Programming languages – Haskell

Introductory instruction 2024/25

T. Goluch

1. Introduction

The Haskell language implements the functional programming paradigm. The language consist of functions and constants. Functions are main language constructions. It can be easily passed as parameters to other functions. What may surprise is the fact that there are no variables, there are only constants. If we specify once that x = 5 then x is a constant and will not change its value anymore. Similarly, the functions are clean which means they have no side effects. Unlike methods in object-oriented languages, whose operation may depend on the state of the object, in Haskell for the same parameter values the function will always return the same result, as in the function in mathematics.

2. Compilation, run and debbuging

The most popular environments for developing Haskell projects include:
From the most popular environments allowing development of Haskell's projects, we can distinguish:

- GHCup – https://www.haskell.org/ghcup/ – Recommended as the main installer. Installs (by default in the C:\ghcup folder) the following programs:
  - ghcup - The Haskell toolchain installer,
  - ghc - The Glasgow Haskell Compiler,
  - msys2 - A linux-style toolchain environment required for many operations,
  - cabal - The Cabal build tool for managing Haskell software (by default in C:\cabal Windows folder),
  - stack (optional) - A cross-platform program for developing Haskell projects,
  - haskell-language-server (HLS) for development (optional) - A language server for developers to integrate with their editor/IDE.

  If you wish to use Visual Studio Code editor is recomended to install HLS.
  More help (first steps and what next): https://www.haskell.org/ghcup/steps/.
- The Haskell Tool Stack – https://docs.haskellstack.org/en/stable/README/.
- Cabal by Chocolatey on Wndows – https://hub.zhox.com/posts/introducing-haskell-dev/.

To check installed tools use text-based user interface (TUI): `ghcup tui` (command description in the bottom bar) or use commands:

- `ghc --version` or `stack ghc -- --version` (checking The Glorious GHC System version).
- `stack --version` (checking Stack version).

Both loops should be available from the command line:
`ghci` or `stack ghci`
In the initial phase of learning we can use the interactive REPL loop (read-eval-print loop):
`stack ghci` or `stack repl` (The Haskell Tool Stack) or `ghci` (Haskell Platform). After launching, a new cursor should be visible: `ghci>`. Inside the loop, we can still execute system commands, such as cleaning the screen in Windows: `ghci> :! cls`.
REPL loop is also available online: https://repl.it/repls/ExtraneousAdorableLink but the experience may be worse than with a locally installed environment. It can be particularly problematic to enter instructions consisting of several lines of code (e.g. when defining complex functions).

The REPL loop allows to incrementally compile and execute sequentially entered commands. For example, to add a constant named x and assign its value 5, we issue a command[1]:

`> let x = 5` or `x = 5`

To check if a constant exists and what value it has, just enter its name:

```
> x
5
```

It's possible also to check variable type (command `:type` or shorter `:t`):

```
> :type x
x :: Num p => p
```

To pass a command consisting of several lines, put it between `:{` i `}:`, for instance:

```
> :{
| let fib 0 = 0
|     fib 1 = 1
|     fib n = fib (n-1) + fib (n-2)
| :}
```

```
> :{
| fib 0 = 0
| fib 1 = 1
| fib n = fib (n-1) + fib (n-2)
| :}
```

Note that after typing the `:{` instruction the cursor `>` should change to `|` and stay until you type: `}:`.

```
> fib 6
8
```

Calculations for longer sequences will quickly turn out to be ineffective. In such case memorization can help:

```
> :{
| memoized_fib = (map fib [0 ..] !!)
|    where fib 0 = 0
|          fib 1 = 1
|          fib n = memoized_fib (n-2) + memoized_fib (n-1)
| :}
```

Leaving ghci: `> :quit`

It will be more convenient to use files with the .hs extension and load them into ghci during run larger programs.

Sample Hello world code in Haskell (hello.hs):

```
hello = do putStrLn "Hello, what is your name?"
           name <- getLine
           putStrLn ("Hello, " ++ name ++ "!")
```

Please be careful of indentation, they are very important because they determine the range of expressions. Code which is part of some expression should be indented further in than the beginning of that expression (even if the expression is not the leftmost element of the line). Please do not use tabs in indentations.

Loading and runing hello.hs:

`ghci hello.hs`

Information: `Ok, one module loaded` and cursor `*Main>` should appear. Now we can call main function (the only one defined in the module /program):

`*Main> hello`

Środowisko dostarcza debbuger który niestety nie należy do najbardziej intuicyjnych. Posłóżmy się jeszcze raz programem obliczającym n-ty ciąg fibonachiego:

```
fib 0 = 0
fib 1 = 1
fib n = fib (n-2) + fib (n-1)
```

After loading it as a module, we can set the breakpoint:

```
ghci> :break 3
Breakpoint 0 activated at D:\_GIT\JP_PL\lab\introHaskell\main.hs:3:9-29
```

Then we invoke the function:

`ghci> fib 3`

The program stops when it first time encounters the indicated line and displays the contents of the variables:

```
Stopped in Main.fib, D:\_GIT\JP_PL\lab\introHaskell\main.hs:3:9-29
```

[1] all commands that needs to be called in REPL are preceded by a cursor character: `>`

```
_result :: a = _
n :: Integer = 3
```
W can see where you currently are:
```
[D:\_GIT\JP_PL\lab\introHaskell\main.hs:3:9-29] ghci> :list
2  fib 1 = 1
3  fib n = fib (n-2) + fib (n-1)
           ^^^^^^^^^^^^^^^^^^^^^
```
To continue, use the command:
```
 [D:\_GIT\JP_PL\lab\introHaskell\main.hs:3:9-29] ghci> :continue
Stopped in Main.fib, D:\_GIT\JP_PL\lab\introHaskell\main.hs:3:9-29
_result :: a = _
n :: Integer = 2
[D:\_GIT\JP_PL\lab\introHaskell\main.hs:3:9-29] ghci> :continue
2
ghci> :continue
```
If the program is no longer able to wait at the breakpoint, the appropriate message will be displayed:
```
not stopped at a breakpoint
ghci>\JP_PL\lab\introHaskell\main
```

Haskell also allows to write programs compiled into native files for various operating systems. Compiling and running hello.hs using **ghc** (you need to leave ghci):
```
ghc --make hello.hs
```
`hello.exe` (Windows)
`./hello` (Linux)

Creating a simple project using the **Stack** environment: `stack new <project_name> simple`.
```
cd <project_name>
```
Downloading the compiler to an isolated location so it doesn't interfere with any other Haskell installation on the system:
```
stack setup
```
Build the executable:
```
stack build
```
After building the binary, you need to go to the folder containing it (the information will be displayed in the last line):
Installing executable <project_name> in drive:<path_to_project>\<project_name>\.stack-work\install\<8_digit_hexagonal_random_number >\bin) e.g.:
```
cd .stack-work\dist\29cc6475\build\<project_name>
<project_name>.exe
```

Let's try loading a program calculating the nth row of Pascal's triangle[2] and containing several functions:
```
factorial :: Integer -> Integer
factorial n = if (n == 1 || n==0) then 1 else  n * factorial(n-1)

newton :: Integer -> Integer -> Integer
newton n k = div (factorial(n)) (factorial(n-k)*(factorial(k)))

pascal :: Integer -> [Integer]
pascal n = [newton n x | x <- [0..n]]
```
```
*Main> factorial 6
720
*Main> newton 6 3
20
*Main> pascal 6
[1,6,15,20,15,6,1]
```
Unfortunately, when compiling this code, we obtain an error that the main function has not been implemented. It is required because it is started by default at the program startup.
```
factorial :: Integer -> Integer
    ...
main :: IO ()
main = do print ("Enter Pascal's row number")
          n <- readLn
```

[2] Description of the Pascal Triangle can be found here: https://en.wikipedia.org/wiki/Pascal%27s_triangle

```
        print (pascal (n :: Integer))
```

Please add the main function and then compile and run program.

3. Advanced IDE

Much better conditions for software development are provided by the IDE **Visual Studio Code** available for various platforms (Windows, Linux, Mac): https://code.visualstudio.com/download.

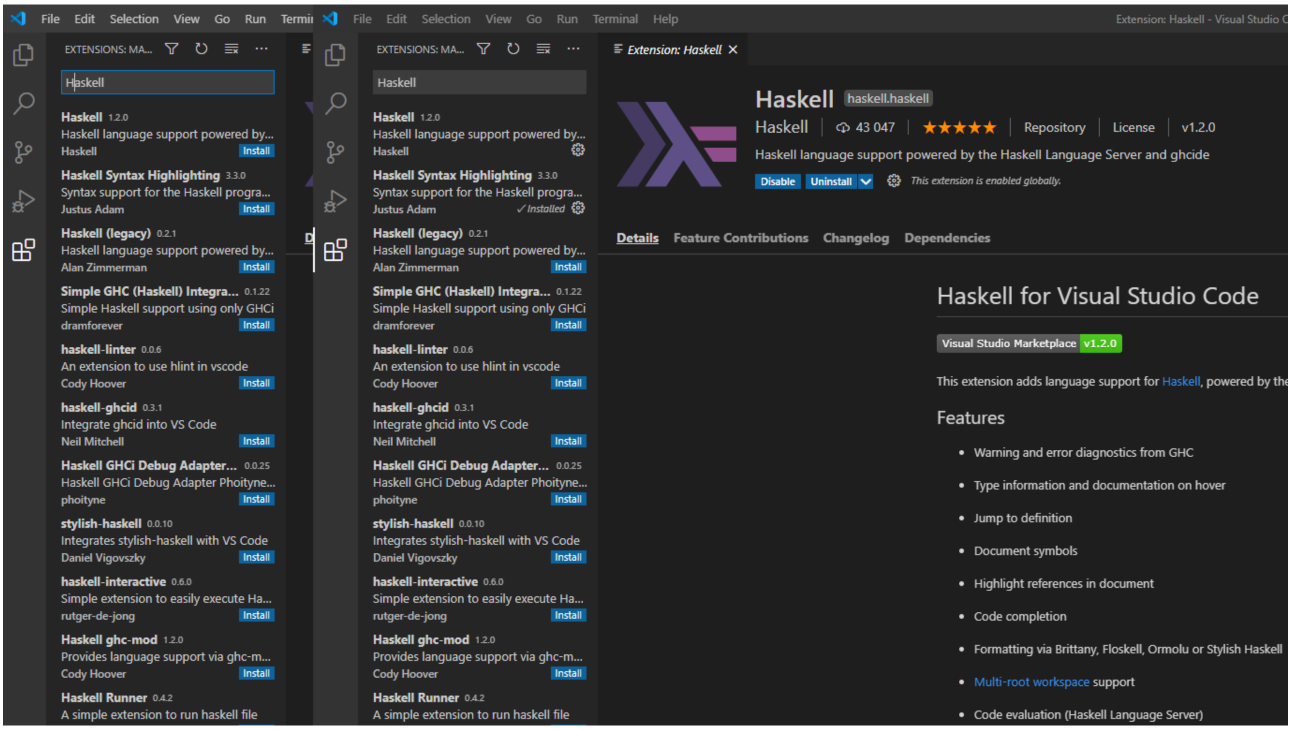
Starting Visual Studio from the project folder:

```
code .
```

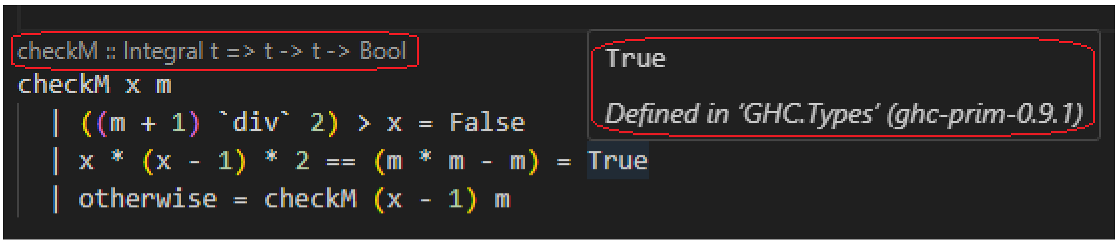
In order to install the Haskell plugin in Visual Studio Code, use the keyboard shortcut Ctrl + Shift + x and type: Haskell. A list of recommended plugins will appear:

- **Haskell (for Visual Studio Code)** – Warning and error diagnostics from GHC, type information and documentation on hover, jump to definition, document symbols, highlight references in document, code completion, formatting via Brittany, Floskell, Ormolu or Stylish Haskell, multi-root workspace support, code evaluation. Requiring a working ghcup installation.

To install the selected plug-in, choose the button Install



Type information and documentation on hover:



Hint in action, suggests using guards instead of extensive if-else conditional statements:

```haskell
checkM x m =
    if ((m + 1) `div` 2) > x
        then False
        else if x * (x - 1) * 2 == (m * m - m)
            then True
            else checkM (x - 1) m
```

```
Use guards
Found:
  checkM x m
  = if ((m + 1) `div` 2) > x then
        False
    else
        if x * (x - 1) * 2 == (m * m - m) then True else checkM (x - 1) m
Why not:
  checkM x m
  | ((m + 1) `div` 2) > x = False
  | x * (x - 1) * 2 == (m * m - m) = True
  | otherwise = checkM (x - 1) m
 hlint(refact:Use guards)
```

4. Types and Typeclasses

Definitions of simple and complex data types start with a capital letter. Haskell has following types:

`> :t True`

`True :: Bool` – logical type

`> :t 'a'`

`'a' :: Char` – character type

`> :t "abc"::`

`"abc" :: [Char]` – identical with `String`

`Int`, `Float` i `Double` are similar to types we know from C and C ++.

`Inteeger` type with arbitrary-precision is limited only by computer resources:

```
> a = 815915283247897734345611269596115894272000000000 :: Integer
> a
815915283247897734345611269596115894272000000000
> :t a
a :: Integer
```

We also have typeclasses:

`> :t 1`

`1 :: Num p => p` – class of numeric types `Num` (with addition and multiplication operations defined).

`Integral` – like `Num`, buts standard instances are integer (with modulo and whole-number division and remainder operations defined)

```
> :t div
div :: Integral a => a -> a -> a
> :t 1.0
```

`1.0 :: Fractional p => p` – class of `Fractional` types like `Num`, but not only integer(with division operator defined).

```
> :t (/)
(/) :: Fractional a => a -> a -> a
> :t pi
```

`pi :: Floating a => a` – class of floating point types `Floating`.

Other typeclasses:

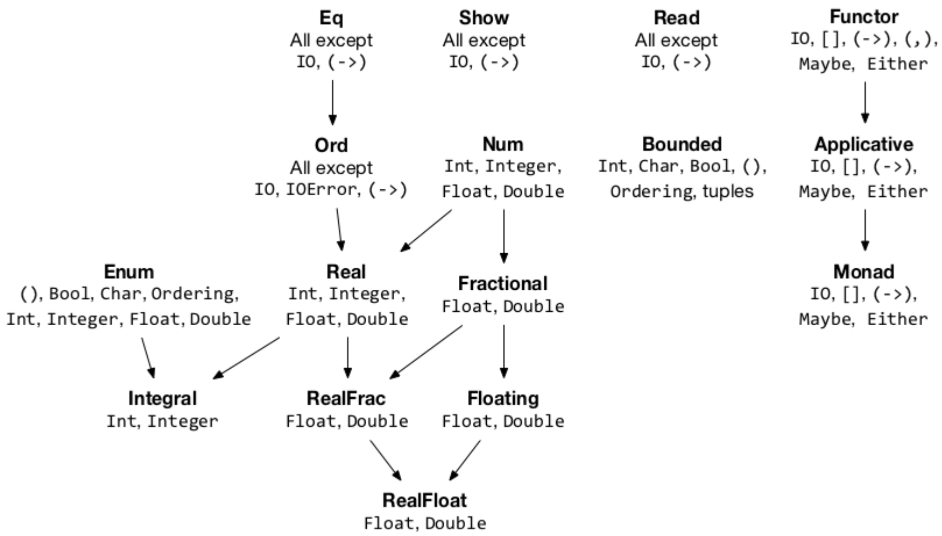
`Eq` – comparable values (with Equivalence relation defined)

```
> 5.0 == 5
True
>  :t (==)
(==) :: Eq a => a -> a -> Bool
```

`Ord` – ordered values (with a linear order relation defined)

```
> 5.1 < 5
False
> :t (<)
```

```
(<) :: Ord a => a -> a -> Bool
```
Show – type values that can be displayed.
```
> :t show
show :: Show a => a -> String
```
Relationships between typeclasses and types are shown in the following diagram:



source: https://cs.anu.edu.au/courses/comp1100/labs/10/

We can easily define our own simple data types:
```
> data Color = Red | Green | Blue
> :{
| isRed :: Color -> Bool
| isRed Red = True
| isRed _ = False
| :}
isRed Blue
False
```
And also complex:
```
> data Chars = Value Char | Join Chars Chars
> :{
| isChar :: Chars -> Bool
| isChar (Value x) = True
| isChar _ = False
| toString :: Chars -> String
| toString (Join x y) = (toString x) ++ (toString y)
| toString (Value x) = [x]
| :}
> toString (Value 'a')
"a"
> toString (Join (Value 'a') (Value 'b'))
"ab"
> toString (Join (Join (Value 'a') (Value 'b')) (Value 'c'))
"abc"
```

5. Functions

Function is a first-class citizen and it cannot start with a capital letter because this notation is reserved for data types. We got to know the constant function called the definition:

`> let x = 2 :: Integer` or shorter: `> x = 2 :: Integer`

However, this time we explicitly declared the variable type as `Integer` (operator :: – we read as: "is of type") We can check it out.
```
> :t x
x :: Integer
```
Form of declaration of each function :

```
function_name :: Type_qualifiers = >arg_type - >arg_2_type ->… - >arg_n_type -> Result_type
```

We can explicitly provide a function definition. This is useful if you want to specify the types of arguments and return value:

```
> add :: Int a -> a  -> a
```

Definition looks as follows:

```
function_name arg_1 arg_2 ... _arg_n = let function_definitions in the command returning the result
```

The definition of a typical function:

`> let add x y = x + y` or shorter: `> add x y = x + y`

Such defined function can be called by passing arguments:

`> add 2 5` or `> add x 5` or `> add x y`

Output: `7`

We might as well check the type of function:

```
> :t add
add :: Num a => a -> a -> a
```

This means that it is a function that accepts two identical parameters from `Num` typeclass and returns an element of the same type. The language contains a rich set of useful functions:

`> succ 8` - successor

```
9
```

`> pred 6.457` - predecessor

```
5.457
```

`> min 9 10` - minimum

```
9
```

`> max 2.4 3` - maximum

```
3.0
```

`> gcd 21 6` – greatest common divisor

```
3
```

`> lcm 21 6` – least common multiple

```
42
```

`> even 7` – is even

```
False
```

`> odd 7` – is odd

```
True
```

Mathematical functions are also available: `exp`, `log`, `sqrt`, `logBase`, `sin`, `cos`, `tan` and π constant:

```
> exp 1
2.718281828459045
> log (exp 1)
1.0
> pi
3.141592653589793
```

The functions are not redefinable (within their own scope) so the code below will not work.

```
> :{
| x = 2
| x = 3
| main = print x
| :}
```

The order of the functions in the file does not matter.

```
> :{
| main = print x
| x = 123
| :}
> main
123
```

Lazy evaluation - calculations are made ONLY when it is necessary.

```
> :{
| divide :: (RealFrac a) => a -> a -> a
| divide x y = let q = (/) x y in if (abs y) < 0.001 then 0 else q
| :}
> divide 5 0.00099
0.0
> divide 5 0.001
5000.0
```

Function instantiation.

```
> rewerse = divide 1
> rewerse 5
0.2
```

Function composition.

```
> identity = rewerse . rewerse
> identity 5
5.0
```

Functions can be passed as parameters to other functions. Selecting only elements specified by the function `f` (called a predicate) on all elements of `x` list (`filter f x`).

```
> filter (>3) [-10..10]
[4,5,6,7,8,9,10]
> filter (\x -> x * x == 1) [-10..10]
[-1,1]
```

6. Opertators

In Haskell we have the following operators defined:

`> 5 + 8` – addition
`13`
`> 5 - 8` – subtraction
`-3`
`> 5 * 2` – multiplication
`10`
`> 5 / 2` – division
`2.5`
``> 5 `div` 2`` – whole-number division
`2`
``> 5 `mod` 2`` – modulo
`1`
``> (-7) `mod` (-3)`` – The modulo result will have divisor sign (second operator).
`-1`
``> -7 `mod` (-3)`` – Please note that functions have priority over operators and will be executed: ``-(7 `mod` (-3))``
`2`
``> 5 `rem` 2`` – remainder
`1`
``> 7 `rem` (-3)`` – The remainder result differs from modulo only for negative numbers.
`1`
``> (-7) `rem` (-3)`` – The remainder result will have dividend sign (first operator).
`-1`

By specifying the name of the binary function in backward apostrophes, we can put it in place of the binary operator in infix notation. You can see it on the example of `div` and `mod` but we can also do it with the add function defined by us:

```
5 `add` 2
7
```

Some operators have alternative versions implemented for other types:

`> 2.5 ^ 4` or `> 2.5 ^^ 4` – power with numeric type exponent
`39.0625`
`> 2 ** 5.3` – power with fractional type exponent
`39.396621227037315`

We can also check the operator type:

```
> :t (^)
(^) :: (Integral b, Num a) => a -> b -> a
> :t (**)
(**) :: Floating a => a -> a -> a
```

Precedence and connectivity of operators in Haskell is presented in the following table[3]:

Precedence	left associative	non-associative	right associative
9	!!		.

[3] https://www.haskell.org/onlinereport/haskell2010/haskellch4.html#x10-820004.4.2

8			^ , ^^ , ⋆⋆
7	⋆, /, ‘div‘,‘mod‘, ‘rem‘, ‘quot‘		
6	+, -		
5			:, ++
4		==, /=, <, <=, >, >=, ‘elem‘, ‘notElem‘	
3			&&
2			||
1	>>, >>=		
0			$, $!, ‘seq‘

Operators with a higher precedence will be executed earlier:

```
> 2 ** 5 + 1
33
> False && True || True
True
```

In order to force a different order of execution, we must use brackets:

```
> 2 ** (5 + 1)
64.0
> False && (True || True)
False
```

Because of the precedence of functions over operators, calling our add function may have an undesirable effect:

`> 2 * 2 `add` 2` lub `> add 2 2 * 2` lub `> 2 * add 2 2`

```
8
```

7. Expressions: if … then … else, let…in, where and |

In Haskell, every expression and function must return something, so the `else` clause in the `if` statement is mandatory.

```
> threshold100 x = if x > 100 then 100 else x
> threshold100 87
87
> threshold100 102.8
100.0
```

Version of `factorial` implementation using conditional statement `if` … `then` … `else`.

```
> factorial n = if n == 1 then 1 else n * factorial (n - 1)
> factorial 40
815915283247897734345611269596115894272000000000
```

`factorial` function type.

```
> :t factorial
factorial :: (Eq p, Num p) => p -> p
```

Version with guards:

```
factorial' n' a | n' == 1 = a | otherwise = factorial' (n' - 1) (a * n')
```

The above one-line version may be useful when entering GHCi, but in program it is recommended to use a more readable form:

```
factorial n
  | n == 1 = 1
  | otherwise = n * factorial (n - 1)
```

Version of `factorial` implementation with use of an additional variable (accumulator) passing intermediate results between recursive calls.

```
factorial' n' a
  | n' == 1 = a
  | otherwise = factorial' (
```

```
> factorial' 40 1
815915283247897734345611269596115894272000000000
```

Versions of factorial implementation using `let` … `in` to hide the accumulator.

```
> factorial n = let factorial' n' a = if n' == 1 then a else factorial' (n' - 1) (a * n') in
factorial' n 1

> :{
| factorial n = let factorial` n' a | n' == 1 = a
|                                   | otherwise = factorial` (n' - 1) (a * n')
|                                     in factorial` n 1
```

```
| :}
```

```
> :{
| factorial n = factorial` n 1 where factorial` n' a | n' == 1 = a
|                                                      | otherwise = factorial` (n' - 1) (a * n')
| :}
```

8. Lists, tuples, maps and filters

Containing module: `> import Data.List`
Lists are homogeneous structures of variable length.

```
[1,2,3,4]
> 1:2:3:4:[]
[1,2,3,4]
> 1:2:[3, 4]
[1,2,3,4]
> [1, 2 .. 4]
[1,2,3,4]
> [1 .. 4]::[Int]
[1,2,3,4]
> numbers = [3,6,4,8,1]
> numbers
[3,6,4,8,1]
> [3,6,4,8,1] ++ [6,8]
[3,6,4,8,1,6,8]
> "Tomasz " ++ "Goluch"
"Tomasz Goluch"
> 'k':"not"
"knot"
> 'a':' ':'k':"not"
"a knot"
```

`> [1..7]!!3` – element at index 3

```
4
> let listOfLists = [[1,2,3],[5],[1,2],[]]
> listOfLists
[[1,2,3],[5],[1,2],[]]
> listOfLists ++ [[1,1]]
[[1,2,3],[5],[1,2],[],[1,1]]
> [5,7]:listOfLists
[[5,7],[1,2,3],[5],[1,2],[]]
> [3,2,1] > [2,1,0]
True
> [3,2,1] > [3,2,2]
False
> [3,2,1] > [2,3,3]
True
> [3,2,1] > [3,2]
True
> [3,2,1] == [3,2,1]
True
```

Typical functions performing on lists.

```
> head [3,2,1,2]
3
> tail [3,2,1,2]
[2,1,2]
> last [3,2,1,2]
2
> init [3,2,1,2]
[3,2,1]
> length [3,2,1,2]
4
> null [3,2,1,2]
False
> reverse [3,2,1,2]
[2,1,2,3]
```

```
> take 2 [3,2,1,2]
[3,2]
> drop 2 [3,2,1,2]
[1,2]
> maximum [3,2,1,2]
3
> sum [3,2,1,2]
6
> product [3,2,1,2]
12
> elem 5 [3,2,1,2]
False
> [5..15]
[5,6,7,8,9,10,11,12,13,14,15]
> ['x'..'z']
"xyz"
> [3,5..15]
[3,5,7,9,11,13,15]
> [0.1, 0.3 .. 1]
[0.1,0.3,0.5,0.7,0.8999999999999999,1.0999999999999999]
> take 10 (cycle [3,2,1])
[3,2,1,3,2,1,3,2,1,3]
> take 5 (repeat 3)
[3,3,3,3,3]
> replicate 5 3
[3,3,3,3,3]
> [2^x | x <- [0..8]]
[1,2,4,8,16,32,64,128,256]
> [2^x | x <- [0..8], 2^x > 30]
[32,64,128,256]
> [x | x <- [0..20], x `mod` 5 == 0]
[0,5,10,15,20]
> [x | x <- [0..20], x `mod` 5 == 0, x `mod` 3 == 0]
[0,15]
> evenOrOdd xs = [ if x `mod` 2 == 0 then "even" else "odd" | x <- xs, x < 10]
> evenOrOdd [2..20]
["even","odd","even","odd","even","odd","even","odd"]
> [x*y | x <- [2,5,10], y <- [1..5]]
[2,4,6,8,10,5,10,15,20,25,10,20,30,40,50]
> [x*y | x <- [2,5,10], y <- [1..5], x*y <= 30]
[2,4,6,8,10,5,10,15,20,25,10,20,30]
> let names = ["Alvin", "Bruce","John"]
> let surnames = ["Lee","Springsteen"]
> [name ++ " " ++ surname | name <- names, surname <- surnames]
["Alvin Lee","Alvin Springsteen","Bruce Lee","Bruce Springsteen","John Lee","John Springsteen"]
> length' xs = sum [1 | _ <- xs]
> length' [1..100]
100
> onlyLowercase st = [ c | c <- st, c `elem` ['a'..'z']]
> onlyLowercase "UPPERCASE lowercase"
"lowercase"
> 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]]
> [[ x | x <- xs, even x ] | xs <- xxs, length xs > 9]
[[2,4,2,6,2,6]]
```

Implementation of function reversing the order of list items:

```
> :{
| reverse::[a] -> [a]
| reverse [] = []
| reverse (x:y) = reverse y ++ [x]
| :}
```

Tuples are heterogeneous structures of constant length

```
> (1, 'a', "number")
(1, 'a', "number")
> fst ("Nicholson", 1937)
"Nicholson"
> snd ("Nicholson", 1937)
```

```
1937
> zip [1,2,3,4,5] ['a','b','c']
[(1,'a'),(2,'b'),(3,'c')]
> zip [1..] ['a','b','c']
[(1,'a'),(2,'b'),(3,'c')]
```

A list of different triples a, b, c, such that a + b + c <= x, and from sides of length a, b, c you can build a right triangle.

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

Infinite Fibonacci sequence:

```
> fibonacci = 1:1:[(a + b) | (a, b) <- zip fibonacci (tail fibonacci)]
```

`> print fibonacci` causes an infinite loop

`> print (take 1000 fibonacci)` it works – because of lazy evaluation

Performing the function `f` on all elements of the list `x`

```
> map' f x = [f xs | xs <- x]
> map' something [1..10]
> map' (+1) [1..10]
> map' (\x -> 3 * x + 2) [1..10]
```

9. Tables[4]

Containing module: `> import Data.Array`

Creating `sqr` array containing the squares of numbers from 1 to 100.

```
> sqr = array (1,100) [(i, i*i) | i <- [1..100]]
```

Take the square of the number 7.

```
> sqr!7
49
```

Checking the dimensions of the array.

```
> bounds sqr
(1,100)
```

Recursively creating an array containing Fibonacci numbers (try find and correct the error).

```
> fibs n = a where a = array (0,n) ([(0, 1), (1, 1)] ++ [(i, a!(i-2) + a!(i-1)) | i <- [2..n]])
```

Take 10th sequence element.

```
> fibs 10 ! 10 => 89
```

10. Hash tables[5]

Containing module:

```
> import Data.Map
```

Creating an H table from an exemplary pair list.

```
> :{
| H :: Map Char Int
| H = fromList [('a', 3), ('b', 5), ('c', 7)]
| :}
```

Search element x in the table H

```
output :: Maybe a
output = lookup x H
```

11. Function: group, sort

The group function takes a list and returns a list consisting of lists of grouped identical subsequent items.

```
> :{
| group :: Eq a => [a] -> [[a]]
| group "Mississippi" = ["M","i","ss","i","ss","i","pp","i"]
| :}
```

[4] https://www.haskell.org/tutorial/arrays.html
[5] http://hackage.haskell.org/package/base-4.3.0.0/docs/Data-HashTable.html

The sort function implements a stable version of the sorting algorithm.

```
> :{
| sort :: Ord a => [a] -> [a]
| sort "Mississippi" = "Miiiippssss"
| :}
```

Functions require importing `Data.List` module.

12. Modules

Haskell contains a lot of modules that extend language capabilities. The first step is to download the current package list:

`cabal update` or `cabal v2-update`

Installation of numbers module:

`cabal install numbers`

Displaying list of installed modules:

`cabal list --installed`

Importing module in ghci loop:

```
> import Data.Number.BigFloat
```

Module usage:

```
Data.Number.BigFloat> pi = 3.14159265358979323846264338327950288419716939937510 :: BigFloat Prec50
Data.Number.BigFloat> :t pi
pi :: BigFloat Prec50
Data.Number.BigFloat> pi
3.14159265358979323846264338327950288419716939937510e0
```

13. Sample laboratory tasks

1) For a given number n, find the sum of all Fibonacci sequence even elements less than n.
2) For a given (as a lists) sets L and M, calculate their difference and symmetrical difference.
3) For a given number n and a list L of numbers, find the sum of all numbers from 1 to n divisible by at least one number from L.

Comments:

a) all functions should have an appropriate header with the type of function,
b) in programs you cannot use functions from Data.List, Data.Array or similar modules.

Haskel in browser:

- http://tryhaskell.org (limited GHCi, only as an introduction)
- http://ideone.com
- http://rextester.com/l/haskell_online_compiler
- https://www.jdoodle.com/execute-haskell-online
- https://paiza.io/en/projects/new?language=haskell
- https://www.tutorialspoint.com/compile_haskell_online.php
- https://repl.it/repls/ExtraneousAdorableLink (GHCi loop)

Tutorials:

- https://wiki.haskell.org/A_brief_introduction_to_Haskell
- https://wiki.haskell.org/Tutorials
- https://learnxinyminutes.com/docs/haskell
- http://learnyouahaskell.com/chapters
- http://exercism.io/languages/haskell/about (registration required)
- https://wiki.haskell.org/Introduction_to_IO (introduction to IO)
- https://wiki.haskell.org/IO_inside (IO inside)
- https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/ghci.html (GHCi users guide)

- https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/ghci.html#ghci-commands (HGCi commands)
- https://betterprogramming.pub/haskell-vs-code-setup-in-2021-6267cc991551 (Haskell VS Code Setup in 2021)