1) Установите интерпретатор **ghci**.

```
C:\Users\iverendeev>ghci
GHCi, version 9.4.2: https://www.haskell.org/ghc/  :? for help
ghci>
```

2) Вычислите в интерпретаторе следующие выражения:

a. `1 + 1`

```
ghci> 1 + 1
2
```

b. `True == False`

```
ghci> True == False
False
```

c. `5 + 7 * (-7 + 1)`

```
ghci> 5 + 7 * (-7 + 1)
-37
```

d. `not True`

```
ghci> not True
False
```

e. `False && True || True`

```
ghci> False && True || True
True
```

f. `2 * 2 /= 4`

```
ghci> 2 * 2 /= 4
False
```

g. `succ 9`

```
ghci> succ 9
10
```

h. `max 5 15 + 4 + min 9 (-1)`

```
ghci> max 5 15 + 4 + min 9 (-1)
18
```

3) Используя только функцию **`max`** из модуля **Prelude**:

```
max::Int->Int->Int
max x y = `если x > y, то x, иначе y`
```

напишите выражение, которые находит максимальное число, среди списка чисел:

```
0, -2, 99, 11, -5, 6, 1521
```

```
ghci> :{
ghci| maxNum :: Ord a => [a] -> a
ghci| maxNum [x] = x
ghci| maxNum (x:x':xs) = maxNum((max x x'):xs)
ghci| :}
ghci> maxNum [2, 3, 4]
4
ghci> maxNum [0, -2, 99, 11, -5, 6, 1521]
1521
```

4) Создаёте модуль **FACTORIAL**:

```
module FACTORIAL where
```

```
fac::Int->Int
fac 0 = 1
fac n = n * fac (n - 1)
```

Загрузите модуль в интерпретатор, использую команду:

```
:load `имя_файла_модуля`
```

Команда **:load** будет искать указанный файл в текущем каталоге.

```
ghci> :l fact.hs
[1 of 1] Compiling FACTORIAL        ( fact.hs, interpreted )
Ok, one module loaded.
```

5) Вычислите **fac 5**

```
ghci> fac 5
120
```

6) Используя **условное выражение** напишите функцию, которая возвращает **0**, если её аргумент больше **5**, иначе возвращает **1**.

```
ghci> :{
ghci| gt_than_5 :: Int -> Int
ghci| gt_than_5 x = if x > 5 then 0 else 1
ghci| :}
```

7) Реализуйте функцию **max'**.

```
ghci> :{
ghci| my_max :: Int -> Int -> Int
ghci| my_max x y = if x > y then x else y
ghci| :}
```

8) Напишите выражение, которое конкатенирует строки: **"Hello"**, **" "**, **"world"**, **"!"**.

```
ghci> "Hello" ++ " " ++ "world" ++ "!"
"Hello world!"
```

9) Используя команду **:info (++)** определите ассоциативность оператора (++).

```
(++) :: [a] -> [a] -> [a]        -- Defined in `GHC.Base'
infixr 5 ++
```

Правая ассоциативность

10) Используя команды **:info (++)** и **:info (^)** определите какой оператор имеет больший приоритет **(+)** или **(^)**.

```
ghci> :info ++ ^
(++) :: [a] -> [a] -> [a]        -- Defined in `GHC.Base'
infixr 5 ++
(^) :: (Num a, Integral b) => a -> b -> a        -- Defined in `GHC.Rea
l'
infixr 8 ^
```

5 против 8 => у ^ приоритет выше

11) Расставьте скобки в выражении **2^4 + 6 * 3^2 – 1**, в соответствии с приоритетом операций в Haskell.

(((2^4) + (6 * (3^2))) – 1)

12) Используя команду **:t `выражение`** определите тип
    a. функции **max**;

```
ghci> :t max
max :: Ord a => a -> a -> a
```

b. функции **head;**

```
ghci> :t head
head :: GHC.Stack.Types.HasCallStack => [a] -> a
```

c. выражения **2 + 2**;

```
ghci> :t 2 + 2
2 + 2 :: Num a => a
```

d. оператора **(^)**.

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

13) Запишите все операторы в выражении в **префиксной** нотации:

a. 1 + 2

```
ghci> (+) 1 2
3
```

b. 4^2 – 1

```
ghci> (-) ((^) 4 2) 1
15
```

c. 5 == 6 – 1

```
ghci> (==) 5 ((-) 6 1)
True
```

14) Запишите все операторы выражения в **инфиксной** нотации:

a. max 5 (-1)

```
ghci> 5 `max` (-1)
5
```

b. min (min 5 2) (max (-1) 4)

```
ghci> (5 `min` 2) `max` ((-1) `max` 4)
4
```

15) Запишите список чисел

a. **[2,3,4,5]** без перечисления всех элементов списка.

```
ghci> [2..5]
[2,3,4,5]
```

b. **[1,3,5,7,9]** без перечисления всех элементов списка.

```
ghci> [1, 3..9]
[1,3,5,7,9]
```

16) Напишите выражение, которое вычисляет из списка **[1, 5, -3, 3, 5]**:

a. голову списка;

```
ghci> head [1, 5, -3, 3, 5]
1
```

b. хвост списка;

```
ghci> tail [1, 5, -3, 3, 5]
[5,-3,3,5]
```

c. новый список, который состоит из первых трёх элементов списка;

```
ghci> [head([1, 5, -3, 3, 5])] ++ [head(tail([1, 5, -3, 3, 5]))] ++ [head(tail(tail([1, 5, -3, 3, 5])))]
[1,5,-3]
```

d. новый список, который состоит из квадратов элементов списка (используйте, генераторы списка)

```
ghci> [i * i | i <- [1, 5, -3, 3, 5]]
[1,25,9,9,25]
```

e. новый список, который содержит умноженные на 2 элементы списка, которые при умножении на три больше 8.

```
ghci> [i * 2 | i <- [1, 5, -3, 3, 5], i * 3 > 8]
[10,6,10]
```

17) Опишите функцию, которая для данного числа **n** создаёт список всех попарных сумм чисел от **1** до **n** (Т.е. **[1+1, 1+2, 1+3,…,1+n, 2+1, 2+2,…,n+n]** всего **n*n** элементов)

```
ghci> :{
ghci| my_func :: Int -> [Int]
ghci| my_func n = [x + y | x <- [1..n], y <- [1..n]]
ghci| :}
ghci> my_func 4
[2,3,4,5,3,4,5,6,4,5,6,7,5,6,7,8]
```

18) Перепишите функцию используя охранные выражения:

```
f::Int->Int->Int

f x y =

  if x > y + 1       then 0 else

  if x > y - 2       then 1 else

  if x * 2  == y + 1 then 2 else

  if x + 10 == y - 2 then 3 else 4
```

Используйте конструкцию **where**, чтобы ввести локальные определения для подвыражений **y + 1** и **y – 2**

19) Используя сравнение с образцом реализуйте функции:

```
fst’ :: (a, b, c)-> a
```

```
ghci| fst' :: (a, b, c) -> a
ghci| fst' (a, b, c) = a
ghci| :}
ghci> fst' (1, 2, 3)
1
```

```
snd’ :: (a, b, c)-> b
```

```
ghci> :{
ghci| snd' :: (a, b, c) -> b
ghci| snd' (a, b, c) = b
ghci| :}
ghci> snd' (1, 2, 3)
2
```

`thd' :: (a, b, c)-> c`

```
ghci| thd' :: (a, b, c) -> c
ghci| thd' (a, b, c) = c
ghci| :}
ghci> thd' (1, 2, 3)
3
```

Получающие соответственно первую, вторую и третью проекцию тройки значений.

20) Реализуйте функцию

`gcd::Int->Int->Int`,

которая находит наибольший общей делитель своих аргументов.

```
ghci> :{
ghci| gcd :: Int->Int->Int
ghci| gcd 0 x = x
ghci| gcd x y = gcd (mod y x) x
ghci| :}
ghci> gcd 4 5
1
```

21) Реализуйте функцию

`delete :: Char -> String -> String`,

которая принимает на вход строку и символ и возвращает строку, в которой удалены все вхождения символа.

Пример: **`delete 'l' "Hello world!"`** должно возвращать **`"Heo word!"`**.

```
ghci> :{
ghci| remove_char :: Char -> String -> String
ghci| remove_char pat str = [ch | ch <- str, ch /= pat]
ghci| :}
ghci> remove_char 'a' "baraban"
"brbn"
```

22) Реализуйте функцию

`substitute :: Char -> Char -> String -> String`,

которая заменяет в строке указанный символ на заданный.

Пример: **`substitute 'e' 'i' "eigenvalue"`** возвращает **`"iiginvalui"`**.

```
ghci> :{
ghci| substitute :: Char -> Char -> String -> String
ghci| substitute x y str = [if i == x then y else i | i <- str]
ghci| :}
ghci> substitute 'a' 'o' "baraban"
"borobon"
ghci>
```

23) Реализуйте две функции с использование хвостовой рекурсии. Все необходимые вспомогательные функции должны быть определены при помощи конструкции let.

Числа Фибоначчи

```
fib :: (Eq p, Num p) => p -> p
fib n = ???
```

```
my_fib :: (Eq p, Num p) => p -> p
my_fib n =
  let
    helper curr prev n | n == 0 = curr
                       | otherwise  = helper (curr+prev) curr (n-1)
    in helper 0 1 n
```

Обращение списка

```
reverse :: [a] -> [a]
reverse list = ???
```

```
my_reverse :: [a] -> [a]
my_reverse list =
  let
    helper [] t = t
    helper (x:xs) t = helper xs (x:t)
  in helper list []
```

24) Напишите рекурсивную функцию

```
balance :: String → Bool,
```

проверяющую балансировку скобок в строке. Как вы должны помнить, строки в Haskell — это список символов **`[Char]`**.

Например, для следующих строк функция должна вернуть **`True`**:

```
(if (zero? x) max (/ 1 x))
I told him (that it's not (yet) done). (But he wasn't listening)
```

А для строк ниже, напротив, функция должна вернуть значение False:

```
:-)
())(
```

Последний пример демонстрирует: недостаточно проверить только, что строка содержит равное количество открывающих и закрывающих скобок.

```
ghci> :{
ghci| isBalanced :: String -> Bool
ghci| isBalanced str =
ghci|     let
ghci|         go (-1) (x:xs) = False
ghci|         go cnt "" | cnt == 0 = True
ghci|                   | otherwise = False
ghci|         go cnt (x: xs) | x == '(' = go (cnt + 1) xs
ghci|                         | x == ')' = go (cnt - 1) xs
ghci|                         | otherwise = go cnt xs
ghci|      in go 0 str
ghci| :}
ghci> isBalanced "(())"
True
ghci> isBalanced "(())((()"
False
```

25) Реализуйте четыре функции:

`reverseAll::[[a]]->[[a]]`

функция, получающая на вход списочную структуру и обращающая все её элементы, а также её саму.

```
ghci> :{
ghci| reverseAll :: [[a]] -> [[a]]
ghci| reverseAll list =
ghci|     let
ghci|         go acc [] = acc
ghci|         go acc (x:xs) = go ((reverse x):acc) xs
ghci|     in go [] list
ghci| :}
ghci> reverseAll [[1, 2, 3], [5, 6, 7]]
[[7,6,5],[3,2,1]]
```

`firstElem::(Eq a) => a -> [a] -> Int`

функция, возвращающая номер первого вхождения заданного атома в список.

```
firstElem :: (Eq a) => a -> [a] -> Int
firstElem item list =
  let
    go :: (Eq a) => a -> [a] -> Int -> Int -> Int
    go elem [] idx target            = target
    go elem (x: xs) idx target  | x == elem && target == -1 = go elem xs (idx + 1) idx
                                | otherwise                 = go elem xs (idx + 1) target
  in go item list 0 (-1)
```

`set::(Eq a) => [a] -> [a]`

функция, возвращающая список из всех атомов, содержащихся в заданном списке. Каждый атом должен присутствовать в результирующем списке в единственном числе.

```
my_set :: (Eq a) => [a] -> [a]
my_set list =
  let
    go [] new = reverse new
    go (x:xs) new = if x `elem` new then go xs new else go xs (x:new)
  in go list []
```

```
freq::(Eq a) => [a] -> [(a, Int)]
```

функция, возвращающая список пар (символ, частота). Каждая пара определяет атом из заданного списка и частоту его вхождения в этот список.

```
import Data.Map

my_freq :: (Ord a) => [a] -> [(a, Int)]
my_freq list = toList $ fromListWith (+) [(s, 1) | s <- list]
```