалізація `fmap` є доволі простою. На порожнє значення `Nothing` повертається `Nothing`. Якщо ми **відображаємо** [map over] порожню коробку, то отримуємо порожню коробку. Це має сенс. Так само, як із списками — там, якщо ми відображаємо порожній список, ми отримуємо порожній список. Якщо ж ми маємо не порожнє значення, а одне значення, «запакова-

не» у `Just`, тоді ми застосовуємо функцію до вмісту `Just`.

```
ghci> fmap (++ " HEY GUYS IM INSIDE THE JUST") (Just "Something serious.")
Just "Something serious. HEY GUYS IM INSIDE THE JUST"
ghci> fmap (++ " HEY GUYS IM INSIDE THE JUST") Nothing
Nothing
ghci> fmap (*2) (Just 200)
Just 400
ghci> fmap (*2) Nothing
Nothing
```

Ще одна річ, яку можна відображати і яка може втілити `Functor`, — це наш тип `Tree a`. Його теж, з натяжкою, можна уявити як коробку (містить кілька значень або жодного значення), а конструктор типу `Tree` приймає рівно один тип-параметр. Якщо ви поглянете на `fmap` так, нібито ця функція була створена лише для роботи із `Tree`, її сиґнатура типу виглядатиме як `(a -> b) -> Tree a -> Tree b`. У цьому прикладі ми скористаємося рекурсією. Порожнє дерево відображатиметься у порожнє дерево. Відображення непорожнього дерева буде деревом, яке складатиметься із (а) кореня дерева, що на вході, але із значенням, до якого було застосовано нашу функцію, і (б) лівого і правого піддерев, які будуть ті ж самі, що й в кореня дерева на вході, от лише їх буде відображено за допомогою нашої функції.

```
instance Functor Tree where
    fmap f EmptyTree                = EmptyTree
    fmap f (Node x leftsub rightsub) =
        Node (f x) (fmap f leftsub) (fmap f rightsub)
```

```
ghci> fmap (*2) EmptyTree
EmptyTree
ghci> fmap (*4) (foldr treeInsert EmptyTree [5,7,3,2,1,7])
Node 28 (Node 4 EmptyTree (Node 8 EmptyTree (Node 12 EmptyTree
 (Node 20 EmptyTree EmptyTree)))) EmptyTree
```

Файно! А як щодо `Either a b`? Чи може це стати функтором? Типоклас `Functor` вимагає конструктора типу, який приймає лише один тип-параметр, але `Either` приймає аж два. Гммм... Придумав! Ми частково застосуємо `Either`, згодувавши йому один із параметрів, а другий лишивши «вільним». Ось як `Either a` є функтором у стандартній бібліотеці:

```
instance Functor (Either a) where
    fmap f (Right x) = Right (f x)
    fmap f (Left x)  = Left x
```

Так-так, і що ж це ми тут напрограмували? Як бачимо, ми пишемо втілення для `Either a`, а не для `Either`. Це тому, що `Either a` є конструктором типу, який приймає один параметр, тоді як `Either` приймає два. Якщо б `fmap` працювала виключно із `Either a`, сиґнатура її типу була б `(b -> c) -> Either a b -> Either a c` (і це те ж саме, що й `(b -> c) -> (Either a) b -> (Either a) c`). У цій реалізації, ми відображаємо у випадку `Right`, і не відображаємо у випадку `Left`. Чому так? Що ж, якщо ми згадаємо, як було означено тип `Either a b`, то пригадається щось таке:

```
data Either a b = Left a | Right b
```

Справді, якби ми хотіли застосувати одну і ту ж функцію до них обох, `a` та `b` повинні були б мати той самий тип. Тобто, якщо б ми спробували відобразити за допомогою функції, яка приймає рядок і повертає рядок, а параметр `b` був би рядком, а параметр `a` був би числом, то це не спрацювало б. Знову ж таки, якщо уявити, що `fmap` оперує лише значеннями `Either`, то можна побачити з тієї уявної типосиґнатури, що перший параметр повинен залишатися незмінним, тоді як другий може змінюватися, і ось як раз той перший незмінний параметр і фігурує в конструкторі значень `Left`.

Це також гарно вписується в нашу **коробкову аналогію** [box analogy], якщо ми уявимо частину `Left` як порожню коробку із повідомленням про помилку, яке каже нам, чому ця коробка порожня, і яке є написаним на боці тієї коробки.

Мапи із модуля `Data.Map` теж можна зробити функторами, оскільки вони містять (чи не містять!) значення. У випадку `Map k v` функція `fmap` відобразить за допомогою функції `v -> v'` по мапі типу `Map k v` та поверне мапу типу `Map k v'`.

Примітка: Зауважте, що в типах `'` ніяк семантично не навантажений, так само, як і в назвах значень. Його зазвичай додають до назви, як от в `v'`, і розуміється, що `v'` — тип схожий на `v`, але трішки змінений.

Спробуйте зрозуміти самостійно, як `Map k` втілює `Functor`!

Попрацювавши із типокласом `Functor`, ми побачили, як саме типокласи можуть **представляти** [to represent] доволі круті концепції з програмування **функціями вищого порядку** [higher-order functions]. Ми також трохи попрактикувалися писати втілення і частково застосовувати конструктори типів. У одному з наступних розділів ми також розглянемо декілька законів, які справджуються для функторів.

І ще одне! Функтори задовольняють декільком законам, завдяки чому в них є декілька властивостей, на які ми можемо спиратися і не замислюватися забагато в роботі із ними. Якщо ми застосовуємо `fmap (+1)` до списку `[1,2,3,4]`, ми очікуємо результат `[2,3,4,5]`, а не його розвернену версію `[5,4,3,2]`. Якщо ми застосовуємо `fmap (\a -> a)` (тотожне перетворення, яке повертає просто свій єдиний параметр, без змін) до якогось списку, ми сподіваємося отримати той самий список. Наприклад, якщо б ми неправильно написали втілення функтора для нашого типу `Tree`, застосування `fmap` до дерева, у якого ліве піддерево вузла має лише елементи, які менші за значення елемента з того вузла, а праве піддерево — ті, які є більші, могло б повернути дерево, де вже немає такого впорядкування. Ми розглянемо закони функторів детальніше у одному із наступних розділів.

8.10 Кшталти та бойове мистецтво володіння типами

Конструктори типів приймають інші типи як параметри і врешті-решт[†] будують нам конкретні типи. Це трохи нагадує мені функції — функції приймають значення як параметри і врешті-решт повертають (будують) якісь інші значення. Ми побачили, що конструктори типів можуть бути частково застосовані (`Either String` є конструктором, який приймає один тип-параметр і повертає конкретний тип, такий як, наприклад, `Either String Int`) — і функції теж можуть. Це все є дуже і дуже цікаво. В цьому підрозділі ми більш формально розглянемо те, яким чином конструктори типів застосовуються до типів-параметрів, так само як ми формально означували (за допомогою оголошень типів), як функції застосовуються до значень.

[†]Мається на увазі, що конкретний тип отримається лише коли всі типи-параметри буде подано, а в «проміжних станах» матимемо частково застосовані конструктори типів.

Вам не обов'язково треба продертися крізь цей підрозділ, щоб продовжити вашу магічну подорож Хаскелом, і якщо ви його не зрозумієте, не переживайте. Однак, якщо ви таки опануєте цей розділ, це допоможе вам більш глибоко зрозуміти, як працює в Хаскелі система типів.

Отже, такі значення як `3`, `"YEAH"` та `takeWhile` (функції є також значеннями, бо ми можемо передавати їх функціям як аргументи, і таке подібне) мають кожне свій тип. Типи є маленькими ярличками, і такий ярличок є в кожного значення, і це дозволяє нам **формально міркувати** [to reason] про значення. Але типи мають свої власні маленькі ярлички, які звуться **кшталтами**. Кшталт є таким собі «типом» самого типу. Це може звучати трішки дивно і заплутано, але насправді це є крута концепція.

Що таке **кшталти** [kinds] і для чого вони годяться? Що ж, давайте перевіряти кшталти типів у GHCi за допомогою `:k`.

```
ghci> :k Int
Int :: *
```

Зірочка-сніжинка? Як незвично. Що ж це означає? `*` значить, що тип є конкретним типом. Конкретний тип є типом, який не приймає жодних типів-параметрів, а значення можуть бути лише в конкретних типів. Якщо б мені потрібно було вголос прочитати `*` (дотепер не треба було), я б сказав *зірочка* або просто *тип*.

Оукей, а зараз погляньмо, який кшталт в `Maybe`.

```
ghci> :k Maybe
Maybe :: * -> *
```

Конструктор типу `Maybe` приймає один конкретний тип (як-от `Int`) та повертає конкретний тип, такий як, наприклад, `Maybe Int`. І це все, що нам каже цей кшталт. Так само як `Int -> Int` означає, що функція приймає `Int` та повертає `Int`, кшталт `* -> *` означає, що це — конструктор типу, який приймає один конкретний тип і повертає конкретний тип. Застосуймо `Maybe` до якогось типу і гляньмо, яким є кшталт результуючого типу.

```
ghci> :k Maybe Int
Maybe Int :: *
```

Саме те, чого я й очікував! Ми подали тип-параметр до `Maybe` і отримали конкретний тип (це як раз і означає те `* -> *`). Паралеллю (хоча й не еквівалентом — типи і кшталти є різними поняттями) до цього буде виконання `:t isUpper` та `:t isUpper 'A'`. Функція `isUpper` має тип `Char -> Bool`, а `isUpper 'A'` має тип `Bool`, оскільки його значення-результат є `True`. Проте, обидва ці типи мають кшталт `*`.

Ми застосували `:k` до типу, щоб одержати його кшталт, точно так само, як ми можемо застосувати `:t` до значення, щоб отримати його тип. Як ми сказали, типи є ярличками значень, а кшталти є ярличками типів, і між ними можна провести певні паралелі.

Розгляньмо інший кшталт.

```
ghci> :k Either
Either :: * -> * -> *
```

Ага, це каже нам, що `Either` приймає два конкретні типи як типи-параметри, щоб утворити конкретний тип. Це також нагадує оголошення функції, яке приймає два значення і щось повертає. Конструктори типів є карійованими (так само, як і функції), тому ми можемо їх частково застосовувати.

```
ghci> :k Either String
Either String :: * -> *
ghci> :k Either String Int
Either String Int :: *
```

Коли б ми хотіли зробити `Either` членом типокласу `Functor`, ми б повинні були частково застосувати його, оскільки `Functor` вимагає типи, які приймають єдиний параметр, тоді як `Either` приймає два. Іншими словами, `Functor` вимагає типи кшталту `* -> *` і таким чином нам би довелося частково застосувати `Either`, щоб отримати тип кшталту `* -> *` замість його початкового кшталту `* -> * -> *`. Якщо ми поглянемо на означення `Functor` ще раз

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

то побачимо, що змінна типу `f` використовується як тип, який приймає один конкретний тип, щоб побудувати конкретний тип. Ми знаємо, він повинен побудувати конкретний тип, бо він використовується як тип значення у функції. Звідси ми можемо зробити висновок, що типи, які хочуть товаришувати із `Functor`-ом, повинні бути кшталту `* -> *`.

А зараз ми повправляємося у бойовому мистецтві володіння типами. Розгляньмо цей типоклас, який я придумаю «на ходу»:

```
class Tofu t where
    tofu :: j a -> t a j
```

Це виглядає по-справжньому чудернацько. Якби було треба, як ми могли б створити тип, який міг би втілити такий дивний типоклас? Ну, спочатку погляньмо на те, якого кшталту він повинен був би бути. Через те, що `j a` використовується як тип значення, яке функція `tofu` приймає як параметр, `j a`

повинен мати кшталт `*`. Вважатимемо, що `a` є `*`, і з цього випливає, що `j` повинен мати кшталт `* -> *`. Бачимо далі, що конструктор `t` теж повинен збудувати нам конкретне значення і він приймає два типи. Знаючи, що `a` має кшталт `*`, а `j` має кшталт `* -> *`, ми виводимо, що `t` мусить мати кшталт `* -> (* -> *) -> *`. Таким чином, він (а) приймає (1) конкретний тип (`a`) і (2) конструктор типу (`j`), який приймає один конкретний тип, і (б) будує конкретний тип. Круто.

Добре, сотворімо тип, що має кшталт `* -> (* -> *) -> *`. Ось один спосіб, як це можна зробити.

```
data Frank a b = Frank {frankField :: b a} deriving (Show)
```

Як переконатися, що цей тип має кшталт `* -> (* -> *) - > *`? Що ж, поля в алгебраїчних типах даних можуть тримати лише значення, і тому вони, вочевидь, повинні бути типів, що мають кшталти `*`. Отож, `a` то є просто `*`, і з цього (і з означення `Frank`) випливає, що `b` приймає один параметр типу, і таким чином кшталтом `b` є `* -> *`. Тепер ми знаємо кшталти обидвох `a` та `b` і через те, що вони є параметрами конструктора типів `Frank`, ми бачимо, що цей конструктор типів має кшталт `* -> (* -> *) -> *`. Перша `*` відповідає `a`, а `(* -> *)` — `b`. Створімо декілька значень такого типу за допомогою конструктору значень `Frank`, і перевіримо, який тип має результат, на всяк випадок.

```
ghci> :t Frank {frankField = Just "HAHA"}
Frank {frankField = Just "HAHA"} :: Frank [Char] Maybe
ghci> :t Frank {frankField = Node 'a' EmptyTree EmptyTree}
Frank {frankField = Node 'a' EmptyTree EmptyTree} :: Frank Char Tree
ghci> :t Frank {frankField = "YEAH"}
Frank {frankField = "YEAH"} :: Frank Char []
```

Чому так? Оскільки `frankField` має тип `a b`, його можна ініціалізувати лише значеннями, які мають «схожий по формі» тип. Тобто, це може бути `Just "HAHA"`, яке має тип `Maybe [Char]`, або ж це може бути значення `['Y','E','S']`, яке має тип `[Char]` (якщо б ми використали наш власний списковий тип для цього, то був би тип `List Char`). І ми бачимо — типи, що їх мають значення типу `Frank`, відповідають кшталту, що його має тип `Frank`. `[Char]` має кшталт `*`, а `Maybe` має кшталт `* -> *`. Для того, щоб можна було побудувати значення якогось типу, це має бути конкретний тип, і тому, якщо маємо справу із конструктором типів, він має бути повністю застосованим. Ось чому кожне значення `Frank blah blaah` має кшталт `*`.

Втілювати `Tofu` в `Frank` доволі просто. Ми бачимо, що `tofu` приймає `j a`

(прикладом типу «такої форми» міг би бути `Maybe Int`) та повертає `t a j`. Тому, якщо ми замінимо `t` на `Frank`, тип результату (для прикладу, наведеного раніше, де `j a` є `Maybe Int`) буде `Frank Int Maybe`.

```
instance Tofu Frank where
    tofu x = Frank x
```

```
ghci> tofu (Just 'a') :: Frank Char Maybe
Frank {frankField = Just 'a'}
ghci> tofu ["HELLO"] :: Frank [Char] []
Frank {frankField = ["HELLO"]}
```

Не надто корисно, але ми принаймні розім'яли м'язи. А тепер — вйо до власне тренування! Маємо такий тип даних:

```
data Barry t k p = Barry { yabba :: p, dabba :: t k }
```

А зараз ми хочемо зробити його втіленням типокласу `Functor`. `Functor` хоче типи кшталту `* -> *`, але не схоже, що `Barry` має такий кшталт. Яким є кшталт `Barry`? Ну, бачимо, що `Barry` приймає три типи-параметри, отже, це буде `<<something>> -><<something>> -><<something>> ->*`. Особливо не напружуючись, бачимо, що `p` є конкретним типом і таким чином має кшталт `*`. Для `k` можна припустити кшталт `*`, і з цього випливає, що `t` повинен мати кшталт `* -> *`. Тепер просто замінимо ті *«something»*-заглушки на отримані кшталти, щоб отримати для `Barry` кшталт `(* -> *) -> * -> * -> *`. Перевіримо цей результат в GHCi.

```
ghci> :k Barry
Barry :: (* -> *) -> * -> * -> *
```

Ба! Ми були праві. Як приємно. Тепер, щоб зробити цей тип членом типокласу `Functor`, ми повинні частково застосувати перші два типи-параметри, щоб отримався кшталт `* -> *`. Це значить, що початком означення втілення буде `instance Functor (Barry a b) where`. Якщо ми, за звичкою, поглянемо на `fmap` так, наче вона була придумана саме для `Barry`, то ця функція матиме тип `fmap :: (a -> b) -> Barry c d a -> Barry c d b` (отримано простою заміною `f` з означення `Functor`-а на `Barry c d`. Третій тип-параметр в `Barry` буде мінятися, і як бачимо, він затишно оселився у своєму власному окремому полі.

```
instance Functor (Barry a b) where
    fmap f (Barry {yabba = x, dabba = y}) = Barry {yabba = f x, dabba = y}
```

Ось! Ми просто застосували `f` до першого поля.

Озирнемося на щойно пройдений шлях. У цьому підрозділі ми детально розглянули, як працюють параметри типів, і зробили щось на кшталт формалізації цього процесу за допомогою кшталтів, аналогічно до того, як ми формалізували параметри функцій за допомогою типосиґнатур. Ми також побачили, що можна накреслити певні паралелі між функціями та конструкторами типів. Проте вони є зовсім різні. В реальному житті вам наврядче доведеться отак ковбаситися із кшталтами і виводити кшталти вручну. Швидше за все, вам просто потрібно буде частково застосувати якийсь ваш власний тип і отримати `* -> *` чи `*`, щоб можна було втілити якийсь зі стандартних типокласів. Однак непогано знати, як і чому це насправді працює. Було цікаво дізнатися, що типи самі по собі мають свої власні маленькі «типики» — кшталти.

Знову ж таки, не обов'язково зрозуміти все, що ми щойно тут робили, аби продовжувати успішно читати решту цієї книги — але якщо ви таки зрозуміли як працюють кшталти, то скоріш за все, ви тепер добре розбираєтеся у системі типів мови Хаскел.

Розділ 10

Функційне розв'язання задач

Переклад українською Ганни Лелів

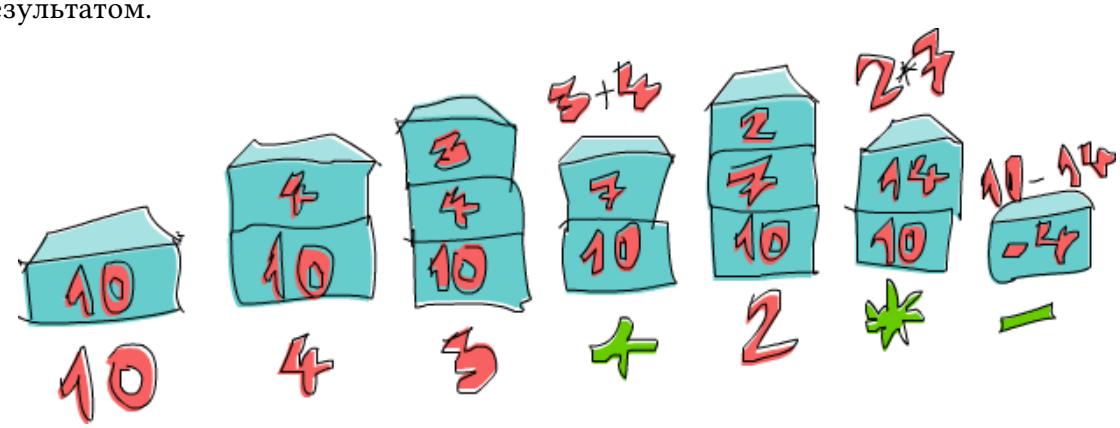
У цьому розділі ми розглянемо кілька цікавих завдань і поміркуємо функційно, аби розв'язати їх якомога елегантніше. Ми не будемо вводити ніяких нових концепцій, а просто пограємо трохи новою хаскельною мускулатурою, яку ми натренували раніше, та відшліфовуватимемо наші набуті навички кодування. У кожному 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 6.5 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 6.7), 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
```

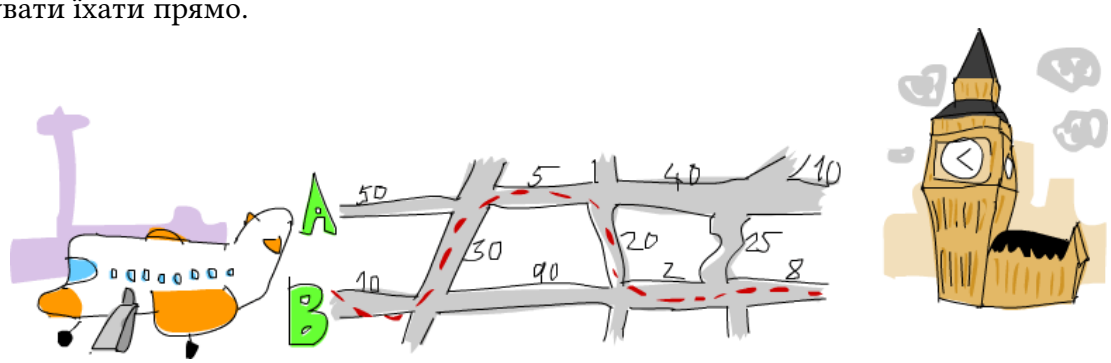
Я гадаю, що написати функцію з 10 рядочків тексту, яка вираховує ЗПЗ-вирази з чисел із плаваючою комою і яку можна легко модифікувати і підтримувати, — це круто.

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.

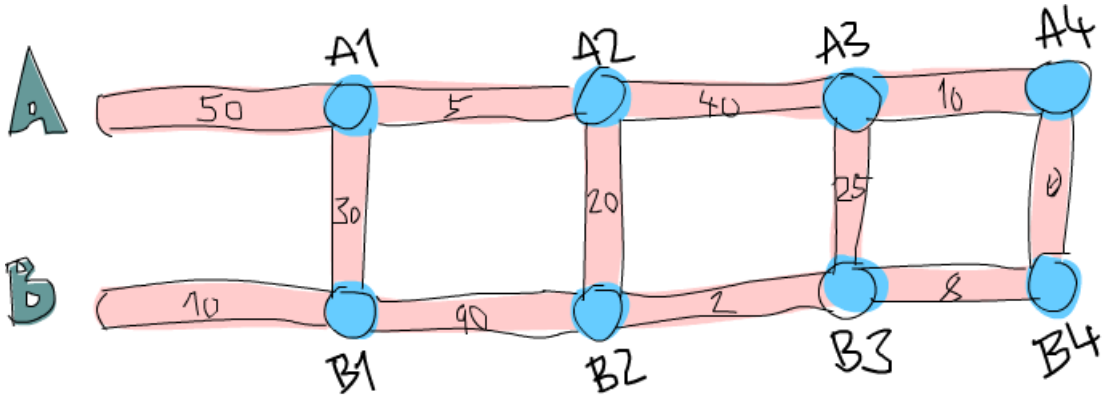
Спробуймо розв'язати це завдання за три кроки, як задачу про калькулятор зворотного польського запису:

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`).

Розділ 14

Застібки-блискавки

Переклад українською Семена Тригубенка

Той факт, що Хаскел є чистофункційною мовою, має купу переваг, і водночас змушує нас розв'язувати задачі не так, як у нечистофункційних мовах. Завдяки прозорості посилань не можна розрізнити два значення, якщо вони означають одне й те саме.

Отже, якщо ми маємо дерево повне п'ятірок (а чому б і ні?) і хочемо поміняти одну з них на шістку, нам треба якимось чином достеменно знати, яку саме п'ятірку в нашому дереві ми хочемо змінити. Нам треба знати, де вона є у нашому дереві. У нечистофункційних мовах ми могли б просто занотувати, де саме в пам'яті та п'ятірка розташована і поміняти те місце. Але в Хаскелі одна п'ятірка нічим не гірша за іншу, і ми не можемо дискримінувати її за місцем проживання у пам'яті комп'ютера. Ми власне нічого не можемо *змінити*; коли ми кажемо, що міняємо дерево, ми маємо на увазі, що отримуємо одне дерево, а повертаємо — інше, не таке як на вході, але схоже.

Можна зробити ось так: запам'ятовувати шлях від кореня дерева до вузла, який ми хочемо змінити. Скажімо, візьміть оце дерево, йдіть ліворуч, далі — праворуч, а потім — знову ліворуч і поміняйте елемент отам. Так, цей підхід працюватиме, але може бути трохи неефективним. Якщо ми захочемо замінити якийсь вузол поблизу того вузла, який ми щойно змінили, нам треба гуляти тим деревом знову від кореня й до потрібного місця!

У цьому підрозділі ми побачимо, як можна взяти структуру даних і зосередитися на якійсь її частині так, щоб можна було легко змінювати значення по вузлах і ефективно цими вузлами пересуватися. Круто!

14.1 На прогулянці

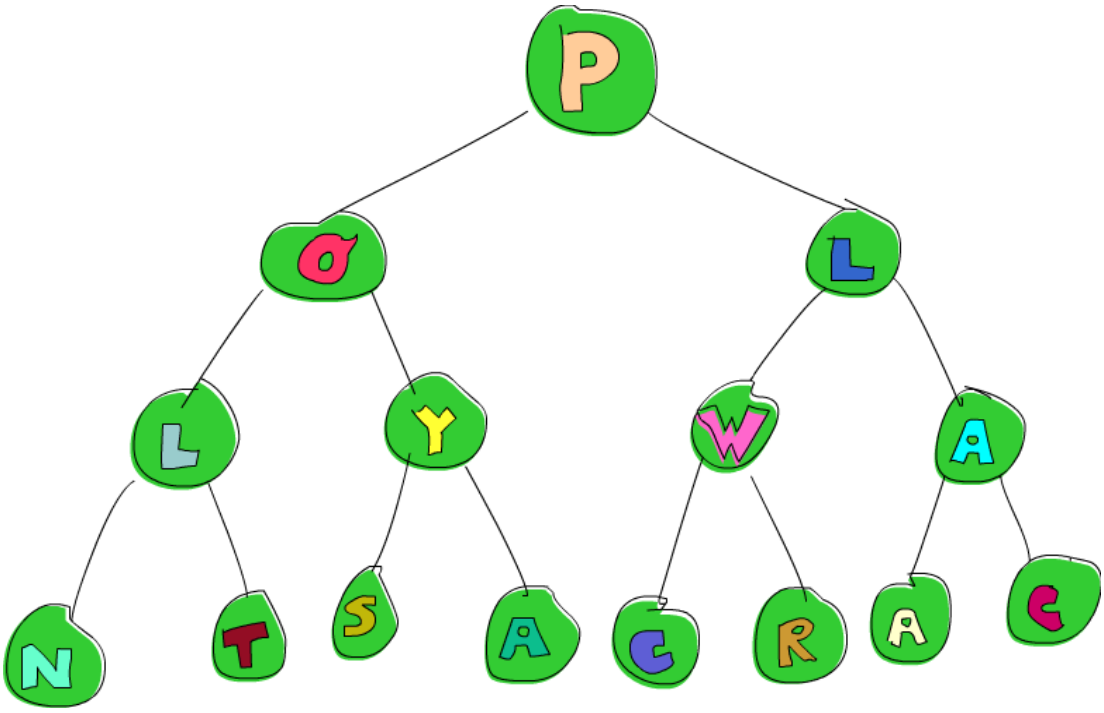
Колись на уроках біології ми вчили, що є багацько різних дерев, тому візьмемо насіння, з якого виросте наше дерево. Ось воно:

```
data Tree a = Empty | Node a (Tree a) (Tree a) deriving (Show)
```

Отже, наше дерево або пусте, або має вузол, що містить елемент-значення і пару піддерев. Ось файний приклад такого дерева, і я даю тобі його, О Мій Читачу, безкоштовно!

```
freeTree :: Tree Char
freeTree =
    Node 'P'
        (Node 'O'
            (Node 'L'
                (Node 'N' Empty Empty)
                (Node 'T' Empty Empty)
            )
            (Node 'Y'
                (Node 'S' Empty Empty)
                (Node 'A' Empty Empty)
            )
        )
        (Node 'L'
            (Node 'W'
                (Node 'C' Empty Empty)
                (Node 'R' Empty Empty)
            )
            (Node 'A'
                (Node 'A' Empty Empty)
                (Node 'C' Empty Empty)
            )
        )
```

А ось тут це дерево представлене графічно:



Зверніть увагу на `W` у тому дереві. Наприклад, якщо ми хочемо поміняти її на `P`. Як то можна зробити? Ну, можна, наприклад, зіставляти взірці по тому дереву, аж доки ми не знайдемо той елемент, що розташовано за адресою: спочатку праворуч, потім ліворуч. Ось потрібний нам код:

```
changeToP :: Tree Char -> Tree Char
changeToP (Node x l (Node y (Node _ m n) r)) = Node x l (Node y (Node 'P' m n) r)
```

Тьху. Виглядає бридко і неохайно, та ще й трохи заплутано. Що тут відбувається? Ми зіставляємо взірець із нашим деревом і називаємо корінь `x` (`x` звертається до `'P'` у корені) і його ліве піддерево — `l`. Замість того, щоб дати ім'я правому піддереву, ми продовжуємо зіставляти по ньому. І продовжуємо в тому дусі аж до того моменту, як дійдемо до піддерева із коренем, що містить наше рідне `'W'`. Як то буде зроблено, ми перебудовуємо наше дерево, але з новим піддеревом, що містить `'W'` у корені замість колишнього `'P'`.

Чи є кращий спосіб? Може створімо функцію, що братиме дерево і путівник по ньому? Путівник міститиме вказівки типу `L` чи `R`, що означатимуть ліворуч і праворуч, відповідно, і ми замінюватимемо елемент у вузлі, до якого потрапляємо, керуючись тими вказівками. Ось воно:

```
data Direction = L | R deriving (Show)
type Directions = [Direction]

changeToP :: Directions-> Tree Char -> Tree Char
changeToP (L:ds) (Node x l r) = Node x (changeToP ds l) r
```

```
changeToP (R:ds) (Node x l r) = Node x l (changeToP ds r)
changeToP [] (Node _ l r) = Node 'P' l r
```

Якщо перший елемент у списку вказівок є `L`, ми будуємо нове дерево точнісінько так, як старе, тільки елемент у його лівому піддереві зміниться на `'P'`. Коли ми рекурсивно викликаємо `changeToP`, ми подаємо як параметри лише хвіст списку вказівок, бо ми вже прислухалися до однієї з них, коли пішли ліворуч. Ми робимо так само у випадку `R`. Якщо список директив пустий, це означає що ми прибули в точку призначення, тому ми повертаємо дерево на кшталт того, що було подано, от лише `'P'` тепер є елементом вузла-кореня того дерева.

Щоб не друкувати усе дерево, створимо функцію, що бере наш список вказівок-директив і каже, який елемент «мешкає» за тією адресою:

```
elemAt :: Directions -> Tree a -> a
elemAt (L:ds) (Node _ l _) = elemAt ds l
elemAt (R:ds) (Node _ _ r) = elemAt ds r
elemAt [] (Node x _ _) = x
```

Ця функція насправді є дуже схожою на `changeToP`, от лише замість запам'ятовування по ходу справи і реконструювання дерева в такий спосіб, вона ігнорує все окрім пункту її призначення. Ось тут ми міняємо `'W'` на `'P'` і бачимо, чи містить те нове дерево наші зміни:

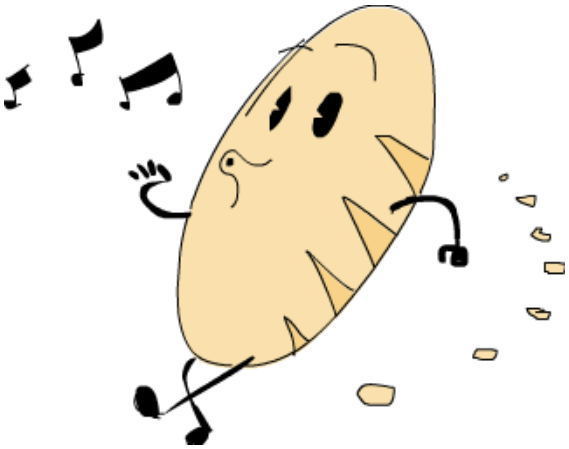
```
ghci> let newTree = changeToP [R,L] freeTree
ghci> elemAt [R,L] newTree
'P'
```

Кльово, здається що спрацювало. У цих функціях список вказівок грає роль *фокусу*, тому що він «фокусує» нашу увагу на потрібному піддереві нашого дерева. Наприклад, список вказівок `[R]` фокусує нашу увагу на піддереві праворуч від кореня. Пустий список вказівок зосереджує нашу увагу на всьому дереві.

Хоча така техніка виглядає модною, вона може бути неефективною, особливо коли ми хочемо міняти елементи один за іншим, багато разів. Хай у нас є величезна тополя і довгий, як жердяка, список вказівок, що веде нас до якогось елемента внизу того дерева. Ми прислуха́ємося до тих вказівок, спускаємося тим деревом і міняємо елемент внизу. Якщо ми хочемо змінити елемент неподалік, маємо починати з коріння і йти знайомим шляхом донизу — увесь шлях, знову. Воловодіння безконечне.

У наступному підрозділі ми знайдемо кращий шлях. Будемо фокусуватися на піддереві, але у спосіб, що даватиме змогу ефективно перемикатися на піддерева неподалік.

14.2 Хлібні крихти по стежці



Добре, — аби сфокусуватися на піддереві, ми хочемо щось ліпше, ніж просто список вказівок, яких ми весь час дотримуємося, починаючи рух від кореня нашого дерева. А чи не буде ліпше, якщо ми почнемо в корені і рухатимемося ліворуч чи праворуч, один крок за одну операцію, і лишатимемо щось на кшталт хлібних крихт? Тобто, коли йдемо ліворуч, пам'ятатимемо, що ми пішли ліворуч, а коли праворуч — то праворуч, відповідно. Звичайно, таке можна спробувати.

Представлення у програмі для такої навігаційної стежки буде схожим на список, що складатиметься з `Direction` (можливі варіанти — `L` чи `R`), от лише не називатимемо його `Directions`. Краще назвемо його `Breadcrumbs`, бо тепер наші вказівки буде подано у зворотньому порядку, оскільки ми залишатимемо їх під час подорожі тим деревом:

```
type Breadcrumbs = [Direction]
```

Ось функція, що бере дерево і стежку і рухається ліворуч, додаючи `L` у голову списку, який представляє стежку:

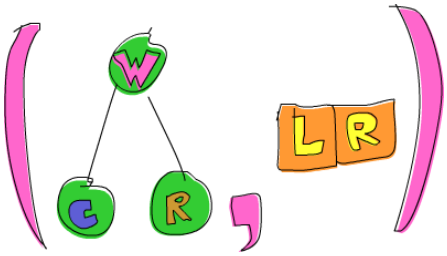
```
goLeft :: (Tree a, Breadcrumbs) -> (Tree a, Breadcrumbs)
goLeft (Node _ l _, bs) = (l, L:bs)
```

Ми ігноруємо елементи в корені і правому піддереві і просто повертаємо ліве піддерево разом із старими крихтами плюс `L` у голові. Ось функція для ходіння праворуч:

```
goRight :: (Tree a, Breadcrumbs) -> (Tree a, Breadcrumbs)
goRight (Node _ _ r, bs) = (r, R:bs)
```

Вона працює так само. Використаймо ці функції і підемо нашим деревом `freeTree` праворуч, а потім ліворуч:

```
ghci> goLeft (goRight (freeTree, []))
(Node 'W' (Node 'C' Empty Empty) (Node 'R' Empty Empty),[L,R])
```



Окей, то тепер ми маємо дерево із `'W'` у корені і `'C'` у корені його лівого піддерева, а `'R'` — у корені правого. Навігаційна стежка є `[L,R]`, оскільки ми пішли спочатку праворуч, а потім — ліворуч.

Щоб зробити прогулянку деревом зрозумілішою, ми можемо задіяти функцію `-:`, що працюватиме ось як:

```
x -: f = f x
```

Це дасть нам змогу викликати функції у новий спосіб, де спочатку ми пишемо аргумент, а потім `-:` і вже після того ім'я функції. Тобто замість `goRight (freeTree, [])`, ми можемо тепер написати `(freeTree, []) -: goRight`. Використовуючи це, ми перепишемо вищенаведене як вираз, із якого легше бачити, що ми йдемо спочатку праворуч, а потім ліворуч:

```
ghci> (freeTree, []) -: goRight -: goLeft
(Node 'W' (Node 'C' Empty Empty) (Node 'R' Empty Empty),[L,R])
```

14.2.1 Назад нагору

А що, якщо ми тепер захочемо піти назад наверх нашим деревом? Із нашої стежки хлібних крихт ми знаємо, що поточне дерево є лівим піддеревом батьківського дерева, а воно, в свою чергу, є правим піддеревом свого, і це все. У нас немає достатньо інформації про батьківське дерево, щоб можна було на нього назад залізти. Здається, що окрім напрямку руху, хлібна крихта має ще й містити решту інформації, потрібної для зворотнього ходу. У цьому випадку, це є елемент з вузла-кореня батьківського дерева разом із його правим піддеревом.

Загалом, одна хлібна крихта має містити усі дані потрібні для відбудови батьківського вузла. Тому там має бути інформація про всі шляхи, якими ми не пішли, і має також бути відомо про напрям, у якому ми пішли. А от інформація про піддерево, на якому ми сфокусувалися, нам не потрібна. Адже ми вже маємо те піддерево як першу компоненту кортежу, тому якби вона також була в хлібній крихті, ми б зберігали цю інформацію двічі.

Модифікуймо наші крихти, щоб вони також містили інформацію про все, чим ми знехтували, коли рухалися ліворуч чи праворуч. Замість `Direction`, ми побудуємо новий тип даних:

```
data Crumb a = LeftCrumb a (Tree a)
```

```
              | RightCrumb a (Tree a) deriving (Show)
```

Тепер замість простого `L` маємо `LeftCrumb`, що містить елемент вузла, з якого ми прийшли, і праве піддерево, яке ми не відвідали. Замість `R` — маємо `RightCrumb`, що містить елемент з вузла, з якого ми прийшли, і ліве піддерево, куди ми не завітали.

Ця навігаційна хрустка смакота тепер містить усе що треба, щоб відбудувати дерево, яким ми спустилися. Тому замість хлібних крихт ми тепер користуємося чимось на кшталт дискет, які ми залишаємо на нашому шляху, і які містять набагато більше інформації порівняно з просто напрямом, який ми обрали.

По суті, кожна крихта тепер є деревом із діркою. Коли ми заглиблюємося у дерево, хлібокрихта зберігає усю інформацію, яку зберігав вузол, із якого ми прийшли, *за винятком* піддерева, яке ми вибрали для фокусування. До того ж, місце з діркою позначено. У випадку `LeftCrumb`, ми знаємо що ми рушили ліворуч, тому піддерево, якого не вистачає, є лівим піддеревом.

Змінимо також наш синонімічний тип `Breadcrumbs`, щоб скористатися нашими нововведеннями:

```
type Breadcrumbs a = [Crumb a]
```

Рухаючись нагору, нам треба оновити функції `goLeft` і `goRight` і зберігати інформацію про шляхи, якими ми не пішли, а не просто знехтувати ними, як ми робили раніше. Ось, `goLeft`:

```
goLeft :: (Tree a, Breadcrumbs a) -> (Tree a, Breadcrumbs a)
goLeft (Node x l r, bs) = (l, LeftCrumb x r:bs)
```

Видно, що вона дуже схожа на її попередника `goLeft`, от лише замість простого додавання `L` у голову навігаційної стежки, ми додаємо `LeftCrumb`, щоб зазначити, що ми пішли ліворуч, і навантажуємо нашу `LeftCrumb` елементом вузла, з якого подорожуємо (це є `x`), й правим піддеревом, яке ми вирішили не відвідувати.

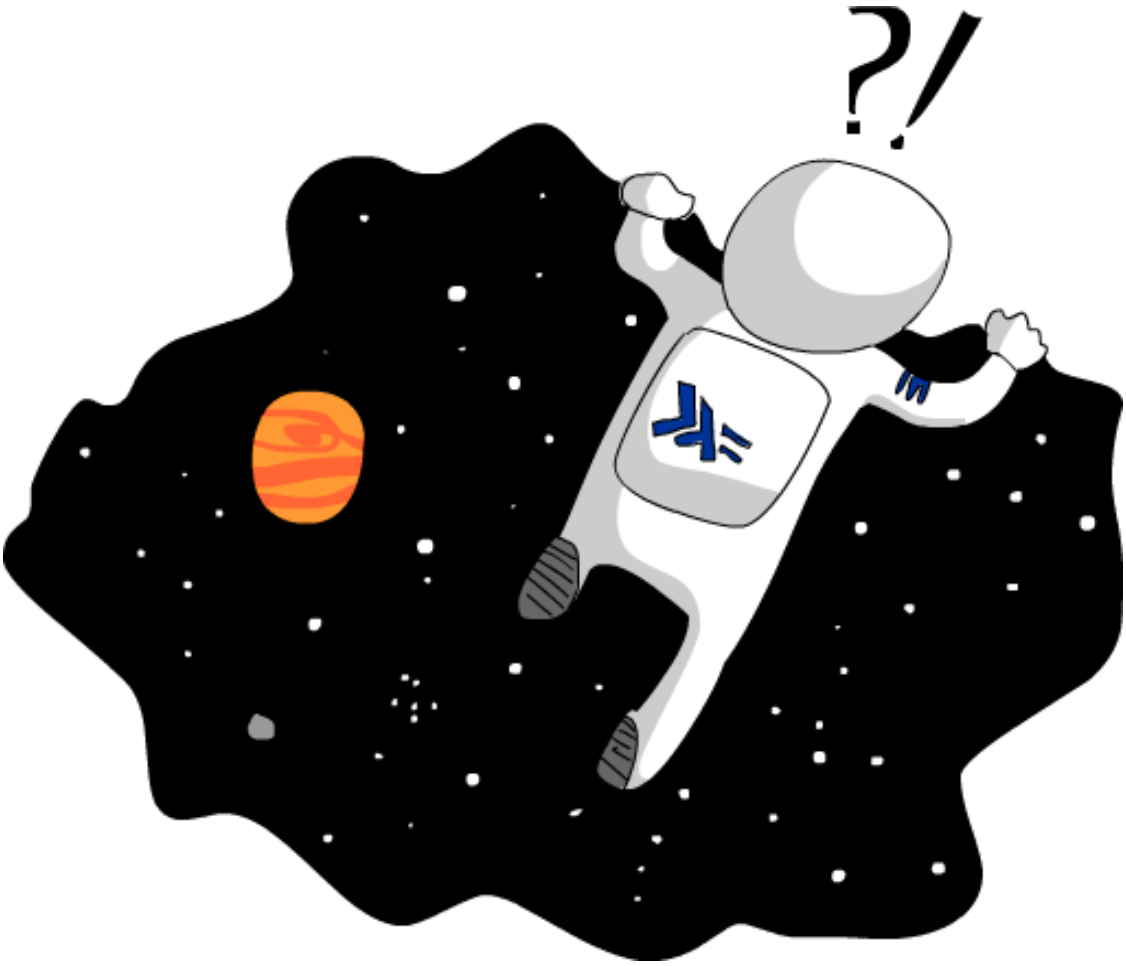
Зверніть увагу, що ця функція спирається на припущення, що поточне дерево в фокусі не є `Empty`. Пусте дерево не має піддерев, тому якщо ми спробуємо піти ліворуч пустим деревом, отримаємо помилку, позаяк зіставлення по `Node` не спрацює, а взірця, який подбає про `Empty`, іще не означено.

`goRight` є подібною:

```
goRight :: (Tree a, Breadcrumbs a) -> (Tree a, Breadcrumbs a)
goRight (Node x l r, bs) = (r, RightCrumb x l:bs)
```

Раніше ми мали змогу йти ліворуч або праворуч. А тепер ми маємо змогу йти назад наверх, бо ми запам'ятали різні речі про батьківські вузли і шляхи, що їх не відвідали. Ось код функції `goUp`:

```
goUp :: (Tree a, Breadcrumbs a) -> (Tree a, Breadcrumbs a)
goUp (t, LeftCrumb x r:bs) = (Node x t r, bs)
goUp (t, RightCrumb x l:bs) = (Node x l t, bs)
```



Ми фокусуємося на дереві `t` і перевіряємо найновішу `Crumb`. Якщо вона є `LeftCrumb`, тоді ми будуємо нове дерево із нашим деревом `t` у якості лівого піддерева, і використовуємо інформацію про праве піддерево, що ми не відвідали, і елемент для добудови `Node`. Оскільки ми повернулися назад і, так би мовити, підняли останню крихту і використали її для відбудови батьківського дерева, новий список хлібних крихт її не міститиме.

Зверніть увагу, що ця функція завалюється, якщо ми нагорі дерева і спробуємо залізти ще вище. Незабаром ми скористаємося монадою `Maybe`, щоб описати можливі аварії зміни фокуса.

`Tree a` і `Breadcrumbs a` — то є вся потрібна інформація для відбудови усьо-

го дерева. Крім того, ми запам'ятовуємо, на якому піддереві сфокусувалися. Така схема уможливлює легкі рухи вгору, ліворуч чи праворуч. Така парочка плюс фокус на частину структури даних плюс її оточення називається застібкою-блискавкою, бо рух фокусу догори і вниз по структурі даних нагадує роботу звичайної застібки на звичайних штанях. Тому тип-синонім файно назвати ось як:

```
type Zipper a = (Tree a, Breadcrumbs a)
```

Я волів би назвати той тип-синонім `Focus`, бо така назва підкреслює, що ми фокусуємося на частині структури даних. Однак термін застібка є більш розповсюдженим, то ми не вигадуватимемо колесо і використовуватимемо `Zipper`.

14.2.2 Маніпуляція деревами, що у фокусі

Тепер ми вже можемо рухатися вниз і вгору, тому напишімо функцію, яка модифікує елемент у корінні піддерева, на яке вказує застібка:

```
modify :: (a -> a) -> Zipper a -> Zipper a
modify f (Node x l r, bs) = (Node (f x) l r, bs)
modify f (Empty, bs) = (Empty, bs)
```

Якщо ми фокусуємося на вузлі, ми модифікуємо його корінь функцією `f`. Якщо ми фокусуємося на пустому піддереві, ми лишаємо його як є. Тепер ми можемо розпочати роботу з деревом, перейти по ньому куди треба і модифікувати елемент, тримаючи фокус на тому елементі, щоб ми могли легко перейти вниз чи догори, за потребою. До прикладу:

```
ghci> let newFocus = modify (\_ -> 'P') (goRight (goLeft (freeTree,[])))
```

Ми йдемо ліворуч, потім праворуч і модифікуємо елемент в корені, замінюючи його на `'P'`. Читається краще, якщо задіяти `-:`:

```
ghci> let newFocus = (freeTree,[]) -: goLeft -: goRight -: modify (\_ -> 'P')
```

Ми можемо перейти вверх, якщо забажаємо, і замінити елемент на таємниче `'X'`:

```
ghci> let newFocus2 = modify (\_ -> 'X') (goUp newFocus)
```

Або ж, якщо написати це з `-:`:

```
ghci> let newFocus2 = newFocus -: goUp -: modify (\_ -> 'X')
```

Рухатися нагору так легко завдяки хлібним крихтам, що описують структуру даних, на якій ми не фокусуємося. Тільки тепер її вивернуто, як таку собі

шкарпетку. Ось тому, якщо ми хочемо піднятися нагору, нам не треба починати з кореня і спускатися, бо ми просто знімаємо верхівку нашого інвертованого дерева, і таким чином де-інвертуємо якусь його частину і додаємо її до нашого фокуса.

Кожен вузол має два піддерева, навіть якщо ті піддерева пусті. Отже, якщо ми фокусуємося на пустому піддереві, ми можемо замінити його на непусте, і, таким чином, під'єднаємо дерево до вузла-листя. Код, що реалізує це, — простий:

```
attach :: Tree a -> Zipper a -> Zipper a
attach t (_, bs) = (t, bs)
```

Ми беремо дерево і застібку та повертаємо нову застібку, фокус якої замінено на подане дерево. Ми не тільки можемо розширяти дерева у такий спосіб, себто, заміняючи піддерева новими деревами, а й можемо заміняти цілі, вже існуючі, піддерева. Під'єднаймо дерево у лівий нижній кут нашого `freeTree`:

```
ghci> let farLeft = (freeTree,[]) -: goLeft -: goLeft -: goLeft -: goLeft
ghci> let newFocus = farLeft -: attach (Node 'Z' Empty Empty)
```

`newFocus` тепер вказує на дерево, яке ми щойно причепили, а решта піддерев лежать у хлібних крихтах. Якщо б ми зійшли нагору за допомогою `goUp`, то мали б те саме дерево `freeTree`, але з додатковим `'Z'` внизу, з лівої сторони.

14.2.3 Вгору я біжу прямо на вершину, так-так, де чисте і свіже повітря!

Написати функцію, що іде наверх деревом, незалежно від поточного фокусу, дуже легко. Ось вона:

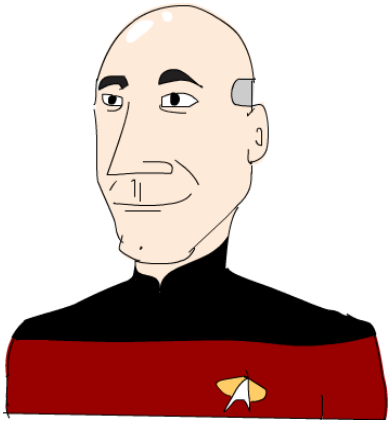
```
topMost :: Zipper a -> Zipper a
topMost (t,[]) = (t,[])
topMost z = topMost (goUp z)
```

Якщо наша навігаційна стежка на стероїдах пуста, це означає, що ми опинилися на вершині нашого дерева, тому ми просто повертаємо поточний фокус. В іншому випадку, ми йдемо нагору, щоб отримати фокус на батьківський вузол і рекурсивно викликаємо `topMost` по ньому. Отож тепер ми можемо ходити по нашому дереву як нам заманеться, ліворуч чи праворуч чи нагору, застосовуючи `modify` і `attach` коли треба, а по завершенню тих модифікацій, ми пересуваємо фокус на коріння, використовуючи функцію `topMost` і бачимо наші зміни в звичній нам «проекції».

14.3 Фокусування на списках

Застібки можна використовувати з будь-якою структурою даних, тому не дивно, що вони застосовуються для фокусування на підсписках списків. Врешті-решт, списки дуже схожі на дерева — у вузлі дерева є елемент (чи немає) і декілька піддерев, а «вузол» списку має елемент і лише один підсписок. Коли ми реалізували наші власні списки в підрозділі 8.6, ми означили цей тип даних ось як:

```
data List a = Empty | Cons a (List a) deriving (Show, Read, Eq, Ord)
```



Порівняйте це з нашим означенням бінарного дерева — відразу видно, що можна думати про списки, як про дерева, де вузли мають лише одне піддерево.

А списки на кшталт `[1,2,3]` можна написати як `1:2:3:[]`. Такі списки складаються з голови списку, тобто, в цьому випадку, `1` і власне хвоста того списку, що дорівнює `2:3:[]`. `2:3:[]` теж має голову, тобто `2` і хвіст `3:[]`. У `3:[]` голова є `3`, а хвіст — пустий список `[]`.

Створімо застібку для списків. Щоб змінити фокус у списках, ми рухаємося або вперед, або назад (а для дерев можливі рухи були: вниз ліворуч, вниз праворуч або нагору). Частина у фокусі буде підсписком, і разом із нею ми лишатимемо хлібні крихти у міру того, як просуватимемося уперед. Ну а з чого складатиметься одна спискова крихта? Коли ми працювали з бінарними деревами, то казали, що хлібна крихта має містити елемент батьківського вузла, разом із усіма піддеревами того вузла, які ми не вибрали для фокусування. Також крихта зазначала, чи ми пішли ліворуч чи праворуч. Отже, там було все окрім піддерева у фокусі.

Списки простіші за дерева, тому нам не треба пам'ятати, ліворуч ми пішли чи ні, оскільки є лише один шлях заглибитися у список. Оскільки кожен вузол має лише один-єдиний підсписок, нам також не треба пам'ятати шляхи, що їх ми не обрали. Виходить, що нам треба тільки запам'ятати попередній елемент. Якщо ми маємо список на кшталт `[3,4,5]` і знаємо, що попередній елемент був `2`, ми можемо піти назад, просто повернувши той елемент до голови нашого списку, отримуючи `[2,3,4,5]`.

Оскільки хлібна крихта містить лише значення, нам не обов'язково запаковувати її всередину типу даних, як ми робили, коли створювали тип даних `Crumb` для застібок для дерев:

```
type ListZipper a = ([a],[a])
```

Перший список то є підсписок у фокусі, а другий — список хлібних крихт. Напишімо функцію, що ходить вперед і назад по списках:

```
goForward :: ListZipper a -> ListZipper a
goForward (x:xs, bs) = (xs, x:bs)

goBack :: ListZipper a -> ListZipper a
goBack (xs, b:bs) = (b:xs, bs)
```

Коли ми йдемо вперед, то фокусуємося на хвості поточного списку і залишаємо значення з голови як крихту. Коли вертаємося, ми беремо найсвіжішу крихту і вставляємо її у голову списку.

Ну а тепер поглянемо на ці функції у дії:

```
ghci> let xs = [1,2,3,4]
ghci> goForward (xs,[])
([2,3,4],[1])
ghci> goForward ([2,3,4],[1])
([3,4],[2,1])
ghci> goForward ([3,4],[2,1])
([4],[3,2,1])
ghci> goBack ([4],[3,2,1])
([3,4],[2,1])
```

Бачимо, що навігаційна стежка у випадку списків то є просто розвернута попередня частина нашого списку. Значення, від якого ми віддаляємося, завжди йде до голови списку крихт, тому легко повернутися, перекинувши його назад з голови списку крихт до голови списку у фокусі.

Тут добре видно, чому ці штуки називають застібками-блискавками: бо вони нагадують такі застібки повзунком, що рухається вверх-вниз.

Якби ми з вами писали текстовий редактор, можна було б узяти список рядків як представлення рядків тексту. Тоді за допомогою блискавки ми би позначили лінію, на якій стоїть курсор. Використання блискавки також полегшило б додавання нових рядків тексту будь-де у тексті, або ж видалення вже існуючих рядків тексту.

14.4 Простенька файлова система

Тепер, коли ми вже знаємо, як працюють блискавки, використаймо дерева, щоб описати файлову систему, а тоді створімо блискавку для тієї файлової системи,

щоб можна було пересуватися з директорії в директорію, точнісінько так, як ми зазвичай це робимо, стрибаючи по власній файловій системі.

Кажучи по-простому, файлова система складається переважно з файлів і директорій. Файли — це одиниці даних, вони мають ім'я, тоді як директорії використовуються для організації тих файлів і можуть містити файли або ж інші директорії. Тобто одиниця файлової системи є або файлом, що має ім'я і якісь дані, або директорією, що має ім'я і зміст — купу інших одиниць, — файлів чи директорії. Ось тип даних для реалізації цього і деякі типи-синоніми, аби ми знали, що є що:

```
type Name = String
type Data = String
data FSItem = File Name Data | Folder Name [FSItem] deriving (Show)
```

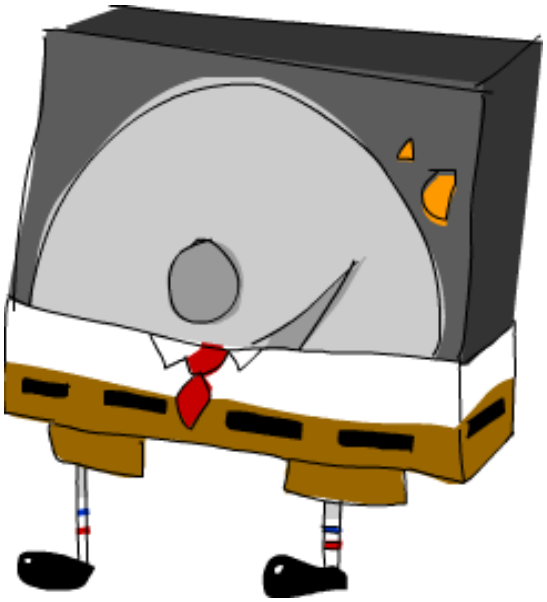
Файл складається з двох рядків — його імені і даних, які він тримає у собі. Директорія складається з рядка з ім'ям і списку-змісту. Якщо той список пустий, маємо пусту директорію.

Ось директорія із декількома файлами і піддиректоріями:

```
myDisk :: FSItem
myDisk =
    Folder "root"
        [ File "goat_yelling_like_man.wmv" "baaaaaa"
        , File "pope_time.avi" "god bless"
        , Folder "pics"
            [ File "ape_throwing_up.jpg" "bleargh"
            , File "watermelon_smash.gif" "smash!!"
            , File "skull_man(scary).bmp" "Yikes!"
            ]
        , File "dijon_poupon.doc" "best mustard"
        , Folder "programs"
            [ File "fartwizard.exe" "10gotofart"
            , File "owl_bandit.dmg" "mov eax, h00t"
            , File "not_a_virus.exe" "really not a virus"
            , Folder "source code"
                [ File "best_hs_prog.hs" "main = print (fix error)"
                , File "random.hs" "main = print 4"
                ]
            ]
        ]
```

Так виглядає зміст мого диску на теперішній момент.

14.4.1 Застібка для нашої файлової системи



Тепер у нас є файлова система, і все що нам залишилося — це застібка для легкої навігації по ній, щоб, своєю чергою, ми могли модифікувати і видаляти файли та директорії. Як із бінарними деревами і списками, ми залишатимемо хлібні крихти з інформацією про всі місця, які ми вирішили не відвідувати. Тобто одна така крихта повинна бути чимось на кшталт вузла, от лише містити вона має все, окрім піддерева у фокусі. Також вона має містити інформацію про позицію дірки, щоб ми знали куди вставляти наш попередній фокус, як рухатимемося нагору.

У цьому випадку крихта має бути звичайною директорією, от тільки вона не повинна містити директорію, яку ми вибрали. Ви запитаєте: а чому крихта є директорією, а не файлом? Ну, тому що, коли ми сфокусувалися на файлі, ми вже не можемо спускатися глибше по файловій системі, тому немає сенсу лишати крихту, яка каже, що ми прийшли з файлу. Файл то є щось на кшталт пустого піддерева.

Якщо ми фокусуємося на директорії `"root"`, а потім на файлі `"dijon_poupon.doc"`, як виглядатиме крихта, що її ми залишаємо? Вона повинна містити ім'я батьківської директорії та її зміст, що складатиметься з двох частин — змісту, що передує файлові, на якому ми сфокусувалися, і решту змісту, що йде після нього. Отже, все що нам треба — це `Name` і два списки для змісту. Тримаючи зміст у двох списках, ми знаємо, де саме є те місце, куди ми спустилися, тому коли знову рухатимемося нагору, знатимемо, де покласти те, чого бракує. Тобто ми знаємо, де сидить дірка.

Ось тип крихти для файлової системи:

```
data FSCrumb = FSCrumb Name [FSItem] [FSItem] deriving (Show)
```

А ось тип-синонім для нашої застібки:

```
type FSZipper = (FSItem, [FSCrumb])
```

Іти назад нагору по цій ієрархії дуже просто. Ми просто беремо найсвіжішу крихту і збираємо новий фокус з поточного фокусу і неї. Ось так:

```
fsUp :: FSZipper -> FSZipper
fsUp (item, FSCrumb name ls rs:bs) = (Folder name (ls ++ [item] ++ rs), bs)
```

Наша крихта знає ім'я батьківської директорії, елементи, які передували елементові у фокусі (тобто `ls`) і елементи після нього (тобто `rs`), тому нам легко рухатися нагору.

Може заглибимося у файлову систему ще трохи? Якщо ми в `"root"` і хочемо сфокусуватися на `"dijon_poupon.doc"`, хлібні крихти, які ми залишаємо, будуть містити ім'я `"root"` разом із усіма елементами, що передують `"dijon_poupon.doc"`, і усіма, що йдуть після нього.

Ось функція, яка бере ім'я і фокусується на файлі чи директорії, що знаходиться у директорії у поточному фокусі:

```
import Data.List (break)

fsTo :: Name -> FSZipper -> FSZipper
fsTo name (Folder folderName items, bs) =
    let (ls, item:rs) = break (nameIs name) items
    in  (item, FSCrumb folderName ls rs:bs)

nameIs :: Name -> FSItem -> Bool
nameIs name (Folder folderName _) = name == folderName
nameIs name (File fileName _) = name == fileName
```

`fsTo` бере `Name` і `FSZipper` і повертає нову `FSZipper`, що вказує на файл із відповідним ім'ям. Цей файл має бути у директорії у поточному фокусі. Ця функція не шукатиме той файл скрізь, вона лише перегляне зміст поточної директорії.

Спершу ми використаємо `break`, щоб розділити список змісту директорії на дві частини — ту, що передує файлу, який ми шукаємо, і ту частину, що йде після нього. Якщо ви пригадуєте, `break` бере предиката і список і повертає пару списків. Перший список у парі міститиме всі елементи, для яких предиката повернула `False`. Щойно предиката повернула `True` для якогось елемента, функція повертає той елемент і решту того списку як список номер два у парі. Ми написали допоміжну функцію під назвою `nameIs`, що бере ім'я і одиницю файлової системи і повертає `True`, якщо імена співпадають.

Тепер `ls` — це список, що містить елементи, які передують тому, що ми шукаємо, `item` — це те, що ми шукаємо, і `rs` — це список з елементів, які йдуть після того, що ми шукаємо. Тепер коли в нас це все є, ми просто видаємо те, що нам повернула функція `break` і будуємо крихту, що має усі дані, яких потребує.

Зверніть увагу: якщо ім'я, яке ми шукаємо, не є присутнім у директорії, взірець `item:rs` спробує зіставитись із пустим списком, і ми отримаємо помилку. Також, якщо наш поточний фокус взагалі не директорія, а файл, ми також матимемо помилку і ще й аварійне завершення програми.

Тепер ми можемо пересуватися вгору і вниз файловою системою. Почнімо з кореня і підемо до файлу `"skull_man(scary).bmp"`:

```
ghci> let newFocus =
          (myDisk,[]) –: fsTo "pics" –: fsTo "skull_man(scary).bmp"
```

`newFocus` тепер є застібкою, що вказує на файл `"skull_man(scary).bmp"`. Перегляньмо першу компоненту застібки (тобто, власне, фокус), щоб переконатися, що це дійсно так:

```
ghci> fst newFocus
File "skull_man(scary).bmp" "Yikes!"
```

Перейдімо нагору і сфокусуймося на сусідньому файлі `"watermelon_smash.gif"`:

```
ghci> let newFocus2 = newFocus –: fsUp –: fsTo "watermelon_smash.gif"
ghci> fst newFocus2
File "watermelon_smash.gif" "smash!!"
```

14.4.2 Маніпуляції з нашою файловою системою

Тепер ми знаємо, як рухатися нашою файловою системою, і тому зможемо легко нею маніпулювати. Ось функція для перейменування файлу чи директорії, яка зараз у фокусі:

```
fsRename :: Name –> FSZipper –> FSZipper
fsRename newName (Folder name items, bs) = (Folder newName items, bs)
fsRename newName (File name dat, bs) = (File newName dat, bs)
```

Тепер ми можемо змінити ім'я директорії `"pics"` на `"cspi"`:

```
ghci> let newFocus =
          (myDisk,[]) –: fsTo "pics" –: fsRename "cspi" –: fsUp
```

Ми спустилися до директорії `"pics"`, змінили її і повернулися назад нагору.

А як щодо функції, яка створює новий елемент у нашій поточній директорії? Дивіться і дивуйтеся:

```
fsNewFile :: FSItem -> FSZipper -> FSZipper
fsNewFile item (Folder folderName items, bs) =
    (Folder folderName (item:items), bs)
```

Просто, як двері. Зверніть увагу, що ця функція завалиться, якщо ми спробуємо додати новий елемент, а сфокусуємося на файлі, а не на директорії.

Додаймо до нашої директорії `"pics"` якийсь файл і після того повернімося у корінь:

```
ghci> let newFocus =
          (myDisk,[]) -: fsTo "pics" -:
              fsNewFile (File "heh.jpg" "lol") -: fsUp
```

Найкрутіша річ з цього усього — це те, що коли ми модифікуємо нашу файлову систему, ми не модифікуємо оригінал, а повертаємо новеньку файлову систему з нашими змінами. У такий спосіб ми маємо доступ як до старої файлової системи (у цьому випадку, `myDisk`) так і до нової (перша компонента `newFocus`). Отже, користуючись застібками, ми отримуємо систему керування версіями як бонус. Іншими словами, ми завжди можемо звернутися до старої версії структур даних після їхньої модифікації, так би мовити. Це не є унікальним для застібок — це властивість Хаскела, бо його структури даних не можна змінити. Завдяки застібкам у нас є змога легко та ефективно пересуватися нашими структурами даних, і саме тут надійність структур даних Хаскела починає по-справжньому вражати.

14.5 Обережно — слизько

Дотепер, гуляючи нашими улюбленими структурами даних, чи то бінарними деревами, списками чи то файловими системами, ми безтурботно не боялися зайти занадто далеко і оступитися. До прикладу, наша функція `goLeft` брала застібку для бінарного дерева і пересувала фокус на ліве піддерево:

```
goLeft :: Zipper a -> Zipper a
goLeft (Node x l r, bs) = (l, LeftCrumb x r:bs)
```

А що, якби ми спробували спуститися пустим деревом? Іншими словами, як бути, якщо те дерево не `Node`, а

`Empty`? У цьому випадку ми отримаємо помилку виконання, бо зіставлення із взірцем не відбудеться, і в нас немає іншого взірця, що подбає про пусте дерево, у якого взагалі немає піддерев. Наразі ми просто припускали, що ми ніколи не сфокусуємося на лівому піддереві пустого дерева, бо в пустого дерева немає лівого піддерева. Просто наразі ми такий рух вважали безглуздим і зручнісінько ігнорували таку прикрість.

Ну а якщо б ми вже були у корені якогось дерева і навігаційна стежка була б порожня як банка, але ми все одно спробували б піти нагору? Відбулося б те саме. Здається, застібки річ така, що будь-який крок може стати нашим останнім (грає зловісна музика). Іншими словами, будь-який крок може бути успішним, але може бути і неуспішним також. Що це нам нагадує? Монади, звичайно ж! А конкретніше, це монада `Maybe`, що додає аварійного контексту до звичайних значень.

Ну то використаймо монаду `Maybe`, щоб додати аварійний контекст до наших рухів. Ми візьмемо наші функції, що працюють із застібками для бінарних дерев і перетворимо їх в монадні. Спершу, подбаймо про можливу аварію у `goLeft` і `goRight`. Дотепер, аварія функцій завжди віддзеркалювалася у їхньому результаті, і цього разу буде те саме. Ось `goLeft` і `goRight` із доданою можливістю аварійного завершення:

```
goLeft :: Zipper a -> Maybe (Zipper a)
goLeft (Node x l r, bs) = Just (l, LeftCrumb x r:bs)
goLeft (Empty, _) = Nothing

goRight :: Zipper a -> Maybe (Zipper a)
goRight (Node x l r, bs) = Just (r, RightCrumb x l:bs)
goRight (Empty, _) = Nothing
```

Класно, тепер якщо ми спробуємо піти ліворуч від пустого дерева, отримаємо `Nothing`!

```
ghci> goLeft (Empty, [])
Nothing
```

```
ghci> goLeft (Node 'A' Empty Empty, [])
Just (Empty,[LeftCrumb 'A' Empty])
```

Виглядає зовсім непогано. Як щодо руху нагору? Мали проблему із рухом нагору, коли хотіли піти наверх, а крихт не було, що означало, що ми були в корені дерева. Ось функція `goUp`, яка викидає помилку, якщо ми вийдемо за межі нашого дерева:

```
goUp :: Zipper a -> Zipper a
goUp (t, LeftCrumb x r:bs) = (Node x t r, bs)
goUp (t, RightCrumb x l:bs) = (Node x l t, bs)
```

Тепер модифікуймо її так, щоб вона завалювалась граціозно:

```
goUp :: Zipper a -> Maybe (Zipper a)
goUp (t, LeftCrumb x r:bs) = Just (Node x t r, bs)
goUp (t, RightCrumb x l:bs) = Just (Node x l t, bs)
goUp (_, []) = Nothing
```

Якщо є крихти, все окей і ми повертаємо успішний новий фокус, а як немає — повертаємо помилку.

Дотепер, ці функції брали застібки і повертали застібки, що дозволяло з'єднувати їх в ланцюг в отакий спосіб для пересування:

```
gchi> let newFocus = (freeTree,[]) -: goLeft -: goRight
```

Але тепер замість старого `Zipper a` вони повертають `Maybe (Zipper a)`, тому таке з'єднання в ланцюг не спрацює. У нас була схожа задача, коли ми возилися із нашим канатохідцем у підрозділі ?? з розділу про монади. Він також ходив один крок за раз і кожен із кроків міг бути фатальним, бо купа пташок могли приземлитися на одну сторону його жердяки для балансування і призвести до його падіння.

Тепер він мабуть сміється, бо тепер наша черга ходити, і ми блукаємо лабіринтами, які самі ж збудували. На жаль-на щастя, ми можемо повчитися у нашого канатохідця і робити те саме, що й він, а саме: замінити звичайне застосування функції на монадне `>>=`, що бере значення із контекстом (у нашому випадку, `Maybe (Zipper a)`, де контекст описує можливість аварії) і годує ним функцію, водночас дбаючи про правильне скористання тим контекстом, за потреби. Тому, за аналогією із канатохідцем, ми робимо бартерний обмін наших операторів `-:` на `>>=`. Добре, тепер таке з'єднання в ланцюг працюватиме знову! Дивіться:

```
ghci> let coolTree = Node 1 Empty (Node 3 Empty Empty)
ghci> return (coolTree,[]) >>= goRight
Just (Node 3 Empty Empty,[RightCrumb 1 Empty])
```

```
ghci> return (coolTree,[]) >>= goRight >>= goRight
Just (Empty,[RightCrumb 3 Empty,RightCrumb 1 Empty])
ghci> return (coolTree,[]) >>= goRight >>= goRight >>= goRight
Nothing
```

Ми використовуємо `return`, щоб загорнути застібку в `Just`, а після — `>>=`, щоб нагодувати нашу функцію `goRight`. Спочатку ми створили дерево з пустим лівим піддеревом, і непустим правим, що складається з вузла із двома пустими піддеревами. Коли ми спробували піти праворуч один раз, результат був успішним, бо операція мала сенс. Двічі праворуч працювало також; наш фокус вказував на пусте піддерево. А от іти праворуч тричі — то є нісенітниця, тому що ми не можемо піти праворуч, виходячи з пустого піддерева, і тому в результаті отримуємо `Nothing`.

Тепер ми підклали під нашими деревами монадні скирти сіна, і вони нас врятують якщо що. Круто.

Наша файлова система також має інші крайові випадки, де можуть бути аварії іншого типу, такі як, наприклад, спроба фокусування на файлі чи директорії, що не існують. Щоб трохи повправлятися, спробуйти озброїти нашу файлову систему функціями, що граціозно завалюються завдяки використанню монади `Maybe`.

Покажчик