

МІНІСТУЕРСТВО ОСВІТИ І НАУКИ  
ХАРКІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ РАДІОЕЛЕКТРОНІКИ

ЗВІТ  
з виконання лабораторної роботи №1  
з дисципліни «Функціональне програмування»  
за темою «Знайомство з мовою функціонального програмування Haskell»

Виконав:  
Ковалик Вадим Валерійович

Перевірив:  
Міхаль Олег Пилипович

Харків 2024

## 1.1 Мета роботи

Ознайомлення з мовою та освоєння типових прийомів роботи мовою функціонального програмування Haskell.

## 1.2 Порядок виконання роботи

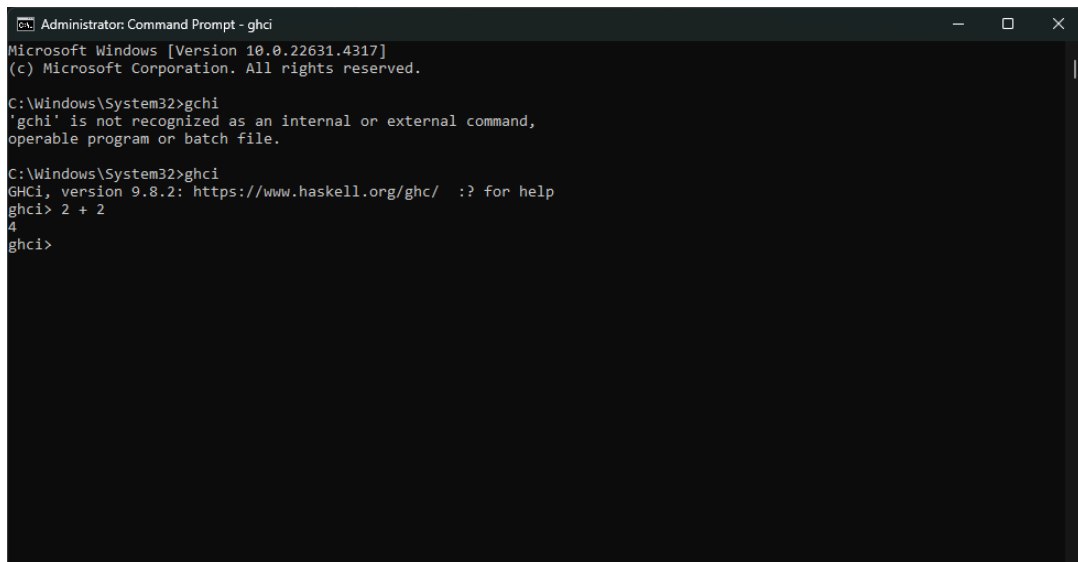
### 1.2.1 Завдання:

На даній лабораторній роботі необхідно виконати ознайомлення з методичними вказівками та запуск Haskell Platform для освоєння роботи в програмі. Потрібно вивчити синтаксис та елементи мови: примітивні типи, списки, кортежі та функції. Для практики слід копіювати приклади з додатка та налагоджувати їх, підставляти власні значення або створювати оригінальні приклади.

### 1.2.2 Хід роботи

#### 1.2.2.1 Запуск Haskell Platform

Для виконання завдань на мові Haskell необхідно встановити та запусити Haskell Platform, яка містить усі необхідні інструменти для розробки та виконання програм на цій мові. Для зручності виконання даної роботи буде використовувати редактор коду Visual Studio Code з необхідними розширеннями для роботи з мовою Haskell.



```
Administrator: Command Prompt - ghci
Microsoft Windows [Version 10.0.22631.4317]
(c) Microsoft Corporation. All rights reserved.

C:\Windows\System32>ghci
'ghci' is not recognized as an internal or external command,
operable program or batch file.

C:\Windows\System32>ghci
GHCi, version 9.8.2: https://www.haskell.org/ghc/ :? for help
ghci> 2 + 2
4
ghci>
```

Рисунок 1.1 – Відображення інтерактивного інтерпретатора Haskell

#### 1.2.2.2 Ознайомлення з основами мови Haskell

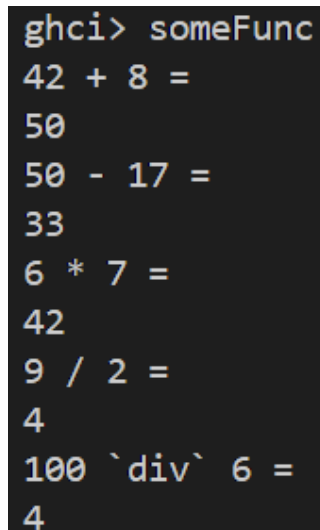
У Haskell примітивні типи включають числа, логічні значення, рядки та символи, а операції над ними мають доволі звичний для більшості мов програмування вигляд.

Числа оголошуються безпосередньо – просто записуються в коді. Наприклад, 42 позначає ціле число 42. Арифметичні операції, такі як додавання, віднімання, множення і ділення, працюють очікувано. Важлива відмінність операція ділення завжди повертає число з плаваючою комою, навіть якщо ділення націло. Для отримання цілочисельного результату використовують оператор `div`, як у прикладі `100 `div` 6`, що дає 16.

### Лістинг 1.1 – Використання арифметичних операцій

```
module Lib
  ( someFunc,
  )
where

someFunc :: IO ()
someFunc = do
  putStrLn "42 + 8 = "
  print (42 + 8 :: Integer)
  putStrLn "50 - 17 = "
  print (50 - 17 :: Integer)
  putStrLn "6 * 7 = "
  print (6 * 7 :: Integer)
  putStrLn "9 / 2 = "
  print (9 `div` 2 :: Integer)
  putStrLn "100 `div` 6 = "
  print (100 `mod` 6 :: Integer)
```



```
ghci> someFunc
42 + 8 =
50
50 - 17 =
33
6 * 7 =
42
9 / 2 =
4
100 `div` 6 =
4
```

Рисунок 1.2 – Результат виконання лістингу

Наступним примітивним типом є логічні значення які представлені `True` і `False`. Вони беруть участь у булевій алгебрі, де наприклад, операція заперечення змінює значення на протилежне. Порівняння, такі як `==` і `/=`, повертають логічні значення. Приклад `5 == 5` дасть `True`, а `8 /= 8` `False`.

## Лістинг 1.2 – Використання булевих операцій

```
module Lib
  ( someFunc,
  )
where

someFunc :: IO ()
someFunc = do
  putStrLn ("not True = " ++ show notTrue)
  putStrLn ("not False = " ++ show notFalse)
  putStrLn ("3 == 3 = " ++ show isThreeEqualThree)
  putStrLn ("5 /= 2 = " ++ show isFiveNotEqualTwo)
  putStrLn ("10 < 20 = " ++ show isTenLessThanTwenty)
  putStrLn ("7 > 9 = " ++ show isSevenGreaterThanNine)
  putStrLn ("True && False = " ++ show andOperation)
  putStrLn ("True || False = " ++ show orOperation)
  putStrLn ("(5 > 3) && not (10 == 20) = " ++ show complexExpression)

notTrue :: Bool
notTrue = not True

notFalse :: Bool
notFalse = not False

isThreeEqualThree :: Bool
isThreeEqualThree = (3 :: Integer) == (3 :: Integer)

isFiveNotEqualTwo :: Bool
isFiveNotEqualTwo = (5 :: Integer) /= (2 :: Integer)

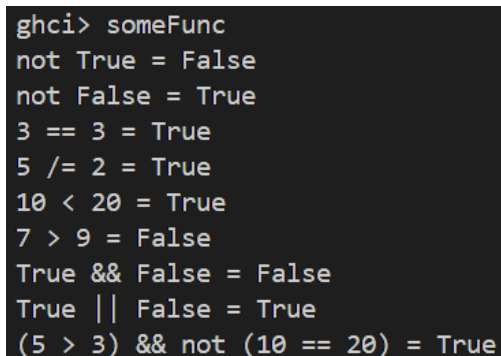
isTenLessThanTwenty :: Bool
isTenLessThanTwenty = (10 :: Integer) < (20 :: Integer)

isSevenGreaterThanNine :: Bool
isSevenGreaterThanNine = (7 :: Integer) > (9 :: Integer)

andOperation :: Bool
andOperation = True && False

orOperation :: Bool
orOperation = True || False

complexExpression :: Bool
complexExpression = ((5 :: Integer) > (3 :: Integer)) && not ((10 :: Integer) == (20 :: Integer))
```



```
ghci> someFunc
not True = False
not False = True
3 == 3 = True
5 /= 2 = True
10 < 20 = True
7 > 9 = False
True && False = False
True || False = True
(5 > 3) && not (10 == 20) = True
```

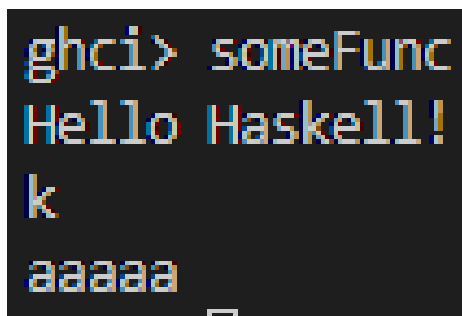
Рисунок 1.3 – Виконання функції someFunc

Рядки та символи – ще один важливий примітивний тип. Рядки укладаються в подвійні лапки, наприклад «Привіт, світ», а окремі символи – в одинарні лапки 'а'. Довгі рядки в одинарних лапках використовувати не можна – це викличе помилку. Рядки можна об'єднувати за допомогою оператора ++, як «Haskell» ++ « language», який поверне «Haskell language». Цікаво, що рядки в Haskell являють собою списки символів, тому можна звертатися до символів рядка за індексом: «Haskell» !! 3 поверне символ 'k'.

### Лістинг 1.3 – Використання рядків та символів

```
module Lib
  ( someFunc,
  )
where

someFunc :: IO ()
someFunc = do
  putStrLn ("Hello" ++ " Haskell" ++ "!")
  putStrLn (["Haskell" !! 3])
  putStrLn (replicate 5 'a')
```



```
ghci> someFunc
Hello Haskell!
k
aaaaa
```

Рисунок 1.4 – Виконання створеної програми

У Haskell списки і кортежі – це два ключові способи групувати дані, але вони мають різні властивості і призначення.

Списки складаються з елементів одного типу і можуть бути довільної довжини, завдяки ледачим обчисленням. Це означає, що елементи списку обчислюються тільки тоді, коли вони необхідні. Наприклад, під час доступу до тисячного елемента нескінченного списку буде обчислено тільки першу тисячу елементів, а інша елементів, а інша частина залишиться недоторканою.

Для створення списків можна використовувати кілька синтаксичних конструкцій:

- Явне перерахування: [10, 20, 30, 40, 50]

- Діапазони: [10..50]
- Нескінченні списки: [1, 2..], які тривають без кінця, але Haskell буде обчислювати їхні елементи тільки за необхідності.

Мова Haskell містить корисні операції та функції над списками такі як:

- Об'єднання списків: [10..50] ++ [51..60], що містить числа від 10 до 60
- Додавання елементів на початок списку: 100:[10..50]
- Звернення до елемента за індексом: [0..] !! 10 поверне 10
- Функція, що повертає перший елемент – head [10..50]
- Функція, що повертає список без першого елемента – tail [10..50]
- Функція, що повертає список без останнього елемента – init [10..50]
- Функція, що повертає список останній елемент – last [10..50]

А також списки мають зручні генератори списків:

- Без додаткових умов відбору: [x^2 | x <- [1..5]] – [1, 4, 9, 16, 25]
- З додатковими умовами відбору: [x | x <- [1..10], x `mod` 2 == 0] – [2, 4, 6, 8, 10]

Кортежі дозволяють зберігати значення різних типів, проте їх довжина завжди фіксована. Наприклад:

- ('a', 100)

Кортежі із двох елементів часто називають парою. Для доступу до елементів пари використовуються функції fst та snd:

- fst('x', 42) – 'x' (перший елемент)
- snd('x', 42) – 42 (другий елемент)

Кортежі можуть містити значення будь-яких типів:

- ("Haskell", True, 3.14)

## Лістинг 1.4 – Використання списків та кортежів

```
module Lib
  ( someFunc,
  )
where

type Student = (String, Int, Float)
```

```

students :: [Student]
students =
  [ ("Alina", 20, 4.5),
    ("Bogdan", 21, 3.8),
    ("Katerina", 19, 4.2),
    ("Dmytro", 22, 4.9)
  ]

getNames :: [Student] -> [String]
getNames = map \(name, _, _) -> name

averageGrade :: [Student] -> Float
averageGrade sts = sum (map \(_, _, grade) -> grade) sts / fromIntegral (length sts)

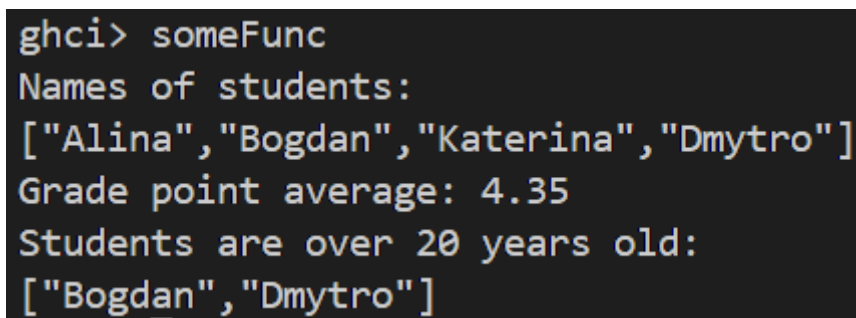
filterByAge :: Int -> [Student] -> [Student]
filterByAge ageLimit = filter \(_, age, _) -> age > ageLimit

someFunc :: IO ()
someFunc = do
  putStrLn "Names of students:"
  print (getNames students)

  let avgGrade = averageGrade students
  putStrLn $ "Grade point average: " ++ show avgGrade

  let olderStudents = filterByAge 20 students
  putStrLn "Students are over 20 years old:"
  print (getNames olderStudents)

```



```

ghci> someFunc
Names of students:
["Alina","Bogdan","Katerina","Dmytro"]
Grade point average: 4.35
Students are over 20 years old:
["Bogdan","Dmytro"]

```

Рисунок 1.5 – Вивід результату програми

У Haskell функції є ключовим інструментом для роботи з даними та побудови програми. Вони дозволяють створювати гнучкий і виразний код завдяки таким механізмам, як каррінг, анонімні функції, оператори та композиції функцій. Функції можуть мати кілька аргументів і можуть бути викликані в інфікській формі. Haskell підтримує визначення користувацьких операторів.

Охоронні вирази дозволяють обирати певну гілку виконання в залежності від умов. Зіставлення зі зразком допомагає обробляти різні форми вхідних даних. Для роботи зі списками часто використовують патерни.

Функції вищого порядку приймають інші функції як аргументи або повертають їх. Анонімні функції описуються через символ `\`. Функції `foldl` і `foldr` агрегують список у одне значення, а їх різниця проявляється при побудові нових списків.

Каррінг дозволяє створювати нові функції частковим застосуванням аргументів. Композиція функцій з оператором `(.)` об'єднує кілька функцій у ланцюжок. Функція `$` дозволяє уникати зайвих дужок у коді. Аналогічно, композицію можна використовувати для скорочення коду.

### Лістинг 1.5 – Використання функцій

```
module Lib
  ( someFunc,
  )
where

factorial :: Integer -> Integer
factorial n
  | n < 0 = error "Factorial is not defined for negative numbers"
  | n == 0 = 1
  | otherwise = n * factorial (n - 1)

factorials :: [Integer] -> [Integer]
factorials = map factorial

someFunc :: IO ()
someFunc = do
  let numbers = [0 .. 5]
  let results = factorials numbers
  print results
```

```
ghci> someFunc
[1,1,2,6,24,120]
```

Рисунок 1.6 – Відображення виконання програми

У Haskell система типів є надзвичайно потужною і строгою. Кожне вираження має тип, який може бути описаний за допомогою сигнатури. Сигнатура записується у формі `expression :: type signature`.

Для управління потоком виконання у Haskell використовуються вирази `if` та `case`. Вираз `if` вимагає, щоб обидві гілки мали однаковий тип, тоді як `case` дозволяє виконувати співставлення з образцем для обробки різних варіантів вхідних даних.

У Haskell немає традиційних циклів, замість цього використовуються рекурсія, відображення, фільтрація і згортка.



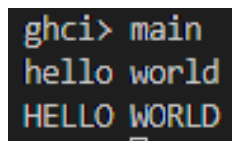
## Лістинг 1.6 – Виконана вправа реалізації функції `interact`

```
module Lib
  ( main,
  )
where

import Data.Char (toUpper)

myInteract :: (String -> String) -> IO ()
myInteract f = do
  input <- getLine
  let output = f input
  putStrLn output

main :: IO ()
main = myInteract $ map toUpper
```



```
ghci> main
hello world
HELLO WORLD
```

Рисунок 1.7 – Результат виконання

### 1.3 Висновок

Під час виконання лабораторної роботи ми ознайомилися з мовою Haskell та основами функціонального програмування. Haskell вирізняється суворою типізацією, рекурсією замість циклів та ледачими обчисленнями, що дає змогу працювати навіть із нескінченними списками. Опановано базові типи даних: примітивні значення, списки та кортежі. Важливими інструментами стали функції вищого порядку, каррінг і композиція, що спрощують роботу з даними.

Під час роботи ми зіткнулися з труднощами через сувору типізацію, що підкреслює важливість уважності. Загалом мова вразила своєю виразністю та чистотою, відкриваючи нові підходи до розв’язання складних задач.