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

2017-05-21T00:06:11Z
Версія v4.7-54-gda41cf2

Зміст

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:

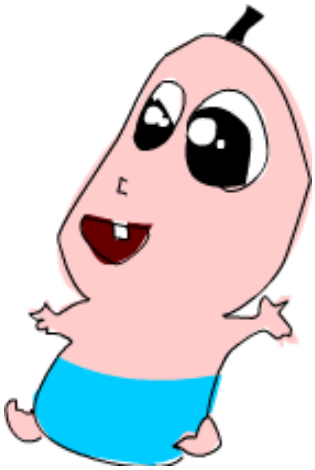
```
doubleUs x y = doubleMe x + doubleMe y
```

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]
```

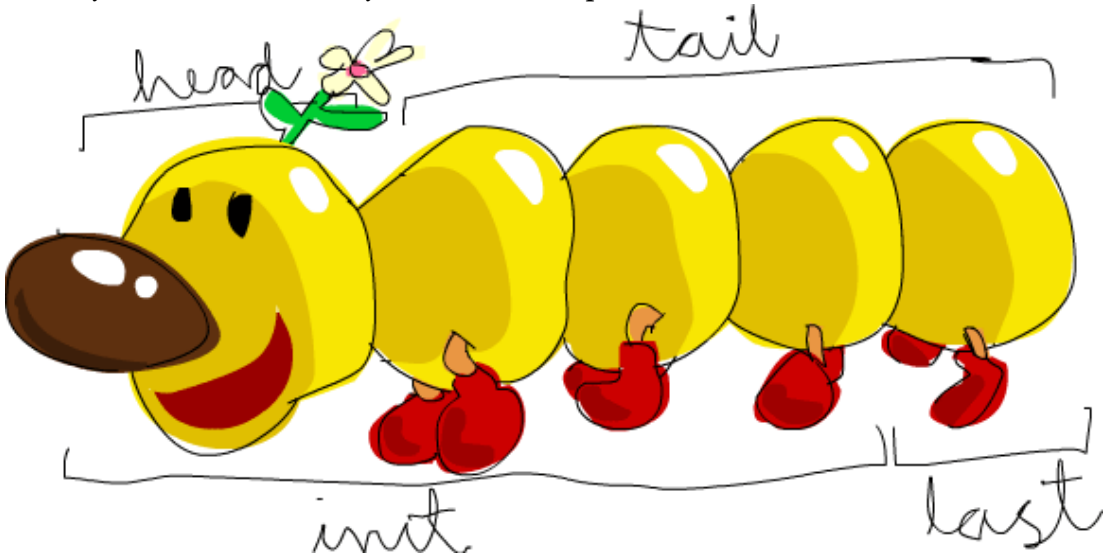
`last` бере список і повертає його останній елемент.

```
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
```

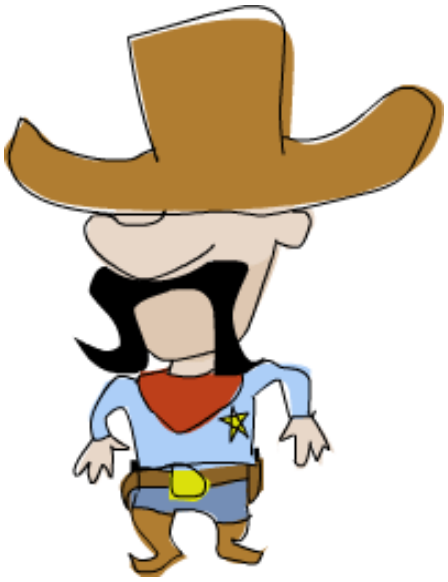
`elem` бере щось і список речей і каже, чи це щось є елементом списку. Зазвичай її викликають як інфіксну функцію, бо так код легше читається.

```
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 **??**, 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?

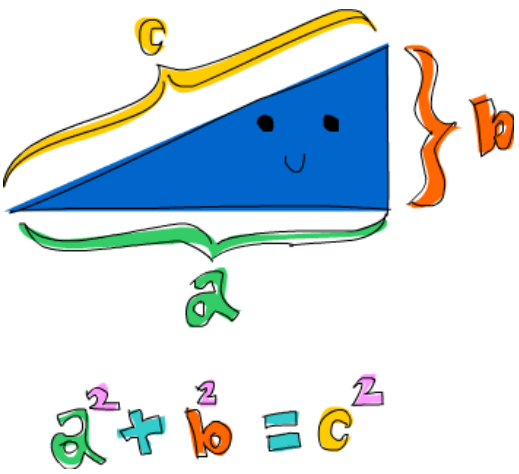
```
ghci> zip [5,3,2,6,2,7,2,5,4,6,6] ["im","a","turtle"]
[(5,"im"),(3,"a"),(2,"turtle")]
```

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.

Покажчик