Laboratorium 1.

1) GHCi: środowisko interaktywne Haskella (GHC interpreter, REPL)

1. W konsoli/terminalu wpisujemy kolejno następujące linie

```
$ mkdir haskell-lab1
$ cd haskell-lab1
$ which ghci ghc runghc
$ ghci --version
```

2. Uruchamiamy GHCi, wpisując w konsoli/terminalu

```
$ ghci
```

3. W konsoli GHCi wpisujemy

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

4. W konsoli GHCi wpisujemy kolejno

```
ghci> :help
ghci> :h
ghci> :?
```

   i przeglądamy dostępne polecenia.

   Uwaga: na początek najbardziej przydatne będą:
   `:load` (skrót `:l`), `:reload` (`:r`), `:type` (`:t`), `:info` (`:i`), `:quit` (`:q`)

5. W konsoli GHCi wpisujemy

```
ghci> :h [naciskamy TAB]
```

6. Dodajemy `e`

```
ghci> :he [naciskamy Enter]
```

7. W konsoli GHCi wpisujemy kolejno

```
ghci> 2
ghci> :t 3
ghci> :t True
ghci> :i Bool
ghci> :t (True, "hello")
```

```
ghci> 2 + 2
ghci> 2 + 3 * 5 + 1.4 / (2 * pi)
ghci> 5 `div` 2
ghci> div 5 2 -- porównujemy wynik z poprzednim
ghci> 5 `mod` 2
ghci> mod 5 2 -- porównujemy wynik z poprzednim
ghci> (sin 3)^2 + (cos 3)^2
ghci> 123 ^ 1234
ghci> 123.456 ^ 123
ghci> (exp 1) ^ 123
ghci> exp 1 ^ 123 -- porównujemy wynik z poprzednim
ghci> exp 1 ** pi
```

```
ghci> (+) 2 3
ghci> -3 + 3
ghci> 3 + -3
ghci> 3 + (-3)
```

```
ghci> True && False
ghci> True || False
ghci> True && 1
ghci> True /= False
ghci> not (3 >= 1)
ghci> not 3 >= 1
ghci> (3 > 2) && (2 > 1)
ghci> 3 > 2 && 2 > 1
```

8. W konsoli GHCi wpisujemy

```
ghci> 2 + [naciskamy Enter]
ghci> 2
```

9. W konsoli GHCi wpisujemy `:{` i naciskamy `[Enter]`, a następnie

```
Prelude| 2 +
Prelude| 2
```

   i na koniec `:}`
10. Naciskamy kilka razy strzałkę w górę a potem w dół (historia poleceń)
11. W konsoli GHCi wpisujemy

```
ghci> :set +t
```

12. Powtarzamy kilka z poprzednich obliczeń i analizujemy sposób prezentacji wyników.
    Co reprezentuje `it` ?
13. Powracamy do początkowego sposobu prezentacji wyników, wpisując w konsoli

```
ghci> :unset +t
```

14. W konsoli GHCi wpisujemy

```
ghci> let e = exp 1 -- definicja 'stałej'
ghci> e -- sprawdzamy wartość
ghci> let n = 123
ghci> let eToN = e ^ n
ghci> eToN
```

15. W konsoli GHCi wpisujemy

```
ghci> :show [naciskamy TAB]
```

16. Wybieramy kolejno

```
ghci> :show bindings
ghci> :show modules
```

i analizujemy wynik

2) Uruchamianie programów jako skryptów (runghc)

1. Tworzymy plik `ex2.hs` (np. `$ touch ex2.hs`) i wpisujemy w nim

```
printHello = putStrLn "Hello"
```

(i zapisujemy zmiany :)

2. W konsoli GHCi wpisujemy

```
ghci> :l ex2.hs
```

3. W konsoli GHCi wpisujemy

```
ghci> pri [naciskamy 2 razy TAB]
```

4. Wybieramy

```
ghci> printHello
```

5. Otwieramy nowe okno konsoli/terminala
6. Wpisujemy w nim kolejno

```
$ which runhaskell
$ which runghc
```

7. W oknie konsoli/terminala wpisujemy

```
$ runghc ex2.hs
```

i analizujemy komunikat błędu

8. Modyfikujemy plik ex2.hs

```
printHello = putStrLn "Hello"

main = printHello
```

(i zapisujemy zmiany :)

9. W oknie konsoli/terminala ponownie wpisujemy

```
$ runghc ex2.hs
```

10. W konsoli GHCi wpisujemy

```
ghci> ma [naciskamy TAB]
```

i analizujemy dostępne funkcje

11. W konsoli GHCi wpisujemy

```
ghci> :r -- reload the current module set
```

12. W konsoli GHCi ponownie wpisujemy

```
ghci> ma [naciskamy TAB]
```

i analizujemy dostępne funkcje

13. Wybieramy

```
ghci> main
```

3) Kompilacja pierwszego programu

1. W konsoli/terminalu wpisujemy kolejno

```
$ ghc --version
$ ghc --help
```

i analizujemy odpowiedzi

2. Sprawdzamy, czy bieżącym katalogiem jest `haskell-lab1`

```
$ pwd
```

jeśli nie, to przechodzimy do niego

3. W konsoli/terminalu wpisujemy kolejno

```
$ ls -a
$ ghc ex2.hs
$ ls -a
```

i analizujemy wyniki

4. W konsoli/terminalu wpisujemy

```
$ ./ex2
```

5. W konsoli/terminalu wpisujemy kolejno

```
$ ghc ex2.hs -o ex2Prog
$ ls -a
$ ./ex2Prog
```

i analizujemy wyniki

4) Proste funkcje jednej zmiennej

1. Tworzymy nowy plik np. `ex4.hs` i wpisujemy w nim

```
sqr :: Double -> Double
sqr x = x * x
```

2. W konsoli GHCi wpisujemy

```
ghci> :l ex4.hs
ghci> sqr 5
ghci> sqr 3 + sqr 4
ghci> (+) sqr 3 sqr 4
ghci> (+) (sqr 3) (sqr 4)
```

3. Dodajemy w pliku `ex4.hs` funkcję

```
vec2DLen :: (Double, Double) -> Double
vec2DLen (x, y) = sqrt (x^2 + y^2)
```

4. W konsoli GHCi wpisujemy

```
ghci> :r -- "reload the current module set"
ghci> vec2DLen (3,4)
ghci> let v1 = (3,4)
ghci> vec2DLen v1 -- jaki jest typ parametru funkcji vec2DLen?
```

5. ***Zadania***:
    1. Zdefiniować funkcję obliczającą długość wektora w 3D

    ```
    vec3DLen :: (Double, Double, Double) -> Double
    vec3DLen (x,y,z) = ...
    ```

    2. Zdefiniować funkcję

    ```
    swap :: (Int, Char) -> (Char, Int)
    ```

    3. Zdefiniować funkcję sprawdzającą, czy przekazane trzy liczby całkowite są równe

    ```
    threeEqual :: (Int, Int, Int) -> Bool
    threeEqual (x, y, z) = ...
    ```

    4. (*opcjonalne*) Zdefiniować (bez użycia `where`, `let...in`) funkcję obliczającą pole trójkąta, jeśli znane są długości `a`, `b` i `c` jego boków (wzór Herona).

5) Definicje funkcji: wyrażenie warunkowe `if...then...else`

1. Tworzymy nowy plik np. `ex5.hs` i wpisujemy w nim

```
sgn :: Int -> Int
sgn n = if n < 0
       then -1
       else if n == 0
            then 0
            else 1
```

2. W konsoli GHCi wpisujemy

```
ghci> :l ex5.hs
ghci> sgn 2
ghci> sgn 0
ghci> sgn -2
ghci> sgn (-2)
```

3. ***Zadania***:
    1. Używając `if...else` zdefiniować funkcję

    ```
    absInt :: Int -> Int -- absInt 2 = absInt (-2) = 2
    ```

    obliczającą wartość bezwzględną liczby całkowitej
    2. Używając `if...else` zdefiniować funkcję

    ```
    min2Int :: (Int, Int) -> Int -- min (1,2) = 1, min (-1, -1) = -1
    ```

    3. (*opcjonalne*) Zdefiniować funkcję `min3Int`
    4. (*opcjonalne*) Zdefiniować funkcję `min3Int` używając `min2Int`
    5. (*opcjonalne*) Zdefiniować funkcje

    ```
    toUpper :: Char -> Char
    toLower :: Char -> Char
    ```

    Pomocne mogą być funcje `toEnum` i `fromEnum`
    6. (*opcjonalne*) Zdefiniować funkcje

    ```
    isDigit :: Char -> Bool
    charToNum :: Char -> Int
    ```

7. (*opcjonalne*) Zdefiniować funkcje

```
romanDigit :: Char -> String
```

(tylko dla zakresu `1..9` :)

6) Definicje funkcji: guards

1. Tworzymy nowy plik np. `ex6.hs` i wpisujemy w nim

```
absInt :: Int -> Int
absInt n | n > 0 = n
         | n < 0 = -n
```

(i zapisujemy :)
2. W konsoli GHCi wpisujemy kolejno

```
ghci> :l ex6.hs
ghci> absInt 1
ghci> absInt (-1)
ghci> absInt 0
```

i analizujemy wyniki
3. Modyfikujemy funcję

```
absInt :: Int -> Int
absInt n | n >= 0    = n
         | otherwise = -n
```

(i zapisujemy zmiany :)
4. W konsoli GHCi testujemy działanie `absInt`
5. ***Zadania***:
    1. Używając "guards" zdefiniować funkcje

    ```
    sgn :: Int -> Int
    min3Int :: (Int, Int, Int) -> Int -- min (1,2,3)=1, min (1,1,3)=1
    ```

    2. (*opcjonalne*) Wykorzystując "guards" przepisać poprzednio zdefiniowane funkcje: `toUpper` , `toLower` , `isDigit` , `charToNum` , `romanDigit`

7) Definicje funkcji: dopasowanie wzorców

1. Tworzymy nowy plik np. `ex7.hs` i wpisujemy w nim

```
not' :: Bool -> Bool
not' True = False
not' False = True
```

(i zapisujemy :)
2. W konsoli GHCi wpisujemy kolejno

```
ghci> :l ex7.hs
ghci> not' True
ghci> not' ('y' == 'n') -- :)
ghci> not' (2 > 3 && 'y' == 'n')
```

3. W pliku `ex7.hs` dodajemy

```
isItTheAnswer :: String -> Bool
isItTheAnswer _       = False
isItTheAnswer "Love" = True -- :)
```

(i zapisujemy zmiany :)
4. W konsoli GHCi wpisujemy kolejno

```
ghci> :r
ghci> isItTheAnswer "42"
ghci> isItTheAnswer "Love"
```

5. Modyfikujemy definicję `isItTheAnswer`

```
isItTheAnswer :: String -> Bool
isItTheAnswer "Love" = True -- :)
isItTheAnswer _      = False
```

(i zapisujemy zmiany :)
6. Sprawdzamy działanie nowej wersji :)
7. ***Zadania***:
    1. Wykorzystując dopasowanie wzorców napisać definicje następujących funkcji logicznych

    ```
    or' :: (Bool, Bool) -> Bool
    and' :: (Bool, Bool) -> Bool
    nand' :: (Bool, Bool) -> Bool
    xor' :: (Bool, Bool) -> Bool
    ```

    2. (*opcjonalne*) Każdą z powyższych funkcji można zaimplementować w różny sposób. Czy istnieją ich optymalne implementacje (w sensie wydajności)?

8) Definicje funkcji: wyrażenie `case...of`

1. Tworzymy nowy plik np. `ex8.hs` i wpisujemy w nim

```
not' :: Bool -> Bool
not' b = case b of
          True  -> False
          False -> True
```

(zapisujemy plik :)

2. W konsoli GHCi wpisujemy

```
ghci> :l ex8.hs
```

i sprawdzamy działanie funkcji

3. W pliku `ex8.hs` dodajemy

```
absInt n =
 case (n >= 0) of
   True -> n
   _    -> -n
```

(zapisujemy plik :)

4. Sprawdzamy działanie funcji
5. ***Zadania***:
   Wykorzystując `case...of` napisać definicje funkcji

```
isItTheAnswer :: String -> Bool
not' :: Bool -> Bool
or' :: (Bool, Bool) -> Bool
and' :: (Bool, Bool) -> Bool
nand' :: (Bool, Bool) -> Bool
xor' :: (Bool, Bool) -> Bool
```

9) Definicje lokalne: klauzula `where`

1. Tworzymy nowy plik np. `ex9.hs` i wpisujemy w nim

```
roots :: (Double, Double, Double) -> (Double, Double)
roots (a, b, c) = ( (-b - d) / e, (-b + d) / e )
   where d = sqrt (b * b - 4 * a * c)
         e = 2 * a
```

(zapisujemy plik :)

2. W konsoli GHCi wpisujemy

```
ghci> :l ex9.hs
```

i sprawdzamy działanie funkcji dla różnych wartości `a,b,c`

3. ***Zadania***:
   1. Wykorzystując klauzulę `where` napisać definicję funkcji

      ```
      unitVec2D :: (Double, Double) -> (Double, Double)
      ```

      obliczającą wersor (wektor jednostkowy) podanego jako argument wektora 2D
   2. (*opcjonalne*) Napisać definicję funkcji `unitVec3D`
   3. (*opcjonalne*) Przepisać funkcję obliczającą pole trójkąta na podstawie wzoru Herona; wykorzystać `where` do definicji zmiennej lokalnej `p` (połowa obwodu trójkąta)

10) Definicje lokalne: wyrażenie `let...in`

1. Tworzymy nowy plik np. `ex10.hs` i wpisujemy w nim

```
roots :: (Double, Double, Double) -> (Double, Double)
roots (a, b, c) =
 let d = sqrt (b * b - 4 * a * c)
     e = 2 * a
 in ( (-b - d) / e, (-b + d) / e )
```

(zapisujemy plik :)

2. W konsoli GHCi wpisujemy

```
ghci> :l ex10.hs
```

i sprawdzamy działanie funkcji dla różnych wartości `a,b,c`

3. ***Zadania***:
   Przepisać rozwiązania zadań z poprzedniego punktu zamieniając `where` na `let...in`

11) Formatowanie kodu: off-side rule

1. Tworzymy nowy plik np. `ex11.hs` i wpisujemy w nim

```
printHello = putStrLn "Hello"
main = printHello
```

(zapisujemy plik :)

2. W konsoli GHCi wpisujemy

```
ghci> :l ex11.hs
ghci> printHello
ghci> main
```

3. Modyfikujemy plik `ex11.hs` dodając dwie spacje na początku obu linii

```
 printHello = putStrLn "Hello"
 main = printHello
```

(zapisujemy zmiany :)

4. W konsoli GHCi wpisujemy

```
ghci> :r
ghci> printHello
ghci> main
```

5. Modyfikujemy plik `ex11.hs` usuwając jendą spacje na początku 1. linii

```
printHello = putStrLn "Hello"
 main = printHello
```

(zapisujemy zmiany :)

6. W konsoli GHCi wpisujemy

```
ghci> :r
```

7. Modyfikujemy plik `ex11.hs` : usuwamy poprzednią zawartość i dodajemy

```
roots :: (Double, Double, Double) -> (Double, Double)
roots (a, b, c) = ( (-b - d) / e, (-b + d) / e )
   where d = sqrt (b * b - 4 * a * c)
         e = 2 * a -- uwaga na przesunięcie!
```

8. W konsoli GHCi wpisujemy

```
ghci> :r
```

9. ***Zadania***:
    1. Zmodyfikować klauzulę `where` w funkcji `roots` używając nawiasów `{}` do definicji granic bloku
    2. Dodać w wybranych (poprzednio utworzonych) plikach komentarze (jedno i wieloliniowe)
    3. (*opcjonalne*) Powtórzyć zmianę (dodanie `{}`) dla implementacji `roots` wykorzystującej `let...in`

12) Polimorfizm i inferencja typów

1. W konsoli GHCi wpisujemy kolejno (i analizujemy wyniki)

```
ghci> let f1 x = x -- proszę zwrócić uwagę na wykorzystanie 'let' w GHCi
ghci> :t f1 -- analizujemy wynik inferencji typu
ghci> let f2 x = True
ghci> :t f2
ghci> let f3 (x,y) = x + y
ghci> :t f3
ghci> :i Num
ghci> let f4 (x,y) = x / y
ghci> :t f4
ghci> :i Fractional
ghci> let f5 (x,y) = x /= y
ghci> :t f5
ghci> :i Eq
ghci> let f6 (x,y) = x > y
ghci> :t f6
ghci> :i Ord
ghci> let f7 (x,y) = if x > y then x + y else x / 4
ghci> :t f7
```

2. ***Zadania***:
    1. Napisać kilka (-naście?) definicji funkcji (bez jawnego podawania typu, jak powyżej) za każdym razem zgadując wynik wnioskowania typów
    2. Przemyśleć, czy warto (w takim razie) podawać (jawnie) definicje typu funkcji