Glasgow Haskell Compiler: неформальний вступ

Юрій Стативка

Жовтень, 2021 р.

1 Інтерпретатор `ghci`

Є кілька реалізацій мови `Haskell` найпопулярнішою з яких є `GHC` — Glasgow Haskell Compiller. `GHC` містить ряд компонентів, серед яких, зокрема, `ghc`, `ghci` та `runghc`. Ці інструменти забезпечують, відповідно: `ghc` — компіляцію до нативного коду та мови `C`, `ghci` — інтерпретацію та налагодження, `runghc` — виконання Haskell-програм як сценаріїв (скриптів) без попередньої компіляції.

Далі всі приклади будуть наводитись для системи `GHC`.

Після запуску `ghci` у консолі (або у окремому вікні для `winghci`) з'явиться повідомлення інтерпретатора

```
>ghci
GHCi, version 8.10.1: https://www.haskell.org/ghc/  :? for help
Prelude>
```

Запрошення `Prelude>` свідчить про автоматичне завантаження стандартного модуля з файлу `Prelude.hs`, який містить найнеобхідніші для стартових потреб функції та називається *Прелюдією*. Після цього інтерпретатор готовий до роботи і очікує вводу *виразів мови* `Haskell` для оцінки, або *команд інтерпретатора* для виконання. Останні починаються з двокрапки (`:`). Наприклад завершити роботу інтерпретатора можна за допомогою команди `:quit`, викликати довідку – командою `:help`, (або їх короткими версіями `:q` та `:h`).

Для оцінки значення виразу необхідно набрати його після запрошення та натиснути клавішу `Enter`. Знайдемо значення деяких числових виразів:

```
Prelude> 2^3+4*(1+2)
20

Prelude> 2^3+4*(1.0+2)
20.0

Prelude> 2^3+4*(1+2)/3
12.0

Prelude> 1/3
0.3333333333333333
```

```
Prelude> 11^34
25547669861876588955101944575940044l

Prelude> 11.0^34
2.554766986187659e35

Prelude> 2e3/2
1000.0

Prelude> 2^3/2
4.0
```

З діалогу видно, що інтерпретатор розрізняє цілі та дійсні числа, останні — в формі з фіксованою чи плаваючою крапкою.

Для зміни запрошення з `"Prelude>"` на `">"` можна скористатись командою інтерпретатора :

```
Prelude> :set prompt "> "
>
```

2 Типи

У `Haskell`'і будь-яка величина є *виразом*, обчислення якого приводить до *значення*. Кожне значення має певний *тип*. До базових типів мови `Haskell` відносять:

- Типи `Integer` и `Int` використовуються для представлення цілих чисел, причому значення типу `Int` обмежені чотирма байтами (від -2^{31} до $2^{31}-1$), а значення типу `Integer` — довільні.
- Типи `Float` и `Double` використовуються для представлення дійсних чисел.
- Тип `Bool` використовується для представлення логічних значень `True` та `False`.
- Тип `Char` використовується для представлення символів.

Імена типів у `Haskell`'і завжди починаються з великої літери.

Однією з особливостей мов функціонального програмування є те, що функції часто виступають як дані — функція може бути аргументом іншої функції або бути результатом виконання функції.

Кожен тип асоційований з певним набором операцій, які можуть не мати сенсу для інших типів. Так, не можна, наприклад, розділити один символ на інший.

Важливим принципом багатьох мов програмування є те, що кожен правильний вираз може бути співвіднесений з деяким типом. Крім того, цей тип може

бути виведений виключно з типів складових частин виразу. Тобто тип виразу повністю визначається типами його складових. `Haskell` належить до мов із суворою типізацією, тому перевірка типів виконується ще до виконання обчислень. І якщо трапляється вираз, який не може бути асоційований з прийнятим типом — він вважається неправильним.

Хоч `Haskell` є мовою з *сильною типізацією*, та в більшості випадків програміст може не оголошувати типи, оскільки інтерпретатор сам здатен *вивести* типи введених змінних. Проте, можливість явного оголошення типів є і реалізується за допомогою конструкції `змінна::Тип`.

```
>4*(2::Double)
8.0
```

2.1 Числа

Операції цілочислового ділення та обчислення залишку демонструє наступний діалог. Кожна з операцій показана в операторній (інфіксній) та функціональній нотації.

```
>50 `div` 6
8

>div 50 6
8

>50 `mod` 6
2

>mod 50 6
2
```

До цілих типів застосовні функції модуля Prelude: `gcd x y` та `lcm x y` для обчислення найменшого спільного кратного та найбільшого спільного дільника відповідно, `even x` та `odd x` для перевірки парності та непарності, тощо.

```
>gcd 12 18
6

>lcm 12 18
36

>even 12
True

>odd 12
False
```

До дробових чисел застосовні функції з Прелюдії, такі як `abs`, `signum`, `round`, `trunc`, `sin`, `asin` та інші. Операції порівняння мають форму `<, <=, >, >=, ==, /=`.

Інформацію про тип виразу надає команда `:type вираз`

```
>:type 1+2
1+2 :: (Num t) => t
```

Тут інтерпретатор повідомляє, що вираз `1+2` має тип, позначений **змінною типу** `t` з обмеженням (**контекст типу**) `(Num t)`, тобто `t` – один з числових типів.

Команди `:set parameter` та `:unset parameter` дозволяють змінити параметри інтерпретатора. Так параметр `+t` впливає на виведення типу результату:

```
> :set +t

> 2+3
5
it :: Integer

> it^3
125
it :: Integer
```

З наведеного прикладу видно, що інтерпретатор зв'язує результат зі спеціальною змінною `it`, доступною для використання. Відповідь інтерпретатора можна читати так: "вираз $2 + 3$ має значення 5 типу Integer". Наступний приклад демонструє відмінності команд `:type` та `:set +t`

```
> 2+3/4
2.75
it :: Double

> :type 2+3/4
2+3/4 :: (Fractional t) => t
```

Для повернення до звичайного режиму використовують команду `:unset +t`

2.2 Логічні величини

`Haskell` має засоби для роботи з логічними величинами:

```
> True
True

> False
False

> not True
False
```

```
> not True || True
True

> not True && True
False
```

Тут, зокрема, представлені оператори *заперечення*, *кон'юнкції* та *диз'юнкції* — `not`, `&&` та `||` відповідно. Переглянути інформацію про імена дозволяє функція *:info*

```
> :info not && ||
not :: Bool -> Bool  -- Defined in GHC.Classes
(&&) :: Bool -> Bool -> Bool  -- Defined in GHC.Classes
infixr 3 &&
(||) :: Bool -> Bool -> Bool  -- Defined in GHC.Classes
infixr 2 ||
```

Як це видно з діалогу, оператори кон'юнкції та диз'юнкції є інфіксними і обидва правоасоціативні. Більш високий пріоритет — у оператора кон'юнкції. Оператори (і не тільки логічні) можна записувати у формі функцій (префіксній), для цього їх слід брати в дужки:

```
> (+) 2 3
5

> (&&) True True
True
```

Натомість бінарні функції (функції з двома аргументами) можна використовувати як інфіксні оператори, для чого їх беруть у обернені лапки. Наступний приклад обчислює найбільший спільний дільник з використанням функції `gcd` у префіксній та інфіксній нотації відповідно:

```
Prelude> 12 `gcd` 18
6

Prelude> gcd 12 18
6
```

На множині логічних величин визначені також операції порядку та рівності:

```
>True == True
True

>True /= True
False
```

```
>True <= True
True

>True <= False
False

>True > False
True
```

Під час обчислення значення правильного виразу може виникнути виключення (помилка), яка позначається символом $\perp$ (основа, bottom) і в Haskell'і не відрізняється від незавершеного обчислення. Оскільки Haskell є мовою з суворою типізацією даних, то всі типи містять $\perp$. Помилки викликають негайне завершення програми. Наступний приклад демонструє таку ситуацію:

```
Prelude> 1/0
Infinity
```

`Prelude` містить дві функції, які призводять до такої помилки – це `error message` та `undefined`, перша з яких супроводжується виведенням рядка `message`, а друга — повідомленням компілятора:

```
Prelude> error "похибка"
*** Exception: похибка

Prelude> undefined
*** Exception: Prelude.undefined
```

2.3 Символьні величини

Робота з величинами типу `Char` демонструється наступним протоколом

```
>:set +t

>'a'
'a'
it :: Char

>'\t'
'\t'
it :: Char

>'\n'
'\n'
it :: Char

>'5'
'5'
it :: Char
```

Величини типу `Char` впорядковані та допускають порівняння:

```
>'a'<'b'
True

>'s' == 'd'
False
```

Завантаживши модуль `Data.Char` командою `:module модуль` можна дістатись до різноманітних функцій для роботи з величинами типу `Char`

```
>:module Data.Char

>ord 'a'
97

>chr 97
'a'

>toUpper 'a'
'A'

>isDigit 'a'
False

>isDigit '2'
True
```

2.4 Списки

Окрім вже розглянутих простих типів у `Haskell`'і є можливість означати складені типи. Одним з таких типів є список. Список — це структура для зберігання даних певного (одного) типу. Тип списку задають у формі `[Тип]`. Тому говорять про список цілих (`[Int]` або `[Integer]`) або дійсних (наприклад `[Double]`) чисел, список символів (`[Char]`), або навіть список списків (наприклад `[[Int]]` для списку, кожний елемент якого — список цілих чисел). Список може бути порожнім — `[]`, тобто не містити жодного елемента.

Список може бути заданий явно записом через кому всіх його елементів у квадратних дужках — `[2,12,3,45]`.

```
>:set +t

>[]
[]
it :: [a]

>[12,12,0,-4]
[12,12,0,-4]
```

```
it :: [Integer]

>[True,1==1,1==2,False]
[True,True,False,False]
it :: [Bool]

>['a','s','d']
"asd"
it :: [Char]

>[['a','s'],['d']]
["as","d"]
it :: [[Char]]

>"work"
"work"
it :: [Char]
```

Останні три приклади демонструють той факт, що рядок (тип `String`) у `Haskell`'і тотожний списку символів.

Список може бути заданий як інтервал з використанням `крапкової` нотації (`dot`-нотації):

```
>[-2..4]
[-2,-1,0,1,2,3,4]

>[1..5]
[1,2,3,4,5]

>[1,4..20]
[1,4,7,10,13,16,19]

>['a'..'f']
"abcdef"

>[1/2..5]
[0.5,1.5,2.5,3.5,4.5,5.5]

>[5..1]
[]

>[5,4..1]
[5,4,3,2,1]

>[25,24..20]
[25,24,23,22,21,20]
```

```
>[5..5]
[5]
```

В стилі `dot`-нотації можна задати й нескінченний список (наприклад `[1,3..]`), **але не в діалозі з інтерпретатором**, бо якщо інтерпретатор спробує вивести такий список, то йому знадобиться для цього нескінченний проміжок часу. Тобто інтерпретатор просто не зможе коректно завершити роботу.

Список складається з голови та хвоста. Перший елемент списку вважається головою, а всі решта — хвостом. При цьому хвіст сам є списком. Вважається, що порожній список не має голови, а його хвіст — порожній список.

Правоасоціативна операція `:` додає елемент (як голову списку):

```
>1 : []
[1]

>1 : 2 : 3 : 4 : []
[1,2,3,4]

>1 : (2 : (3 : (4 : [])))
[1,2,3,4]
```

Порожній список `[]` та правоасоціативна операція `:` — конструктори списку.

Для "склеювання" двох списків застовується правоасоціативна операція `++` :

```
>[10,20,30] ++ [1,2]
[10,20,30,1,2]

>[10,20,30] ++ []
[10,20,30]

>[] ++ [1,2]
[1,2]
```

Модуль `Prelude` містить низку функцій для роботи зі списками, серед них — `head, tail, length, reverse` та `concat`:

```
>head [2,3,1,4]
2
it :: Integer

>tail [2,3,1,4]
[3,1,4]
it :: [Integer]

>length [2,3,1,4]
4
it :: Int
```

```
>reverse [2,3,1,4]
[4,1,3,2]
it :: [Integer]
```

Елемент списку, який сам є списком, називають підсписком (даного списку). Так список `[[1,2],[],[0,2]]` містить три підсписки — `[1,2]`, `[]` та `[0,2]`.

Функція `concat` збирає всі елементи підсписків у один список (зменшує глибину списку).

```
>> :type [[[1],[2]],[],[[0],[2]]]
[[[1],[2]],[],[[0],[2]]] :: Num a => [[[a]]]

> :set +t

> concat [[[1],[2]],[],[[0],[2]]]
[[1],[2],[0],[2]]
it :: Num a => [[a]]

> concat (concat [[[1],[2]],[],[[0],[2]]])
[1,2,0,2]
it :: Num a => [a]

> concat [[1,2],[],[0,2]]
[1,2,0,2]
it :: Num a => [a]

> concat [['a','b'],[],['c','f']]
"abcf"
it :: [Char]
```

Функції для роботи зі списками можуть бути застосовані й до рядків:

```
>head "abcde"
'a'
it :: Char

>tail "abcde"
"bcde"
it :: [Char]

>length "abcde"
5
it :: Int

>reverse "abcde"
"edcba"
it :: [Char]
```

```
>last "abcde"
'e'
it :: Char

>init "abcde"
"abcd"
it :: [Char]

> "abcde" !! 0
'a'
it :: Char

>(!!) "abcde" 3
'd'
it :: Char

>'a' : "bcd"
"abcd"
it :: [Char]

>"ab" ++ "cd"
"abcd"
it :: [Char]
```

Функція `words` розбиває рядок, або відповідний список, на список слів, видаляючи всі пробільні символи. Функція `unwords` обернена до `words`:

```
>words "ab c \ncde"
["ab","c","cde"]
it :: [String]

> words "ab cde"
["ab","cde"]
it :: [String]

> words ['a','b',' ','c','d','e']
["ab","cde"]
it :: [String]

>unwords it
"ab cde"
it :: String
```

Функції `lines` та `unlines` видаляють чи додають тільки символ кінця рядка (нового рядка) \n

```
> lines "ab c \ncde"
["ab c ","cde"]
```

```
it :: [String]

> unlines it
"ab c \ncde\n"
it :: String
```

2.5 Кортежі

Іншим складеним типом Haskell'я є кортеж, який, на відміну від списку, не вимагає однотипності всіх елементів. Кортеж — впорядкований набір величин можливо різного типу. Кортеж може містити нуль елементів (0-кортеж `()`), два елементи (2-кортеж), три елементи (3-кортеж) і т.д. Але у всякому випадку кортеж не може містити рівно одного елемента.

Елементи кортежу задають в круглих дужках через кому. Тип кортежу визначається типами відповідних елементів. Так, якщо `a` та `b` певні довільні типи мови Haskell і перший елемент кортежу має тип `a`, а другий — тип `b`, то тип кортежу буде визначений як `(a,b)`. Далі в діалозі визначається тип 2-кортежа з двох елементів (двійка, пара) типу `Integer`. Тип кортежу — `(Integer, Integer)`.

```
>(1,3)
(1,3)
it :: (Integer, Integer)
```

Пара типу `(Integer, [Char])`:

```
>(5,"asd")
(5,"asd")
it :: (Integer, [Char])
```

Кортеж з трьох елементів (трійка). Тип кортежу — `(Char, Integer, Double)`.

```
>('a',1,1.5)
('a',1,1.5)
it :: (Char, Integer, Double)
```

Наступний приклад демонструє, залежність типу кортежу від порядку елементів.

```
>('a',1,1/5)
('a',1,0.2)
it :: (Char, Integer, Double)

>(1,'a',1/5)
(1,'a',0.2)
it :: (Integer, Char, Double)
```

Кортежі можуть складатись і з більшої кількості елементів та називатись, відповідно, четвірками, п'ятірками, тощо.

```
>([2,3,4],'a',"GHCi",5)
([2,3,4],'a',"GHCi",5)
it :: ([Integer], Char, [Char], Integer)

>([2,3,4], 'a', "GHCi", 1.2, 17)
([2,3,4],'a',"GHCi",1.2,17)
it :: ([Integer], Char, [Char], Double, Integer)
```

Для роботи з парами (і тільки з парами), використовують функції з Прелюдії — `fst` та `snd`, які повертають перший та другий елемент кортежу відповідно. В наступному прикладі вказані функції застосовуються до двійки, перший елемент якої — символ `'a'`, а другий — кортеж (двійка) `("asdf 5, 1.2, 17)`:

```
>fst ('a', ("asdf", 5, 1.2, 17))
'a'
it :: Char

>snd ('a', ("asdf", 5, 1.2, 17))
("asdf",5,1.2,17)
it :: ([Char], Integer, Double, Integer)
```

3 Функції

У мові програмування `Haskell` введення функції передбачає її *оголошення* та *означення*. Оголошення функції містить ім'я функції, її область визначення та область значень. Область визначення та область значень в оголошенні називають також *призначенням типу* або *сигнатурою типу*.

Синтаксис оголошення має таку форму

```
ім'я_функції :: область_визначення -> область значень
```

Наприклад `(Integer,Integer) -> Integer` є сигнатурою типу функції, яка на 2-кортежі цілих чисел обчислює ціле значення. Аргументи функції в `Haskell`'і не обов'язково брати в дужки. Так функція, що на двох цілих числах обчислює значення має тип `Integer -> Integer -> Integer`.

Визначення названих функцій може виглядати наприклад так:

```
f1 :: (Integer,Integer) -> Integer
f1 (x,y) = x*y

f2 :: Integer -> Integer -> Integer
f2 x y = x*y
```

Рядок оголошення `f2 :: Integer -> Integer -> Integer` читають так : *функція* `f2` **має тип** `Integer -> Integer -> Integer`. З сигнатури також зрозуміло, що функція `f2` має два аргументи типу `Integer` і повертає результат типу `Integer`.

Створимо файл для введення цих функцій (файл сценарію, сценарій, скрипт). Для цього можна скористатись командою інтерпретатора `:edit ім'я_файлу` для редагування (створення неіснуючого) файлу `func1.hs` – `:edit func1.hs`. З'явиться вікно текстового редактора *за замовчанням* для введення тексту. Після набору тексту та збереження файлу, необхідно закрити редактор та завантажити файл (вважатимемо, що шлях до нього `D:\ghc\hs\func1.hs`) командою `:load D:\\ghc\\hs\\func1.hs` дублюючи кожну обернену риску. Якщо запрошення `GHCi` не перепризначалось, то діалог виглядатиме так:

```
Prelude> :edit func1.hs
Ok, 0 module loaded.

Prelude> :load D:\\ghc\\hs\\func1.hs
[1 of 1] Compiling Main (D:\\ghc\hs\func1.hs, interpreted )
Ok, modules loaded: Main.
```

Відповідь `Ok, 0 module loaded` свідчить про успішне виконання редагування, а також про те, що жоден модуль не завантажений.

Запрошення інтерпретатора, після успішного завантаження файлу, змінилось на `*Main>`, бо `GHCi`, за відсутності оголошення модуля в файлі, вважає, що був завантажений модуль `Main`.

Скористаємось визначеними там функціями:

```
*Main> f1 (2,3)
6
*Main> f2 2 3
6
```

Виконавши команду `:edit func1.hs` доповнимо файл функцією з символами кирилиці в ідентифікаторах:

```
ф3 :: Integer -> Integer
ф3 змінна = змінна + змінна
```

Перезавантажимо файл (команда `:reload`) та скористаємось функцією `ф3`:

```
*Main> :reload
Ok, 1 module loaded.

*Main> ф3 7
14
```

Довідаємось про типи функцій:

```
*Main> :type f1
f1 :: (Integer, Integer) -> Integer

*Main> :type f2
```

```
f2 :: Integer -> Integer -> Integer

*Main> :type ф3
ф3 :: Integer -> Integer
```

Для оголошення модуля в першому непорожньому рядку файлу має бути вказане ім'я модуля, наприклад `module Test1 where`. При цьому ім'я файлу має бути, відповідно, – `Test1.hs`:

```
module Test1 where

f1 :: (Integer,Integer) -> Integer
f1 (x, y) =  x * y

f2 :: Integer -> Integer -> Integer
f2 x y = x*y

ф3 :: Integer -> Integer
ф3 змінна = змінна + змінна
```

Тоді успішне завантаження виглядатиме приблизно так:

```
Prelude> :load D:\\ghc\\hs\\Test1.hs
[1 of 1] Compiling Test1 (D:\\ghc\hs\Test1.hs, interpreted )
Ok, modules loaded: Test1.

*Test1>
```

Імена функцій та змінних у `Haskell`'і мають починатись з рядкової (малої) літери, інші символи – довільні літери латиниці чи кирилиці, цифри, знак підкреслювання (_) та апостроф (').

3.1 Умовні вирази

Для визначення функцій можуть використовуватись **умовні вирази**. Так, добре відома математична функція

$$signum(x) = \begin{cases} 1, & \text{x} > 0, \\ 0, & \text{x} == 0, \\ -1, & \text{x} < 0 \end{cases}$$

може бути означена, наприклад, з використанням умовного виразу

```
if bool_expr then expr1 else expr2
```

Незважаючи на синтаксичну подібність з операторами (statements) імперативних мов програмування, ця конструкція повертає **вираз** `expr1` у випадку, якщо `boolexpr` має значення `True`, або `expr2`, якщо `boolexpr` має значення

`False`, інакше – $\perp$. Отож, очевидно, що типи `expr1` та `expr2` мають бути тотожними, а тип `boolexpr` – `Bool`.

Для уникнення конфлікту імен зі стандартною (модуль `Prelude`) функцією, визначимо функцію

```
signum' :: Integer -> Integer
signum' x =  if x > 0 then 1
                      else if x < 0 then -1
                                    else 0
```

Тут бачимо використання апострофа в ідентифікаторі та застосування двовимірного синтаксису.

Виклик функції приводить до очікуваних результатів:

```
*Main> signum' 5
1

*Main> signum' 0
0

*Main> signum' (-1)
-1
```

Зауважимо, що у `signum' (-1)` дужки необхідні, щоб запобігти конфлікту пріоритетів застосування функції та унарного мінуса.

Мова `Haskell` має багато синтаксичних конструкцій для представлення умовних виразів, зокрема – охоронні вирази. Так розглянута функція може бути представлена у формі:

```
signum'' n | n > 0     = 1
           | n == 0    = 0
           | n < 0     = -1

*Main> signum'' 5
1

*Main> signum'' 0
0

*Main> signum'' (-1)
-1
```

Або з використанням `otherwise`

```
signum''' n | n > 0     = 1
            | n == 0    = 0
            | otherwise = -1
```