Functional Programming Concept with

Tutorial for

Programming Language Laboratory (CS 431)

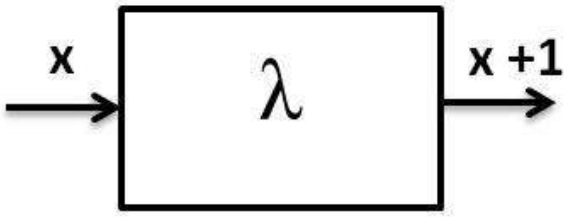
September – November 2020

Functional Programming(FP)

- Key Idea - computation as ‘evaluation of mathematical functions’
  - Idea originated from Lambda Calculus formalism

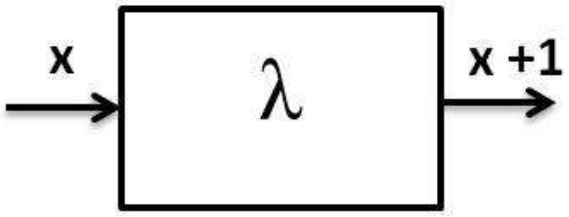
Functional Programming(FP)

- Key Idea - computation as ‘evaluation of mathematical functions’
  - Idea originated from Lambda Calculus formalism



Functional Programming(FP)

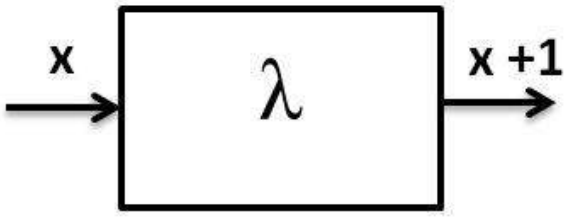
- Key Idea - computation as ‘evaluation of mathematical functions’
  - Idea originated from Lambda Calculus formalism



$\lambda x.\ x+1$

Functional Programming(FP)

- Key Idea - computation as ‘evaluation of mathematical functions’
  - Idea originated from Lambda Calculus formalism

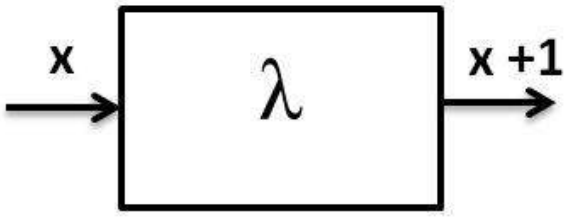


$\lambda x.\ x+1$

True:

Functional Programming(FP)

- Key Idea - computation as ‘evaluation of mathematical functions’
  - Idea originated from Lambda Calculus formalism

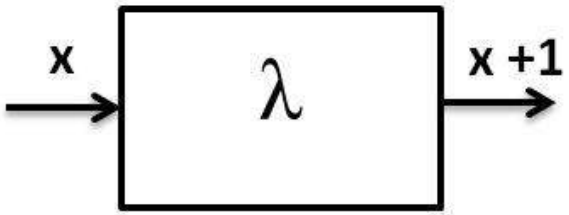


$\lambda x.\ x+1$

True: $\lambda x.\lambda y.x$

Functional Programming(FP)

- Key Idea - computation as ‘evaluation of mathematical functions’
  - Idea originated from Lambda Calculus formalism



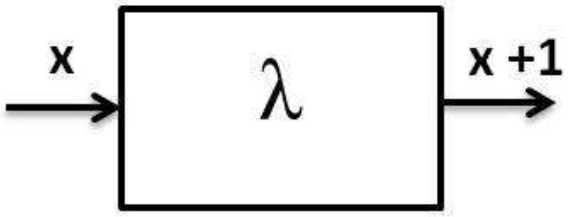
$\lambda x.\ x+1$

True: $\lambda x.\lambda y.x$

False:

Functional Programming(FP)

- Key Idea - computation as ‘evaluation of mathematical functions’
  - Idea originated from Lambda Calculus formalism



$\lambda x.\ x+1$

True: $\lambda x.\lambda y.x$

False: $\lambda x.\lambda y.y$

Functional Programming(FP)

- Key Idea - computation as ‘evaluation of mathematical functions’
  - Idea originated from Lambda Calculus formalism
- Languages that follow functional programming paradigm
  - Haskell
  - LISP
  - Python
  - Erlang
  - Racket
  - F#
  - Clojure
  - Scala

Functional Programming

- Key Idea - computation as 'evaluation of mathematical functions'
  - Idea originated from Lambda Calculus formalism
- Languages that follow functional programming paradigm
  - Haskell
  - LISP
  - Python
  - Erlang
  - Racket
  - F#
  - Clojure
  - Scala

we are going with Haskell this time

Haskell

- Standardized purely functional programming language
- Named after logician and mathematician Haskell Brooks Curry
- History
  - First version (“Haskell 1.0”) was introduced in 1990
  - The latest standard of Haskell is “Haskell 2010”

Haskell - Features

- Purely functional
- Statically typed
- Type inference
- Lazy
- Concurrent
- Packages

Purely functional

- Every function in Haskell is a function in the mathematical sense (i.e., "pure")
  - The pure function returns the same output every time for the same input
  - In a pure functional language, you can't do anything that has a side effect

Purely functional

```
function impure(str: string){
        str = str + “Post”;
        print(str);
        return(str);
}
```

State of function gets changed

Ex. Impure function

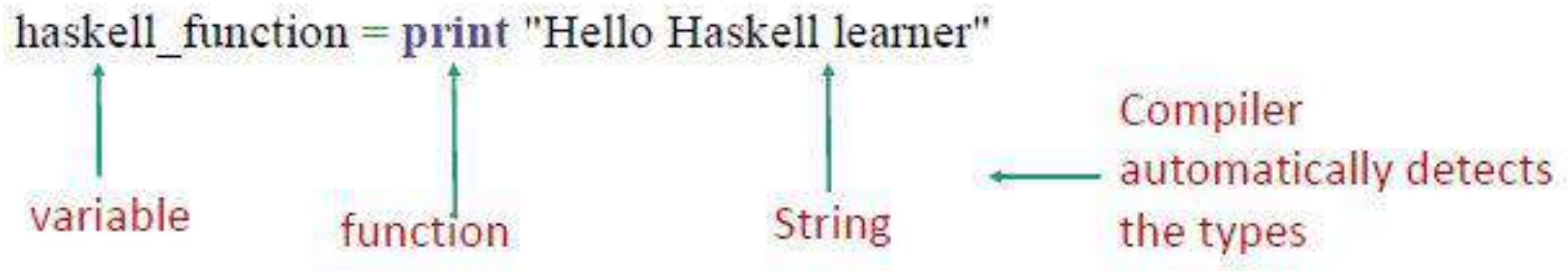
```
function impure(str: string){
        return(str + “Post”);
}
```

No change in state | Immutable

Ex. Pure function

Statically Typed

- Every expression in Haskell has a type which compile time is determined at compile time
  - The compiler knows which piece of code is a number, which is a string and so on



Statically Typed

- All the types composed together by function application have to match up. If they don't, the program will be rejected by the compiler

- Ex. addMe :: Int -> Int -> Int ⟵ type signature or function declaration
- addMe x y = x+y ⟵ function definition

```
*Main> addMe 4 5
9
```

```
*Main> addMe 4 5.5

<interactive>:23:9: error:
    • No instance for (Fractional Int) arising from the literal '5.5'
    • In the second argument of 'addMe', namely '5.5'
      In the expression: addMe 4 5.5
      In an equation for 'it': it = addMe 4 5.5
```

Type Inference

- You don't have to explicitly label every piece of code because the type system can intelligently figure it out

  Eg. If we write a=5+4

  Haskell will automatically infer that a is a number

Lazy

- Nothing is evaluated unless it has to be

  Eg. Function call : f 5 (29^35792)

Both the x and y values are evaluated and passed to function f

Non lazy languages like C or Java

Haskell pass the arguments value as it is without doing any actual computation of 29^35792

Haskell

Saves on CPU usage and user's time!

Concurrency

- Functional programming, by its nature (lack of side effect), is suitable for parallelism
- Concurrency in Haskell is mostly done with Haskell threads
- The Glasgow Haskell Compiler (GHC), comes with concurrency library containing a number of useful concurrency primitives and abstractions technique called Software Transactional Memory (STM)
- STM is an alternative to the lock based synchronization, whose basic objective is to evaluate a set of expression in isolated manner

Haskell - Packages

- Open source contribution to Haskell is very active with a wide range of packages available on the public package servers
- There are 6,954 packages freely available; for instances

bytestring	Binary data	base	Prelude, IO, threads
network	Networking	text	Unicode text
parsec	Parser library	directory	File/directory
hspec	RSpec-like tests	attoparsec	Fast parser
monad-logger	Logging	persistent	Database ORM
template-haskell	Meta-programming	tar	Tar archives

Haskell - Application

Haskell - Application

Haskell - Application

- facebook anti-spam programs

Haskell - Application

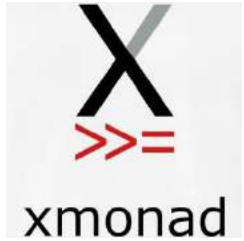
- facebook anti-spam programs

Haskell - Application

- facebook anti-spam programs
- 

a window manager for the X Window System

Haskell - Application

- facebook anti-spam programs
- 

a window manager for the X Window System

darcs

Haskell - Application

- facebook anti-spam programs
- xmonad a window manager for the X Window System
- darcs a revision control system

Some other FP Applications

Some other FP Applications

Some other FP Applications

- 

→Scala

Some other FP Applications

- 

→Scala

Some other FP Applications

- 

→Scala
- 

Erlang

Some other FP Applications

- 

→Scala
- 

Erlang

Some other FP Applications

- 

→Scala
- 

Erlang
- 
 Lisp

Haskell

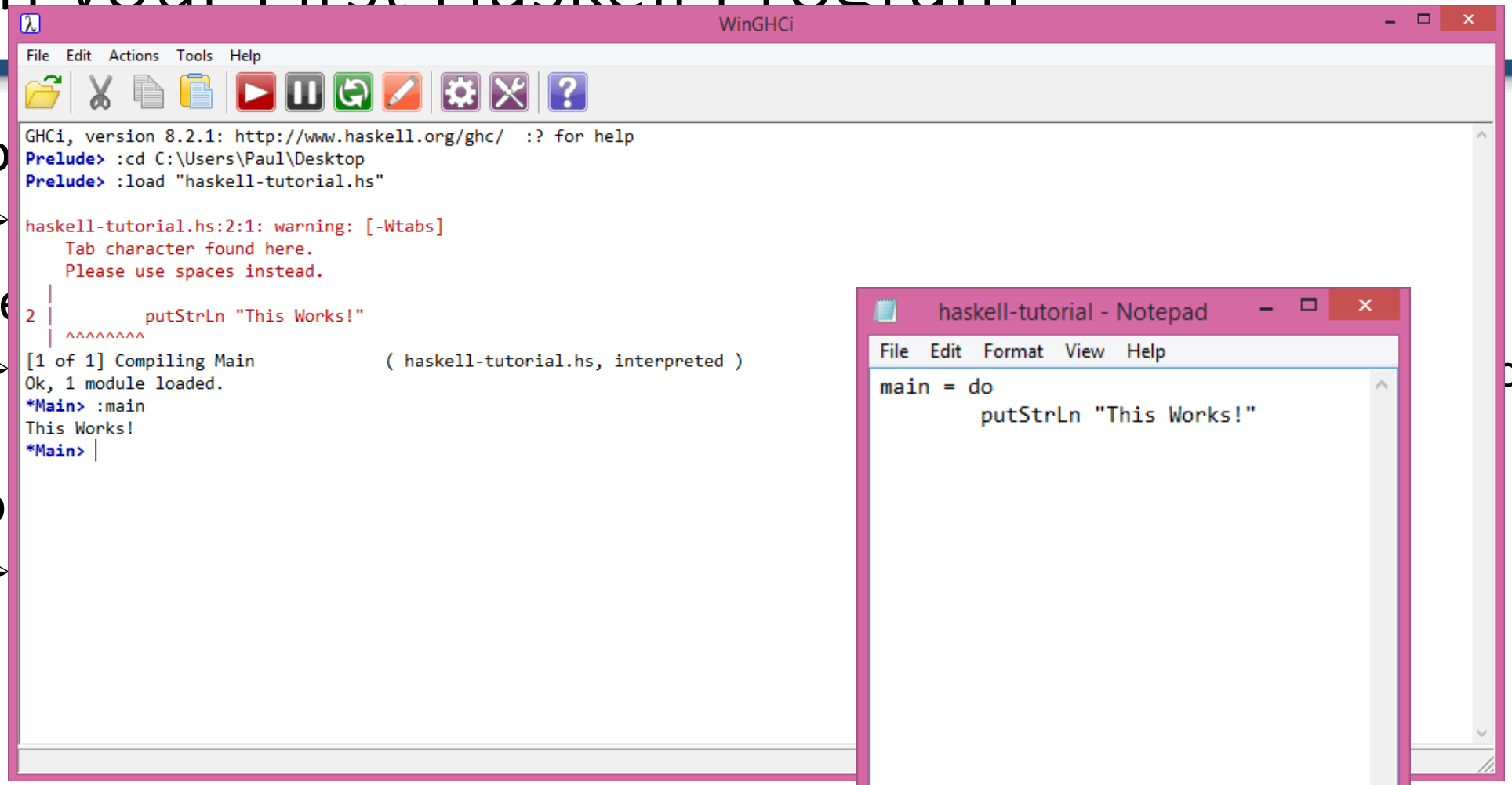
Lets try to understand basic features of Haskell with examples

Run your First Haskell Program

- Download and Install Haskell
  - Download link https://www.haskell.org/downloads
- File extension .hs
  - Open text editor, write your program, save your program with .hs extension (e.g., haskell-tutorail.hs)
- Compilation and Run
  - For Windows OS
    - Open WinGHCi from start menu
    - Load your program (File -> Load..)
    - Run the function you want

Run your First Haskell Program

- Do
  -
- File
  -
- Co
  -

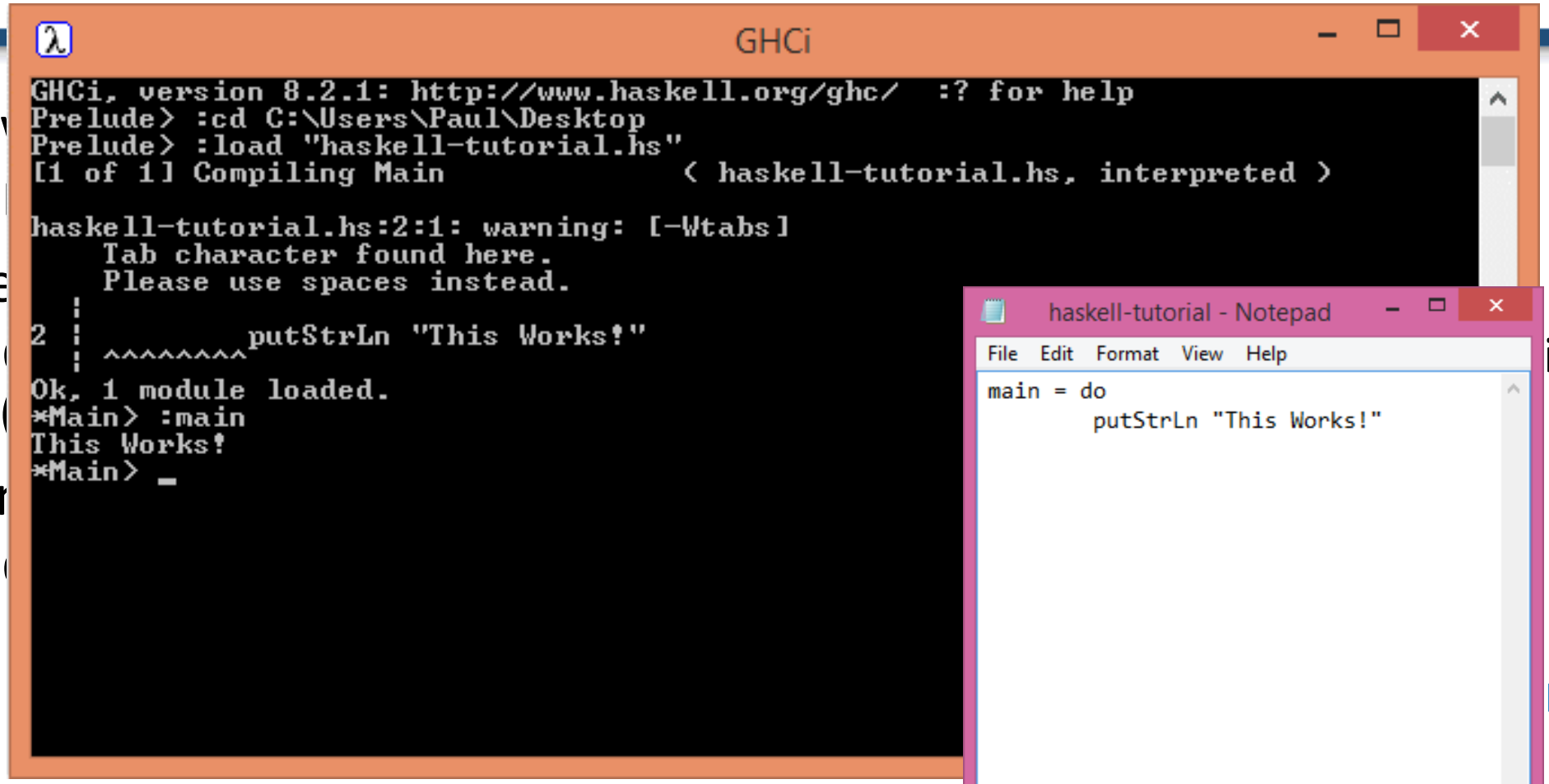


Run your First Haskell Program

- Download and Install Haskell
  - Download link https://www.haskell.org/downloads
- File extension .hs
  - Open text editor, write your program, save your program with .hs extension (e.g., haskell-tutorail.hs)
- Compilation and Run
  - Otherwise
    - Open GHCi
    - Enter into directory where you saved your program (:cd C:\Users\Paul\Desktop)
    - Load your program (:load "haskell-tutorial.hs")
    - Run the function you want

Run your First Haskell Program

- Do
  -
- File
  -
    (
- Cor
  -



ion

p)

- Run the function you want

Few things you may keep in mind

- Once you modify your program
  - Save it
  - Before running its function, recompile it - reload (*main> :r)
- Comment Line
  - --Comment
  - {-
    Multiple Comments
    -}
- Clear Screen
  - Ctrl+S

Date Types

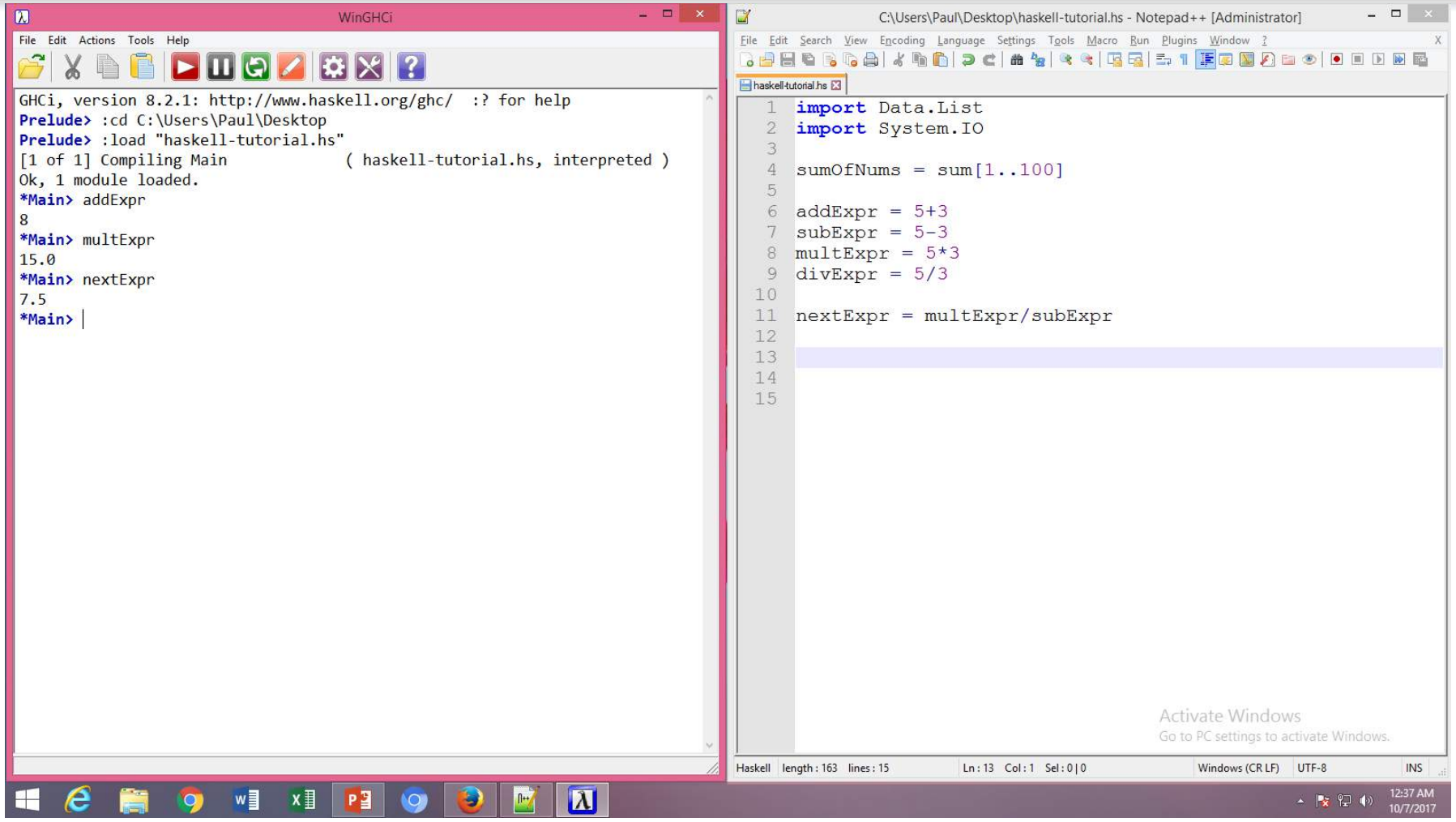
- Haskell uses type inference
  - Range of ‘Int’: -2^63 to 2^63
  - Range of ‘Integer’: Unbound -- as per the capability of memory of the system
  - Other data types: Float,Double, Bool, Char, Tuple -- will be discussing with example
  - permanent3 :: Int

    permanent3 = 3

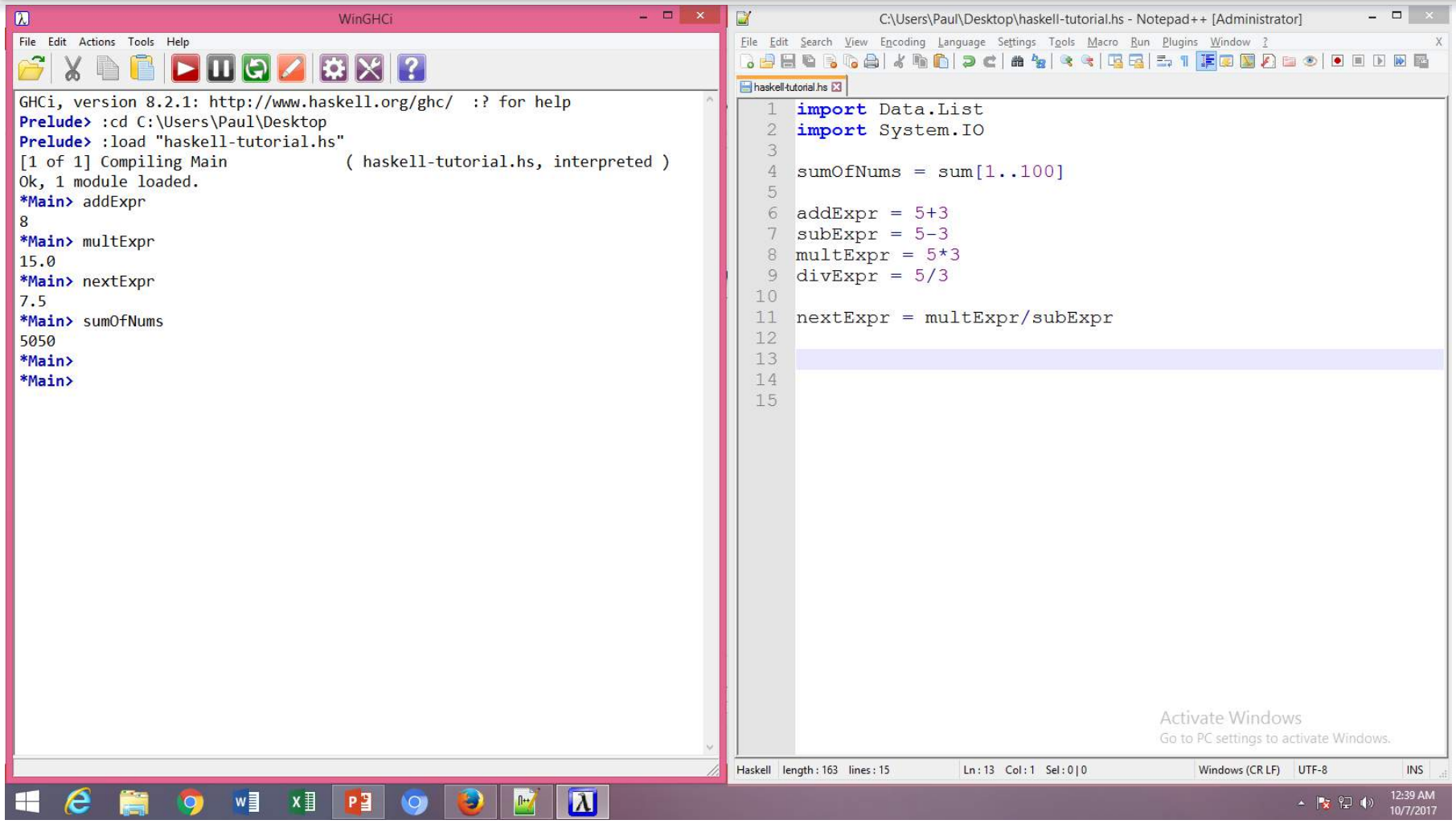
    } permanent value for a variable

    --Never Change

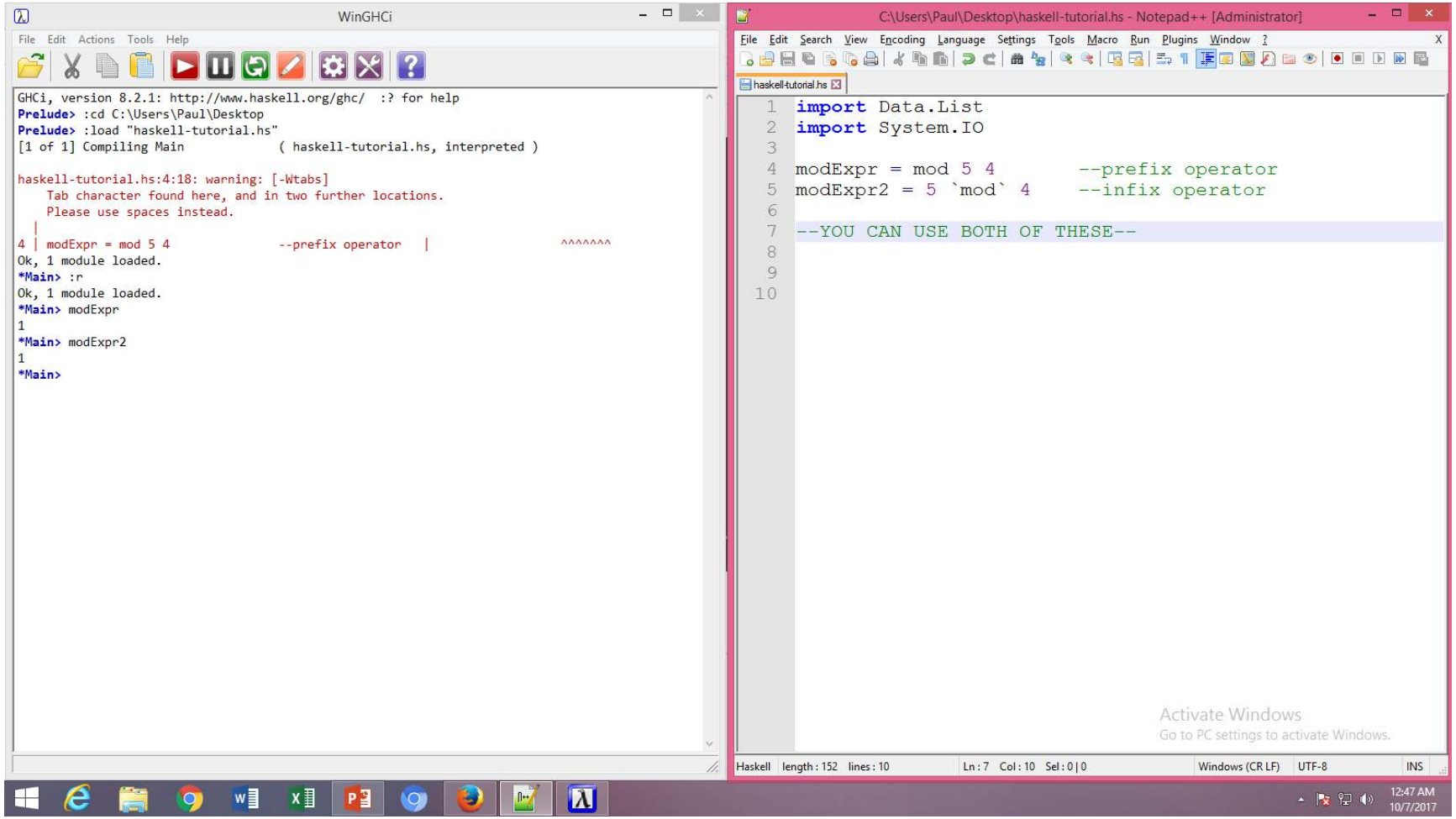
Expressions



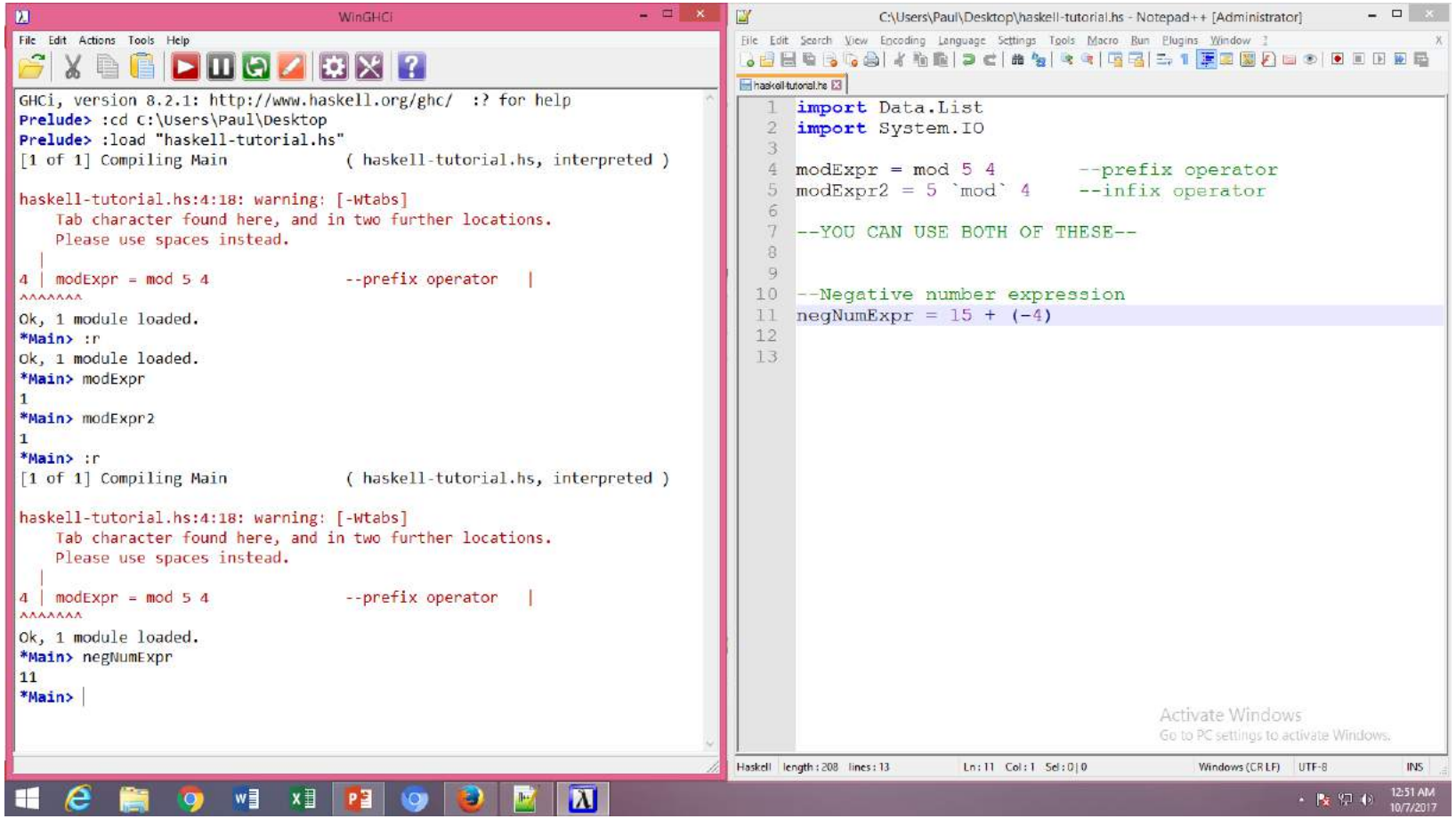
Expressions



Infix and Prefix Operator



Negative Number Expression



Other built-in Math Function

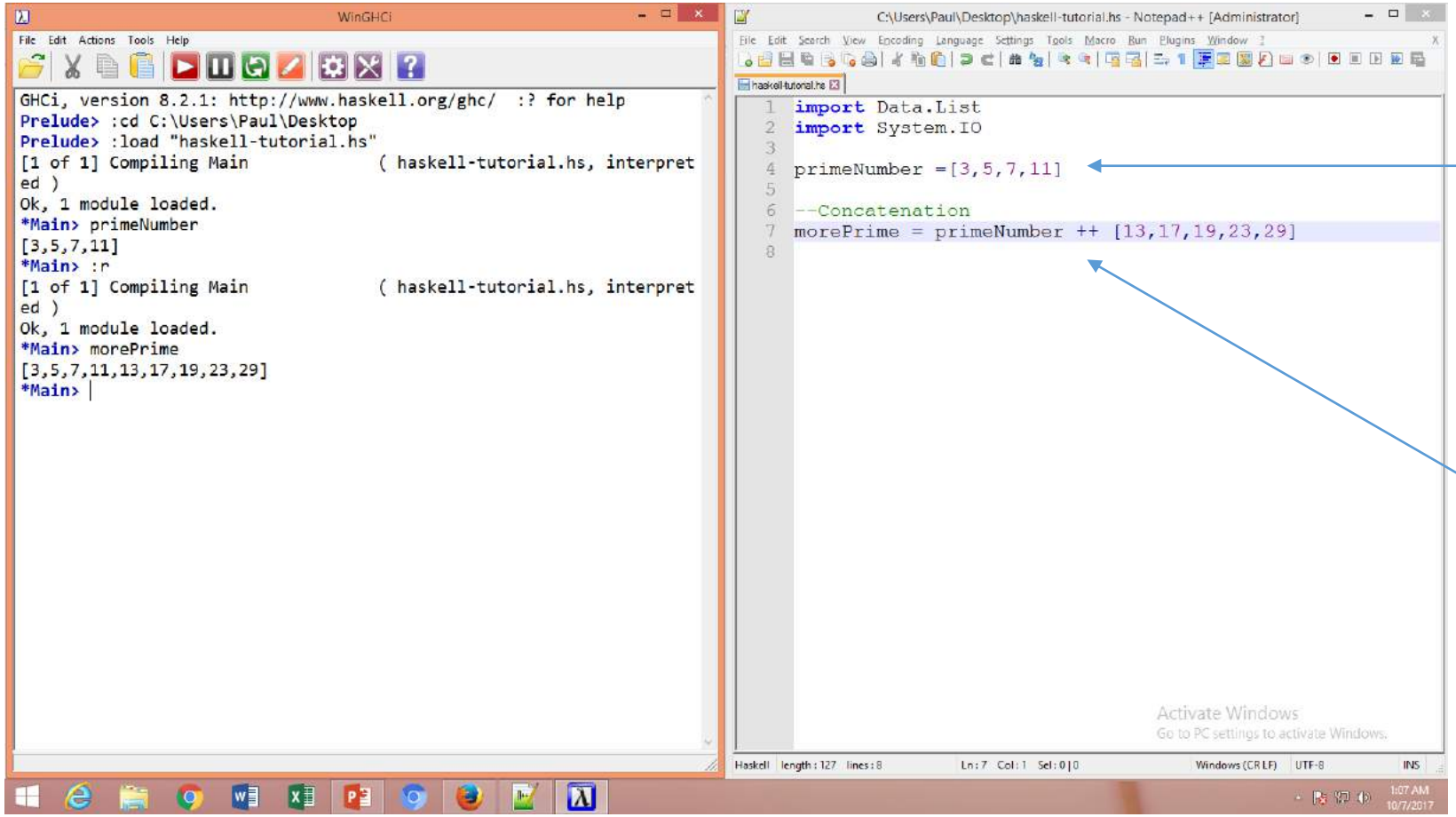
- piVal = pi
- ePow9 = exp 9
- logOf9 = log 9
- Squared9 = 9 ** 2
- truncateVal = truncate 9.999
- roundVal = round 9.999
- ceilingVal = ceiling 9.999
- floorVal = floor 9.999
- Also
  - sin, cos, tan, asign, acos, atan, signh, cosh, tanh, asignh, acosh, atanh

Other built-in Math Function

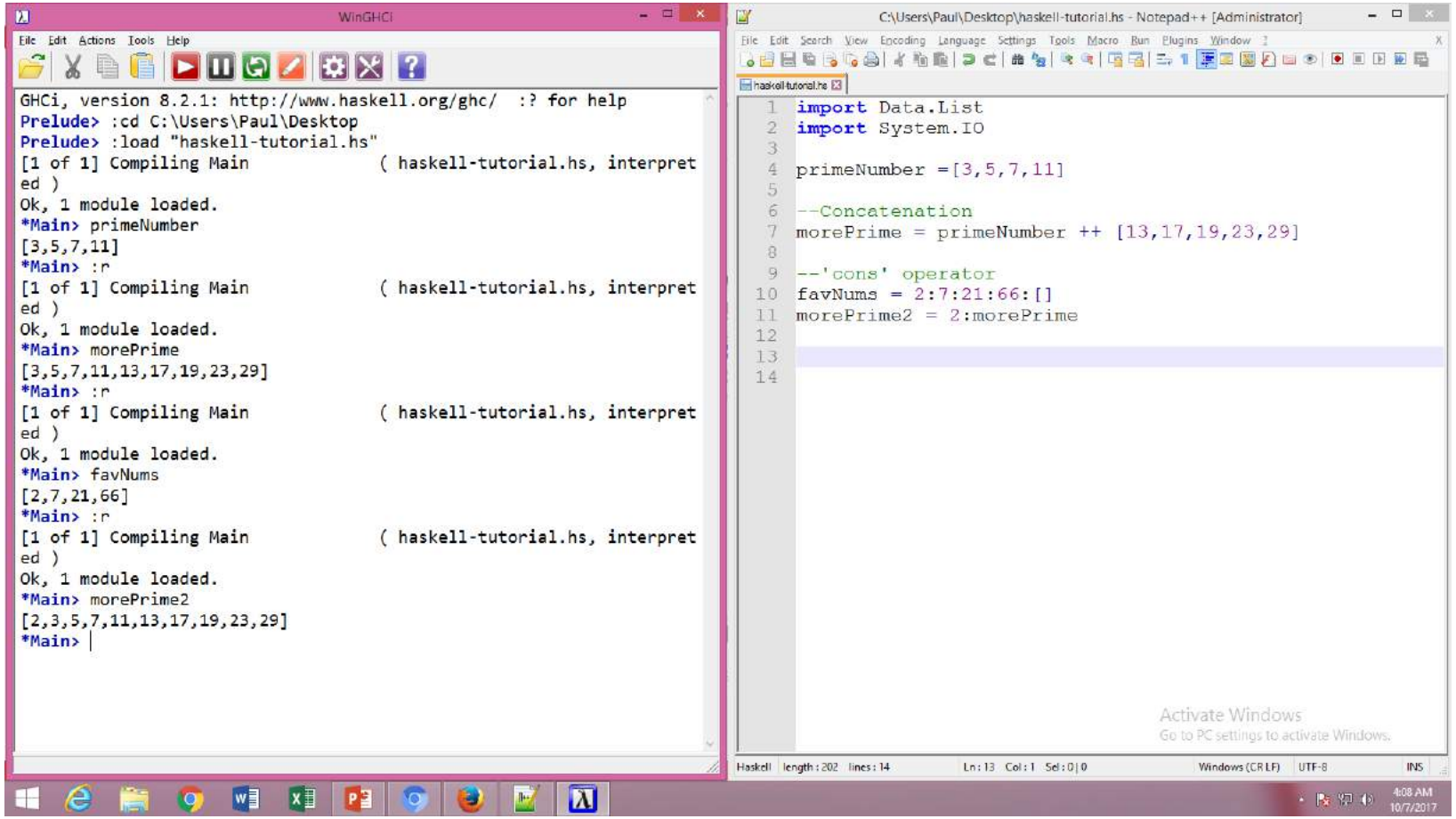
- piVal = pi
- ePow9 = exp 9
- logOf9 = log 9
- Squared9 = 9 ** 2
- truncateVal = truncate 9.999
- roundVal = round 9.999
- ceilingVal = ceiling 9.999
- floorVal = floor 9.999
- Also
  - sin, cos, tan, asign, acos, atan, signh, cosh, tanh, asignh, acosh, atanh

EXPLORE THESE

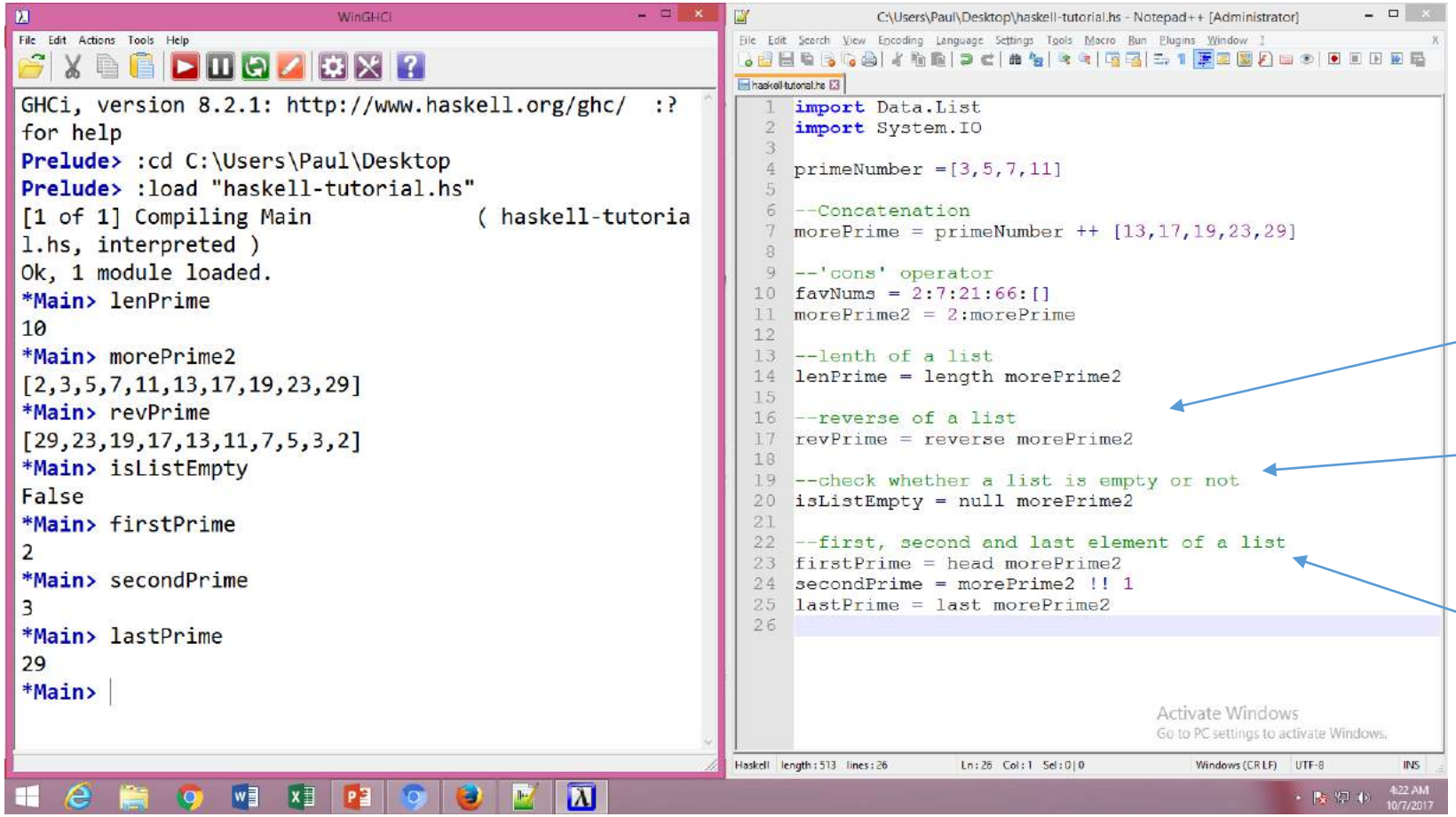
List - Concatenation



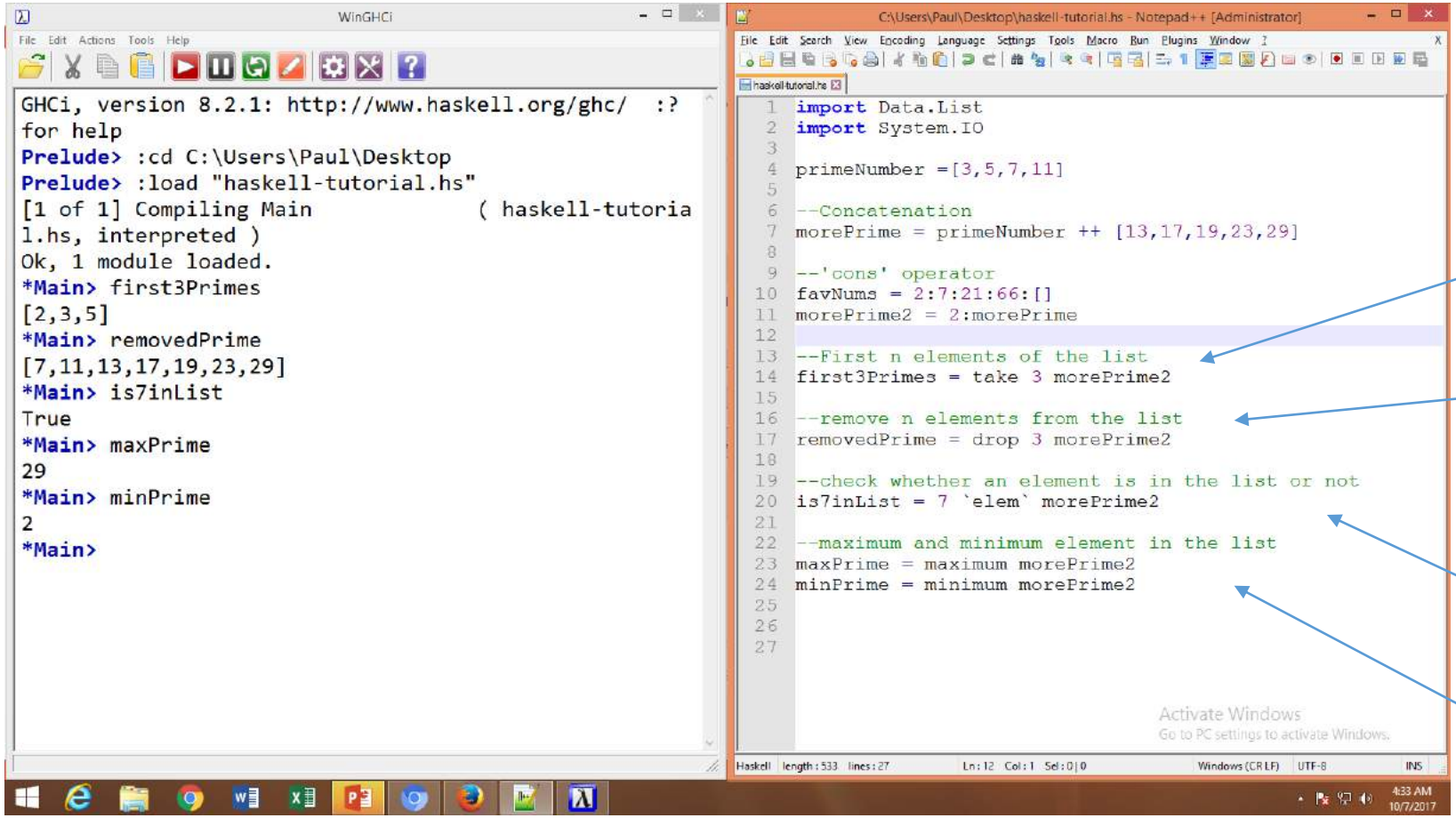
List – ‘cons’ operator



More Operations on List



More Operations on List



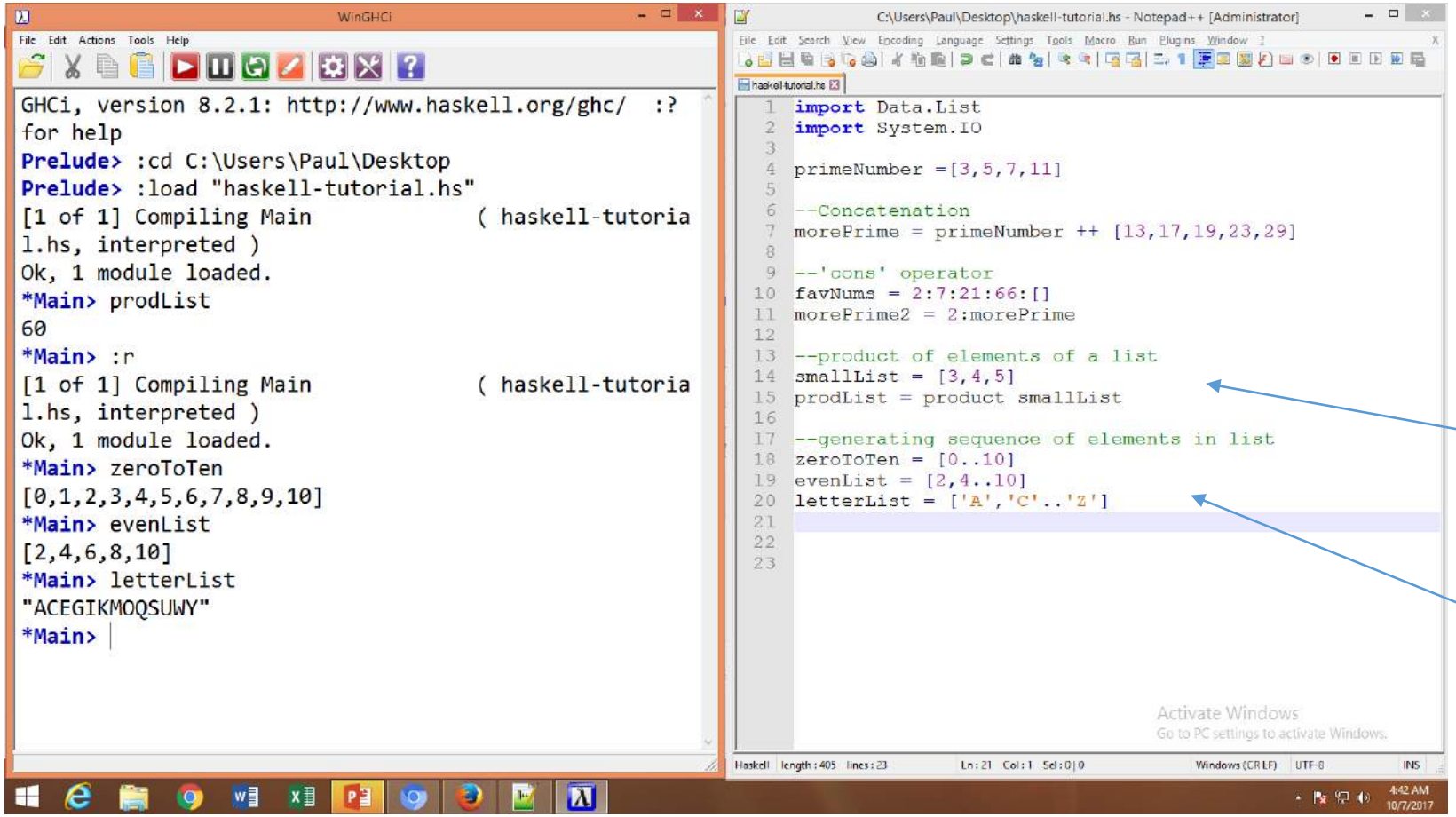
first n element

removing first n element

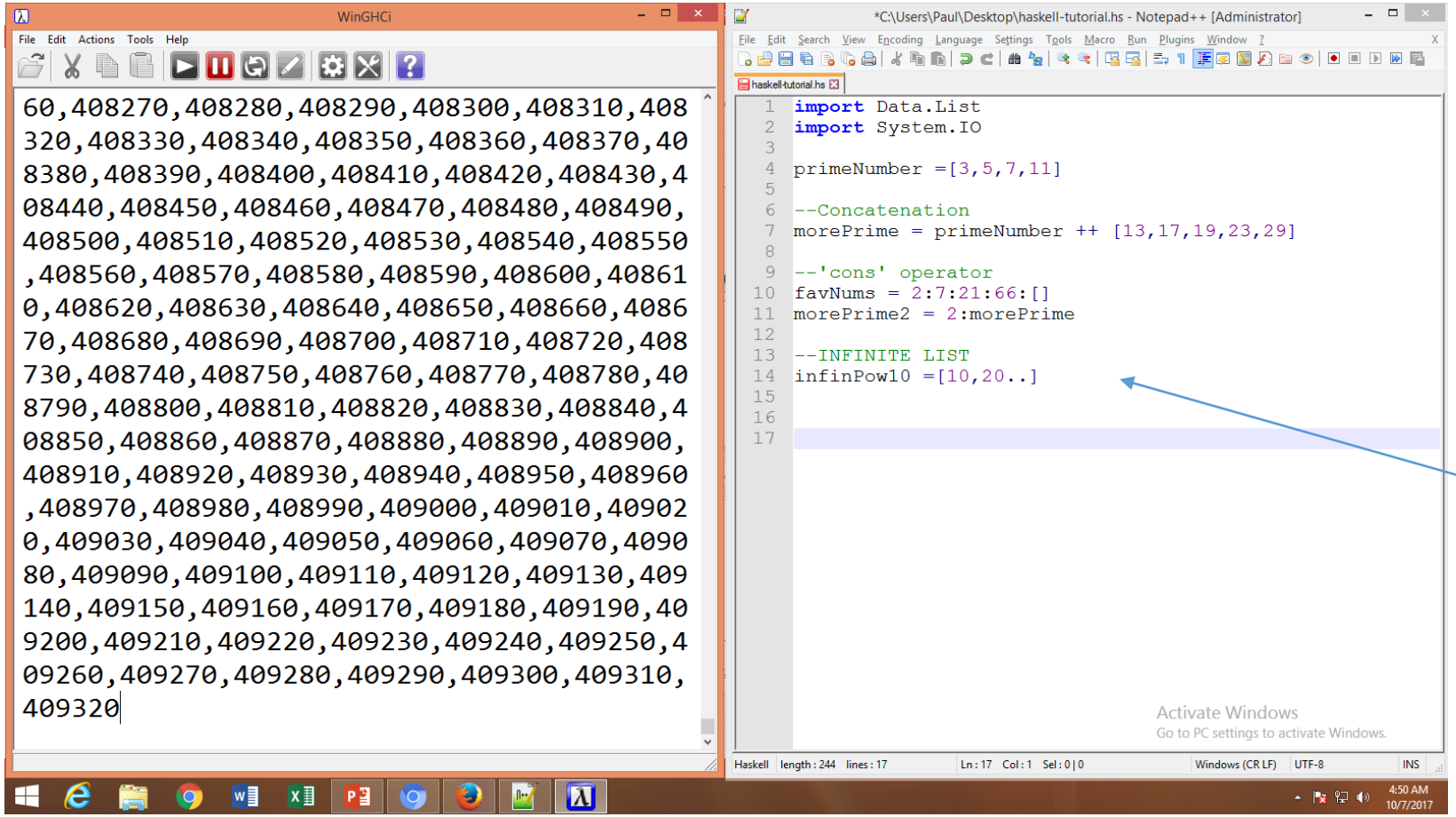
finding an element

max and min

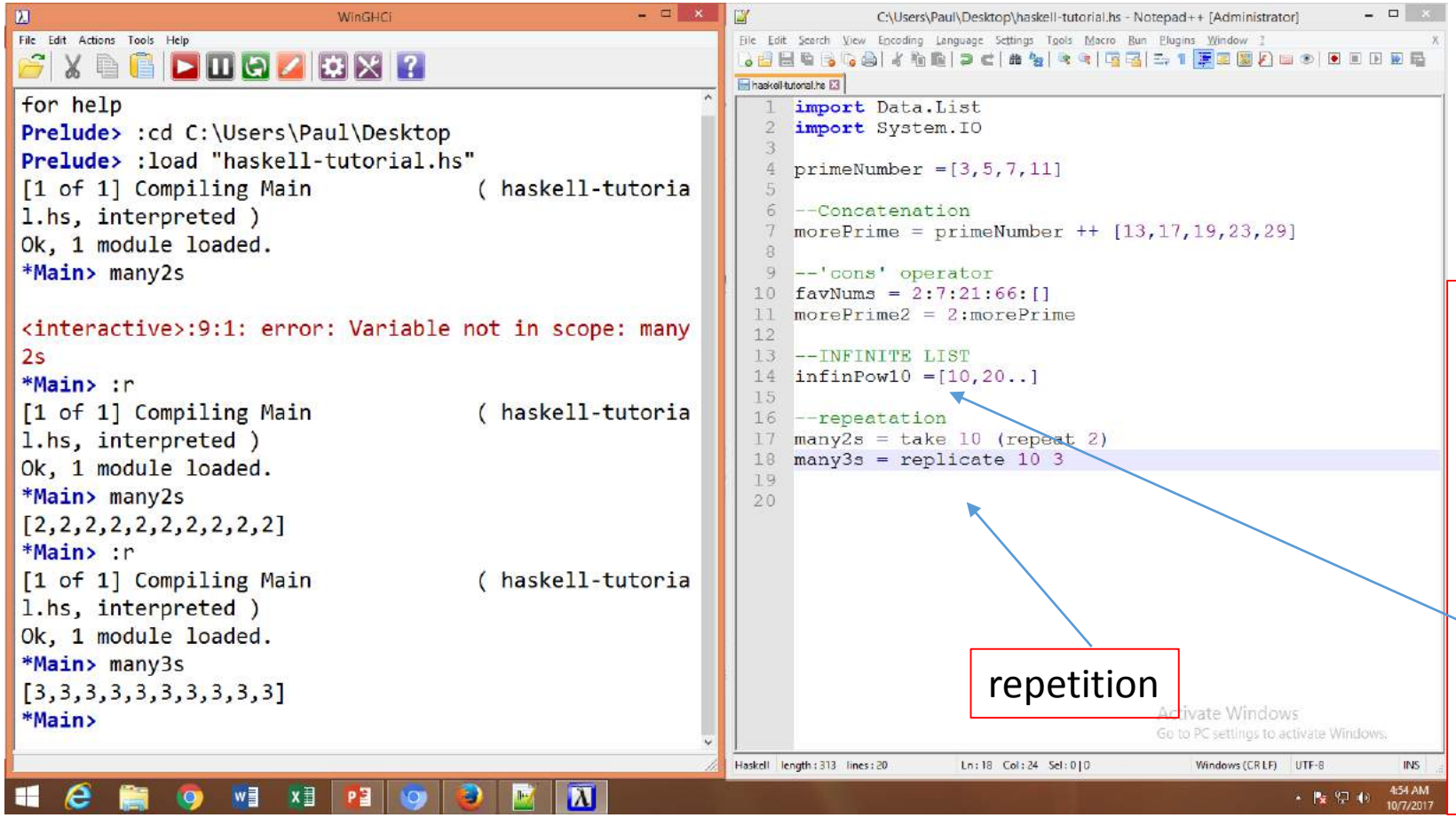
More Operations on List



More Operations on List

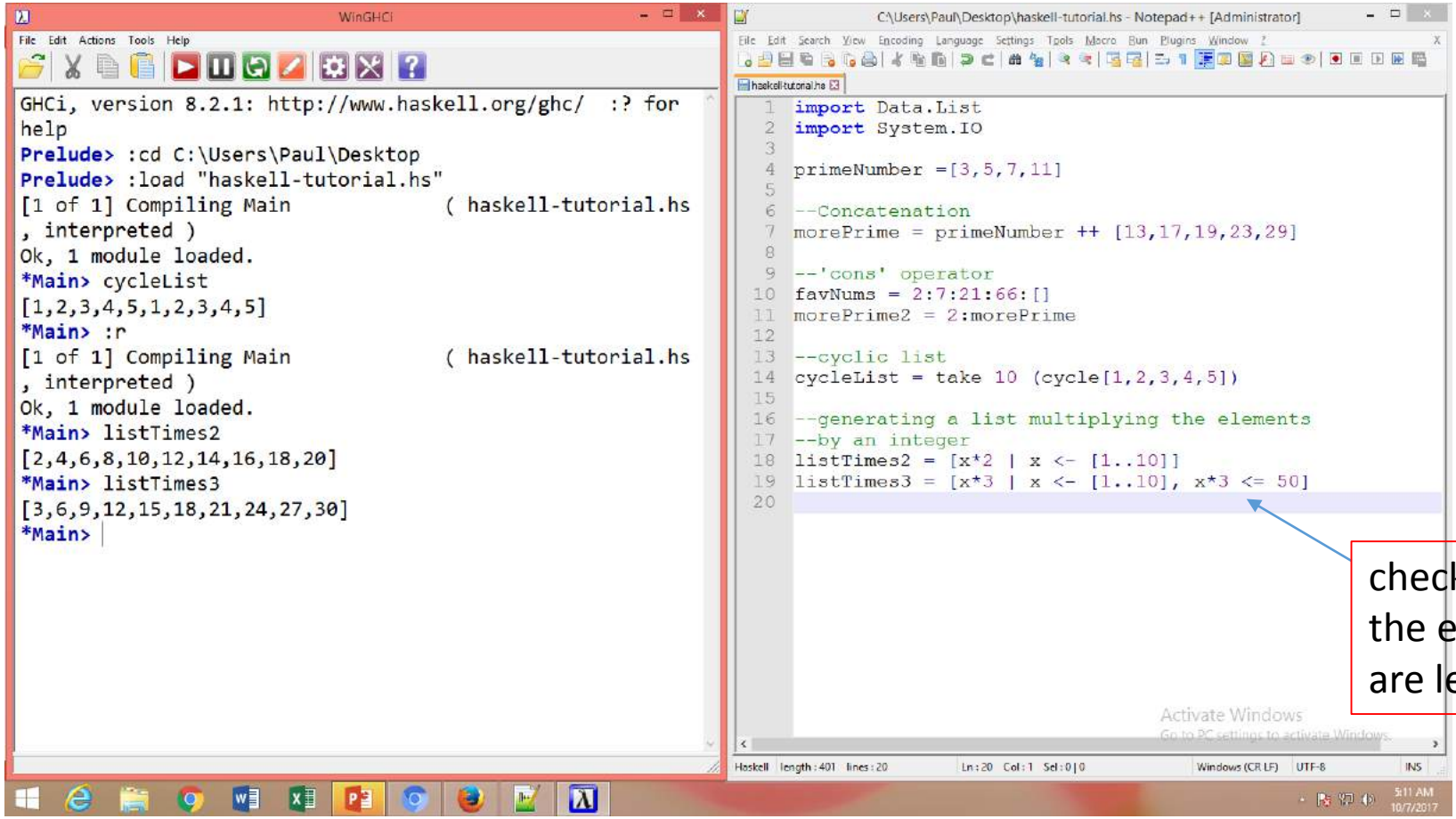


More Operations on List



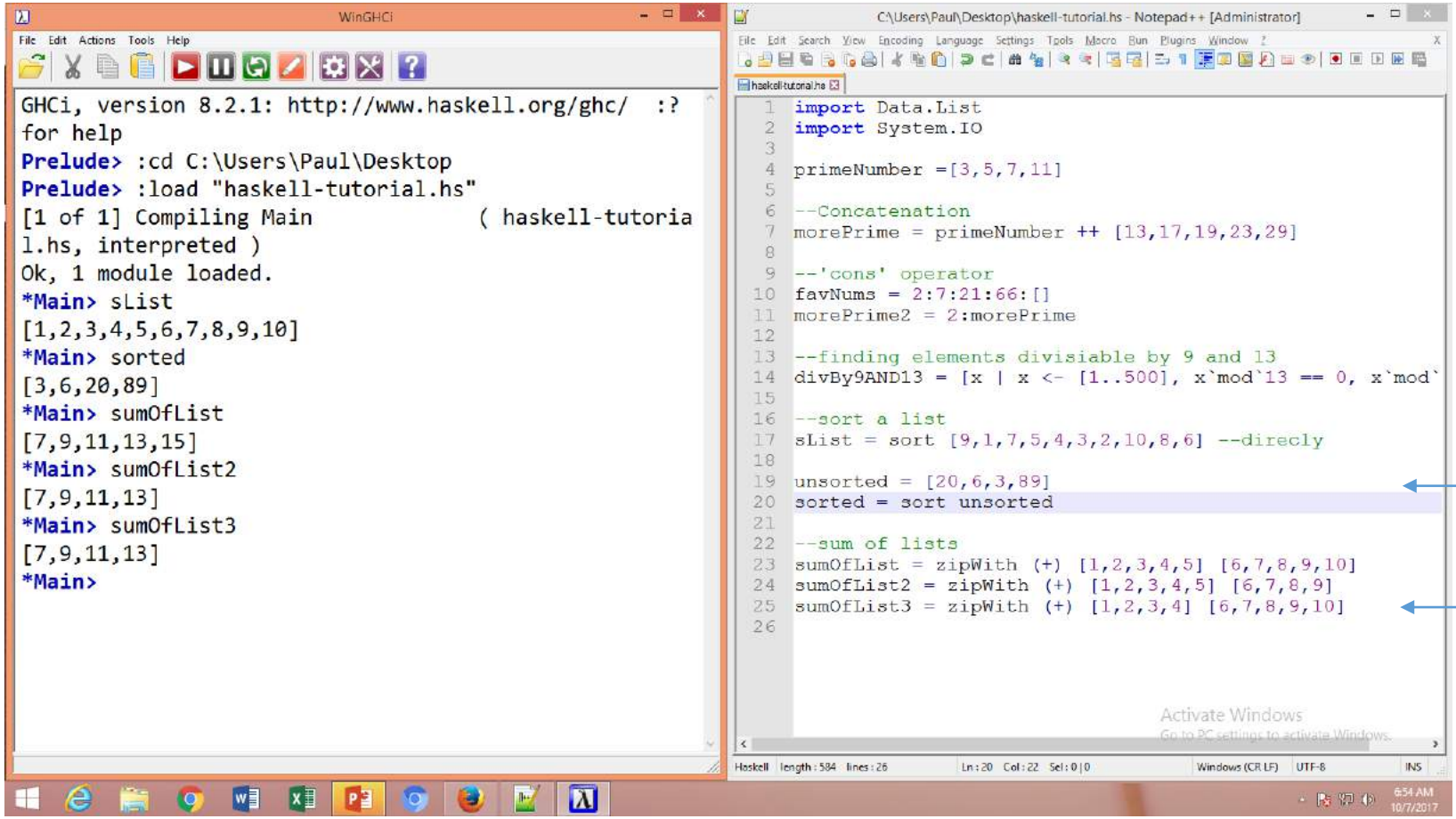
One of the examples of advantages of laziness property and functional approach: here, the presence of ***infinite list*** does not affect other expressions/ functions in the program

More Operations on List

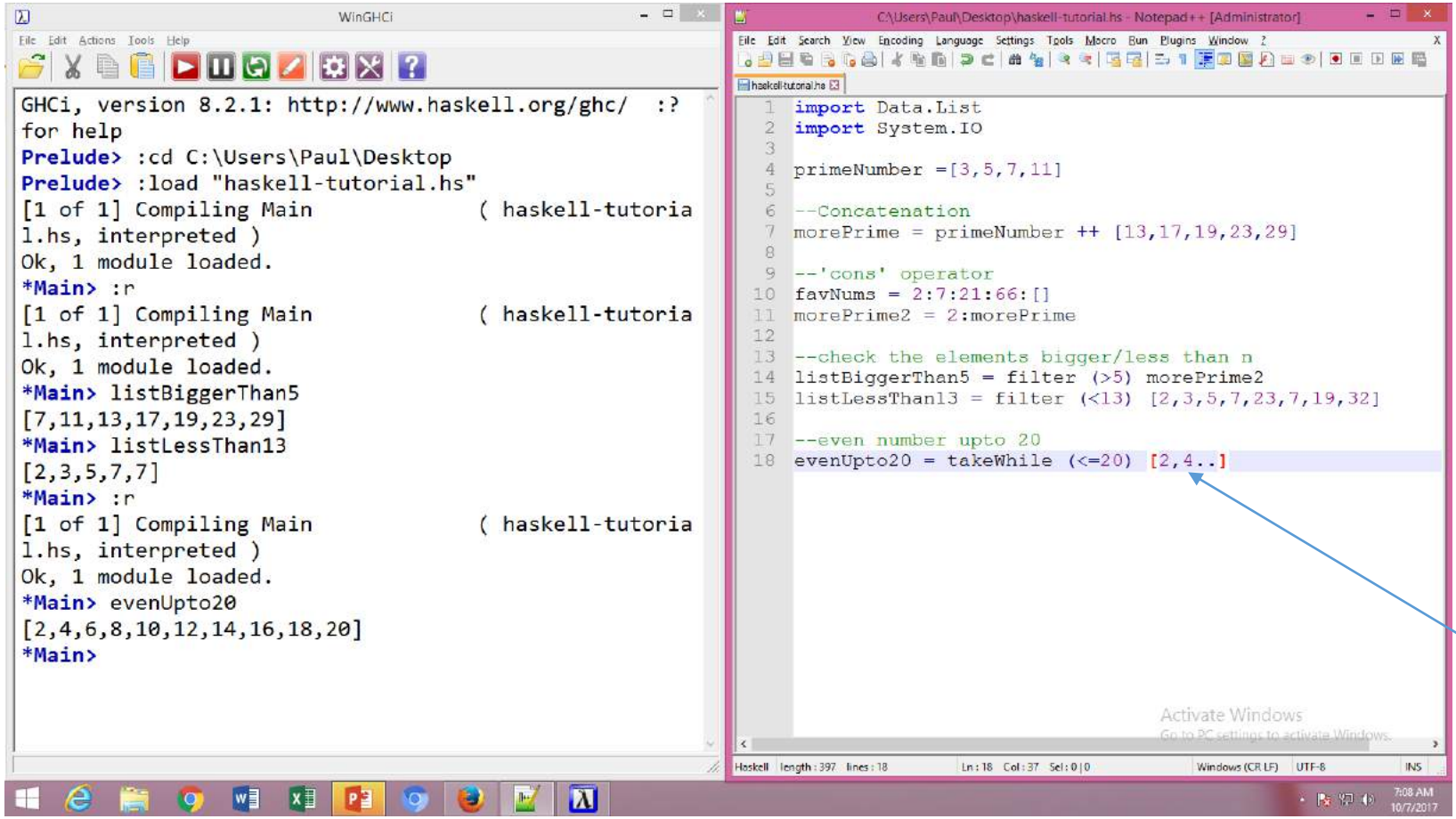


check whether
the element generated
are less than 50 or not

More Operations on List

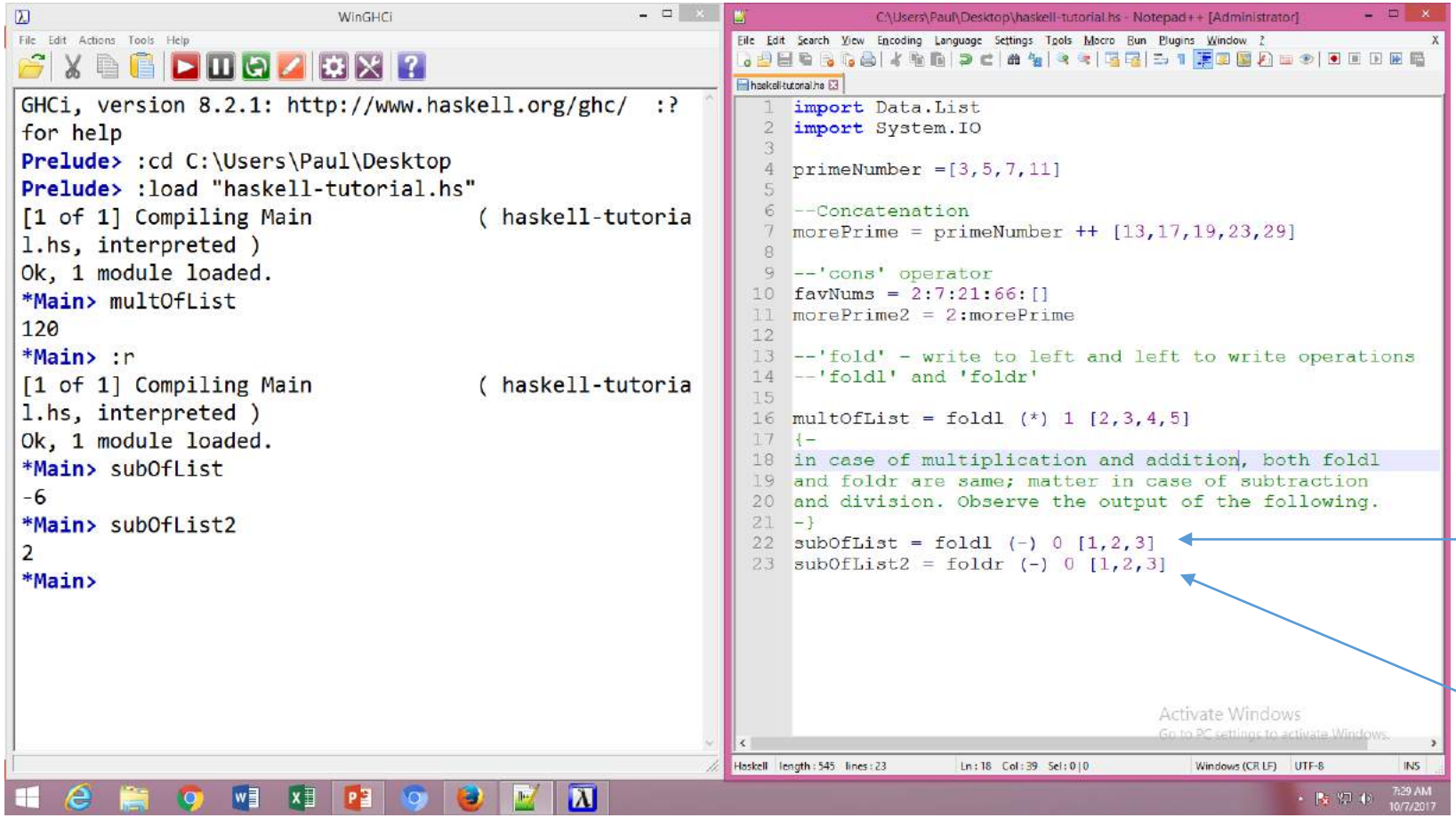


More Operations on List



another example of laziness; although infinite list, check up to 20

More Operations on List

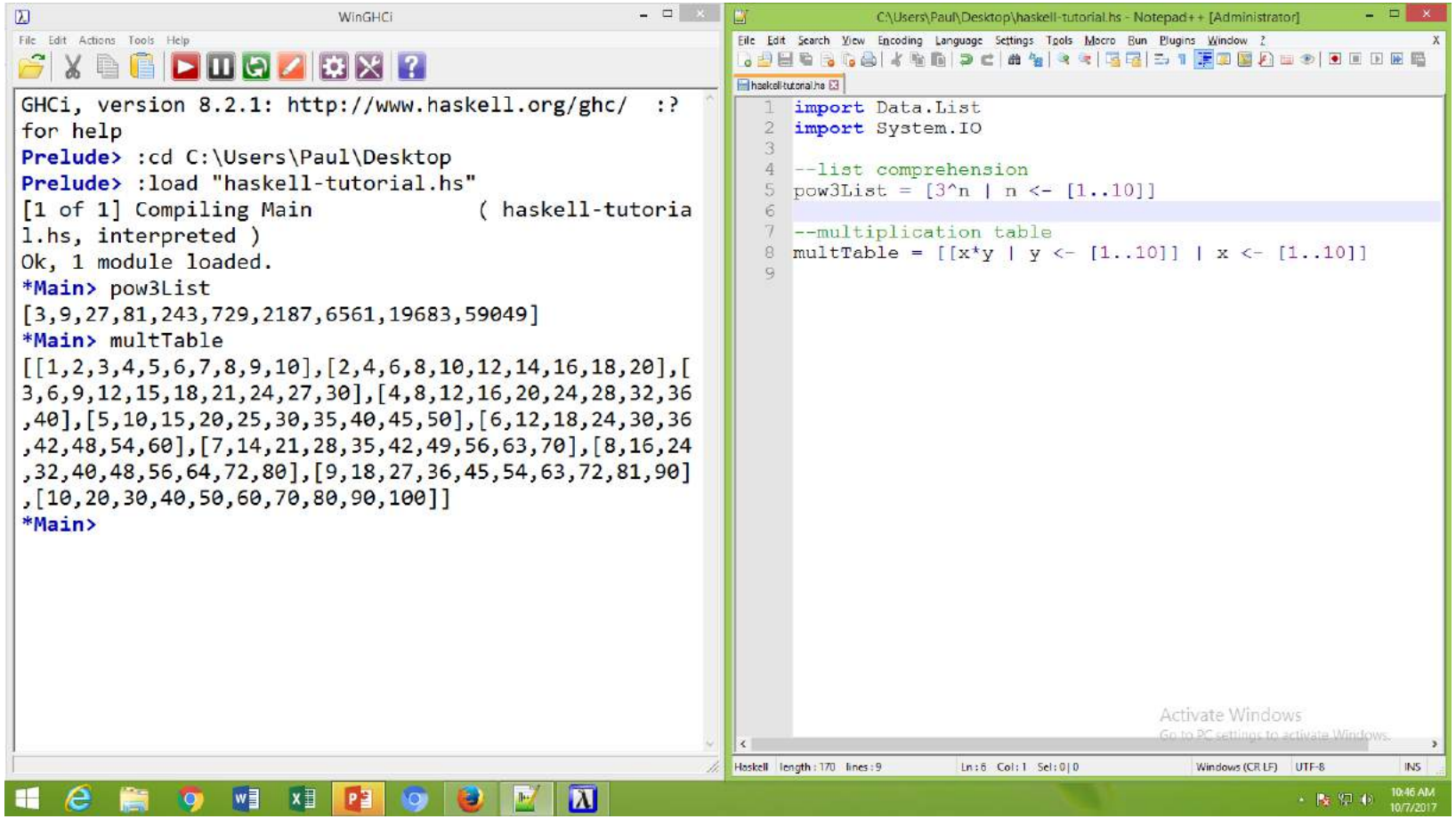


foldl & foldr

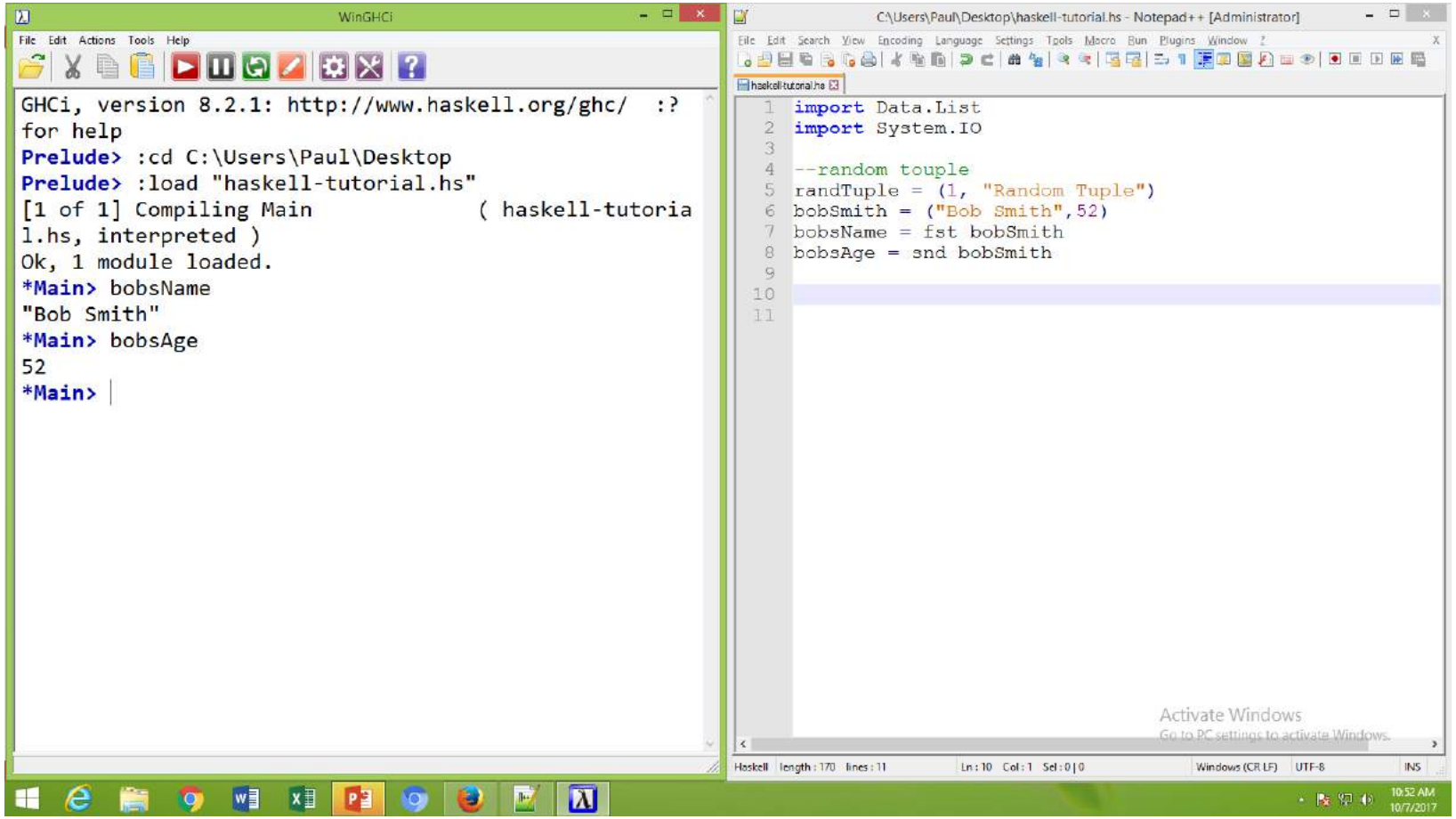
(((0-1)-2)-3)

1-(2-(3-0))

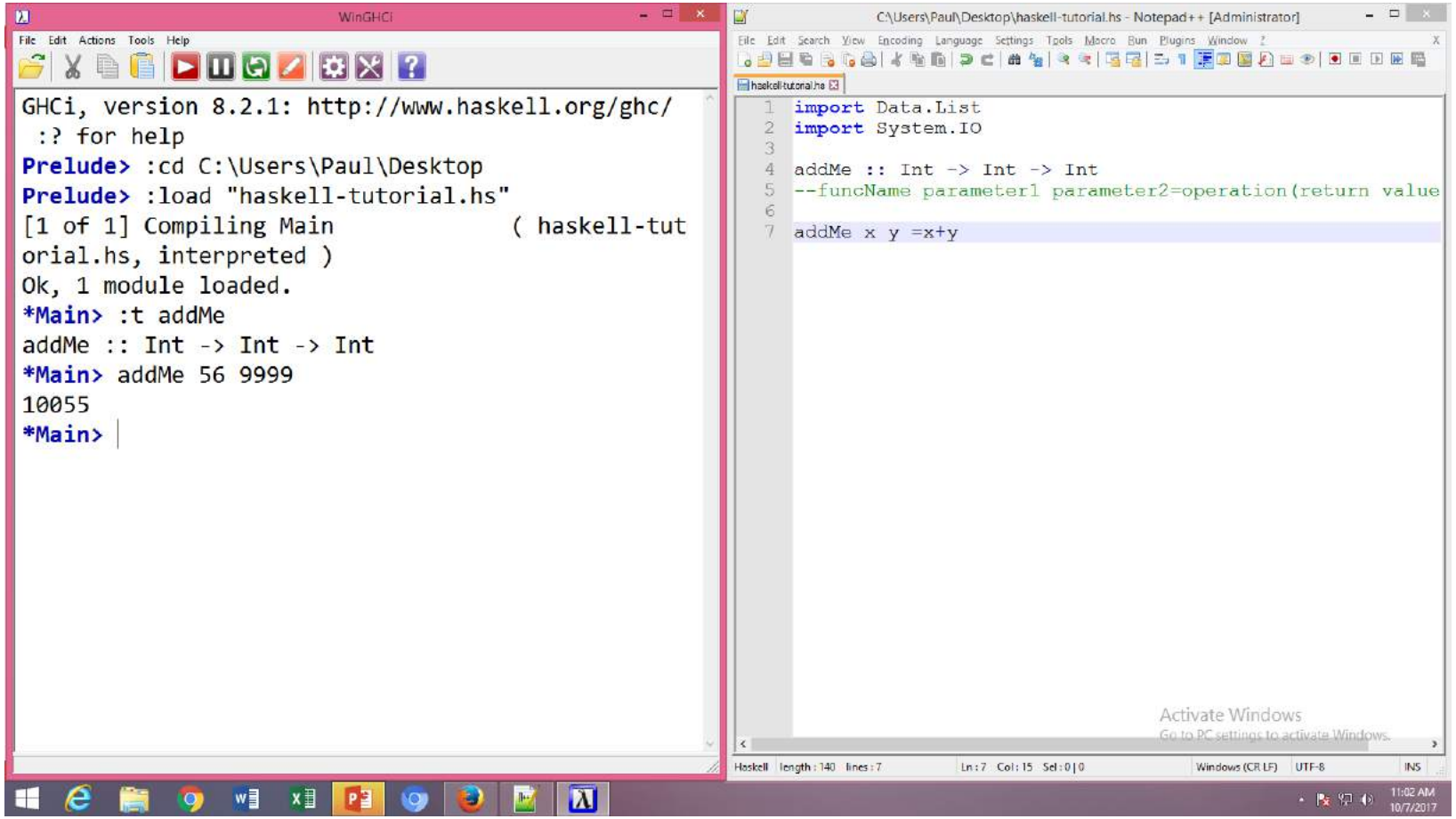
More Operations on List



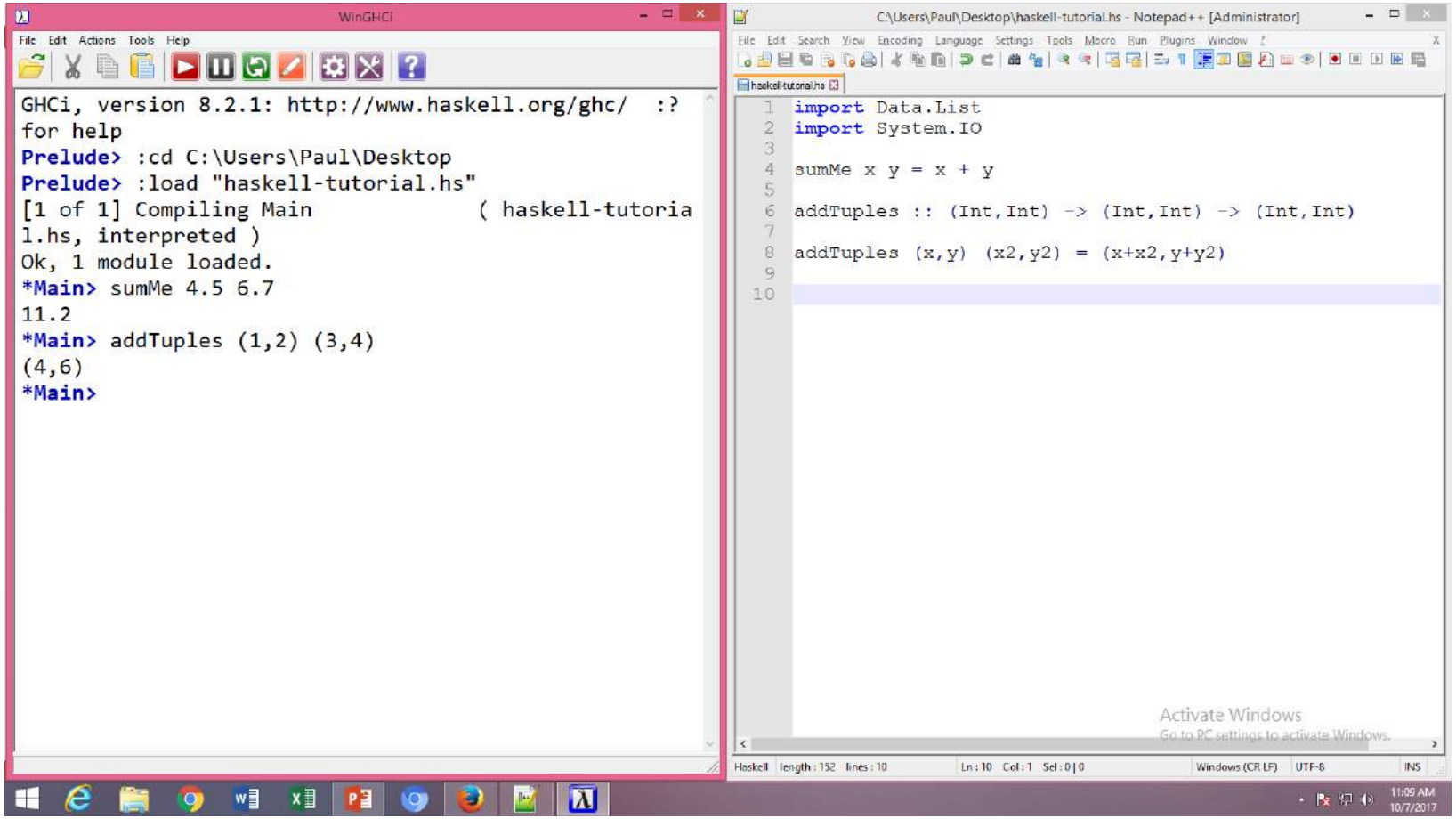
Multiple Data Type



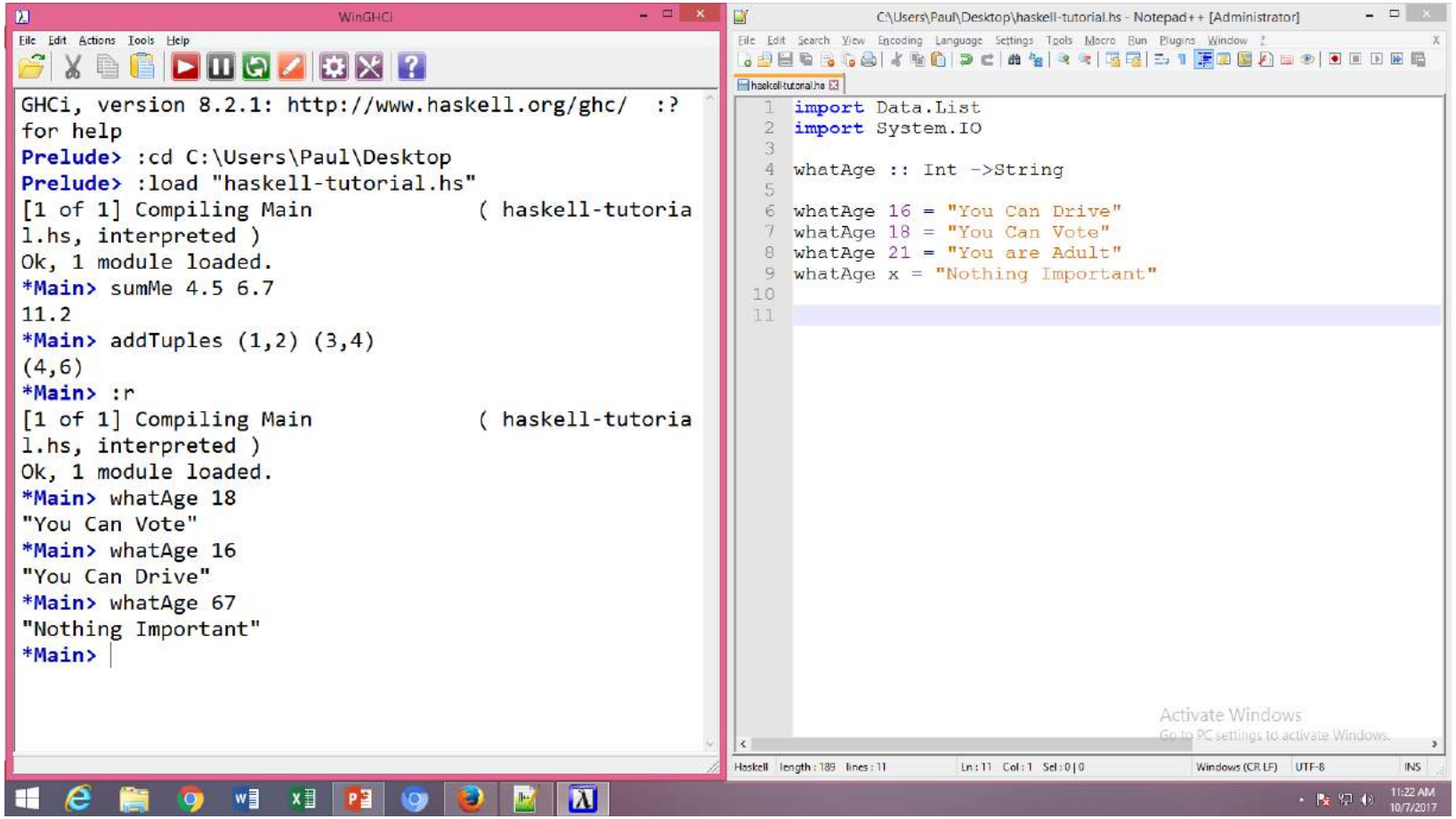
Function Declaration



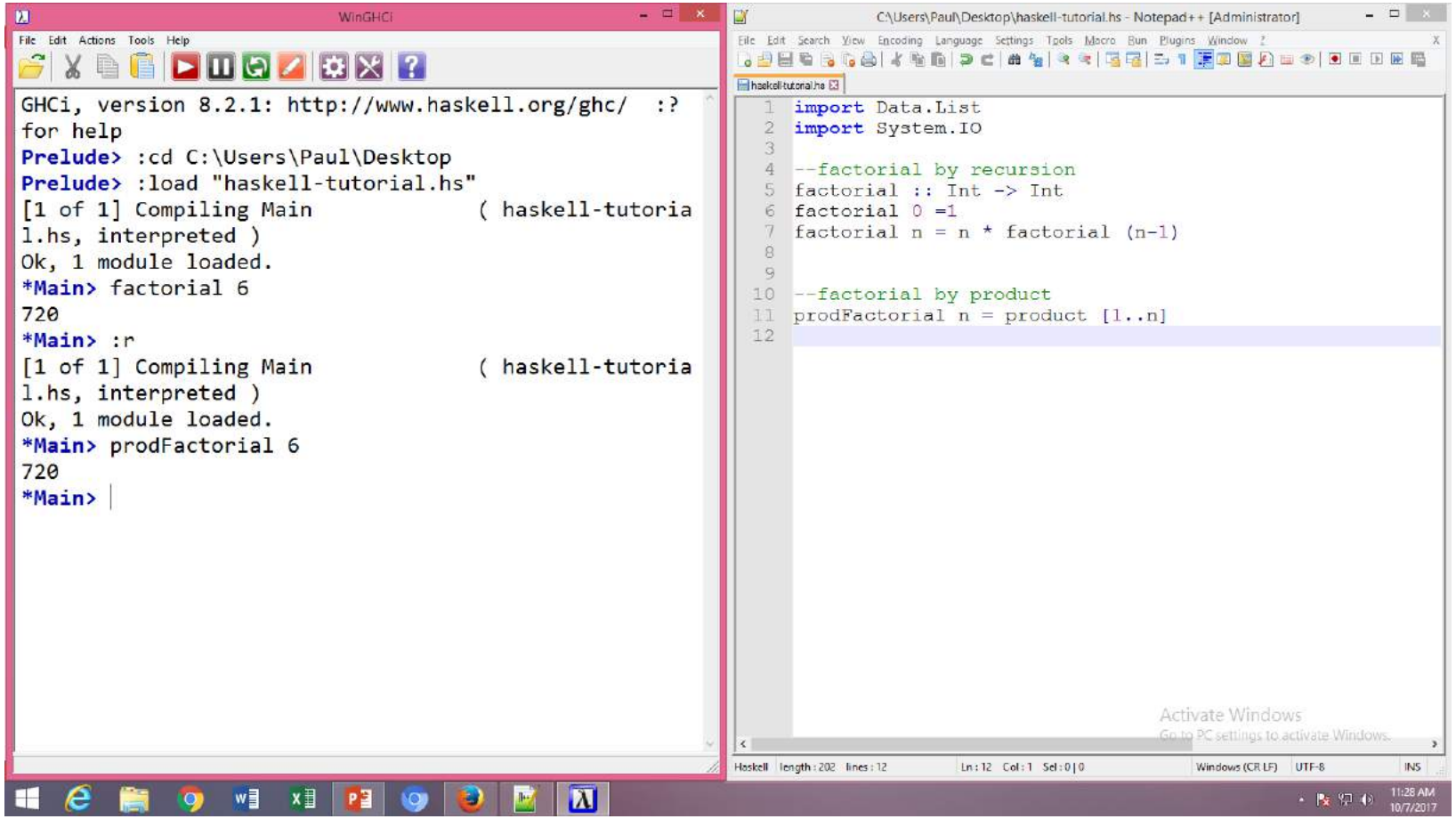
User Type Declaration



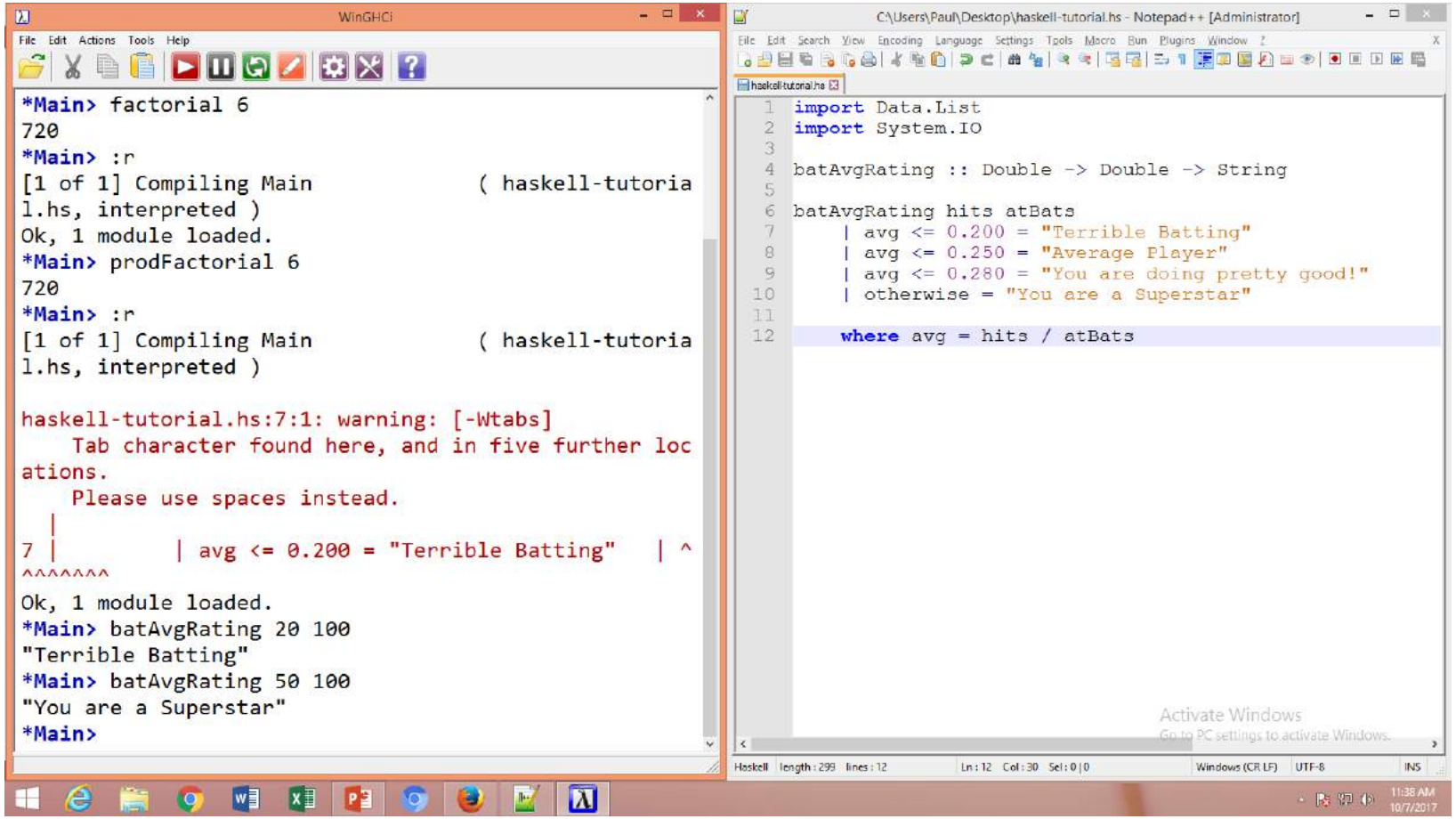
User Type Declaration



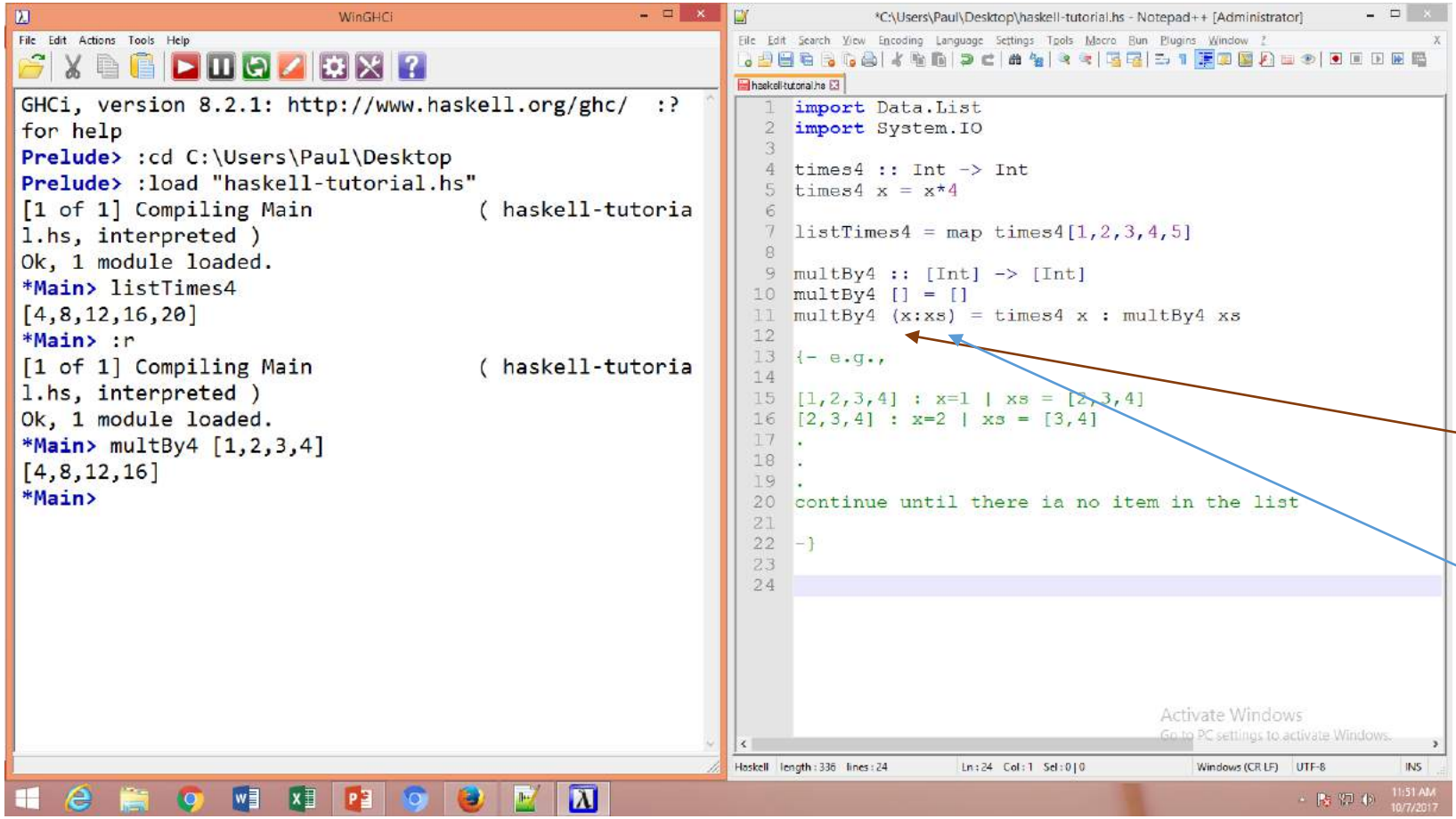
Factorial (by recursion and by product)



Guard (where clause)

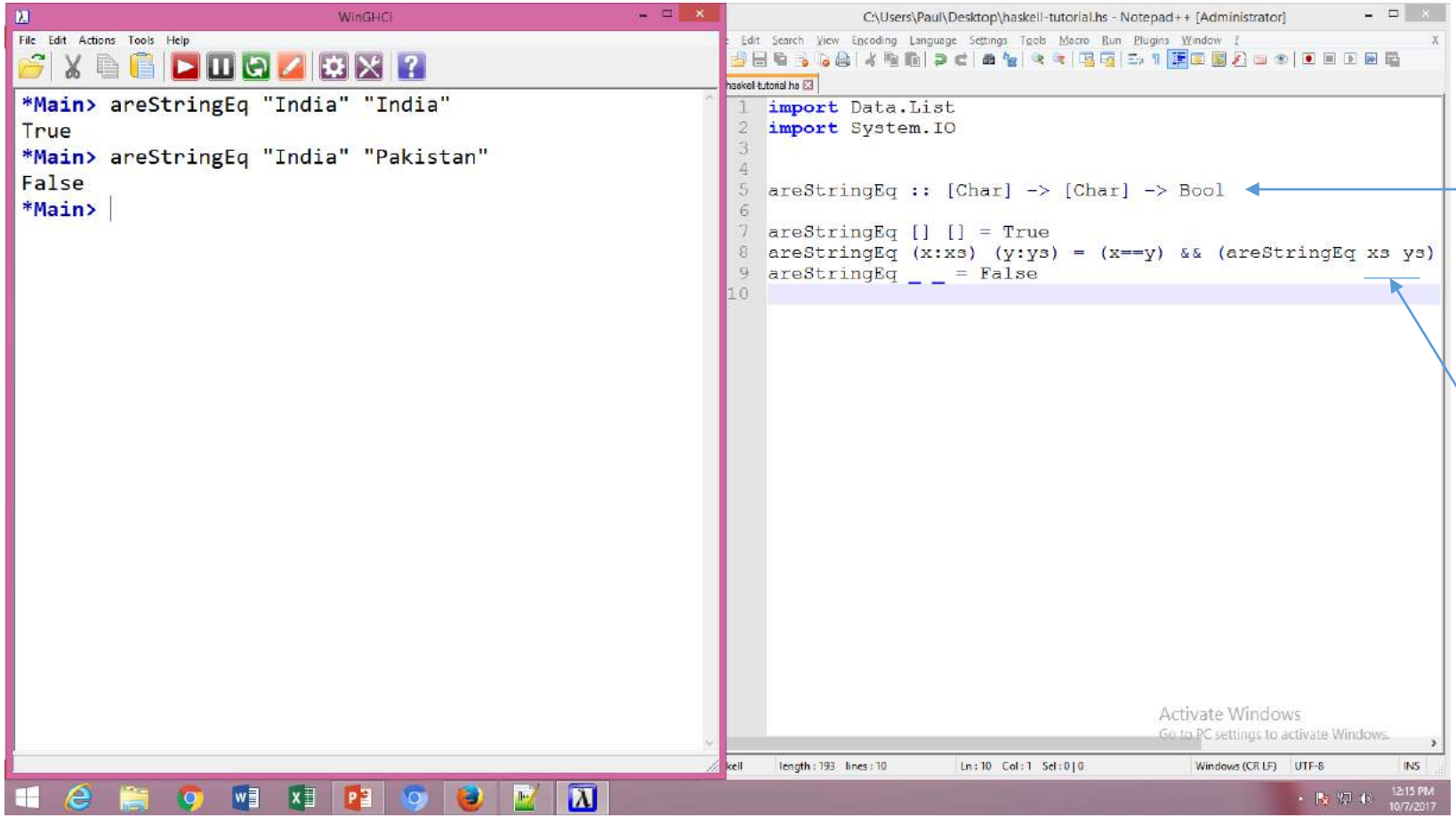


Higher Order Functions



you don't know how many items in the list Beforehand; x represents first element in the list, and xs represents remaining elements of the list

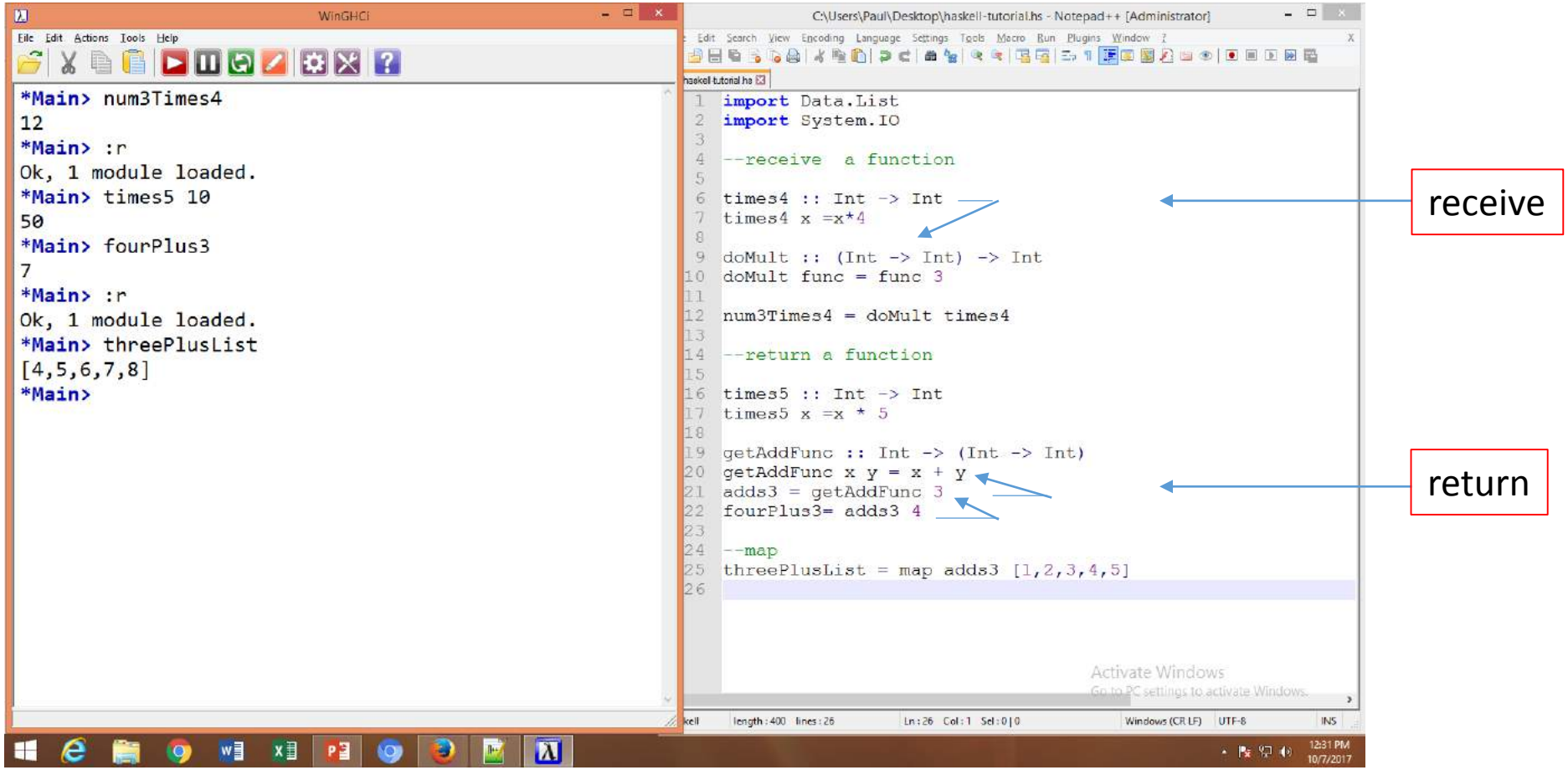
Higher Order Functions



return type

remaining

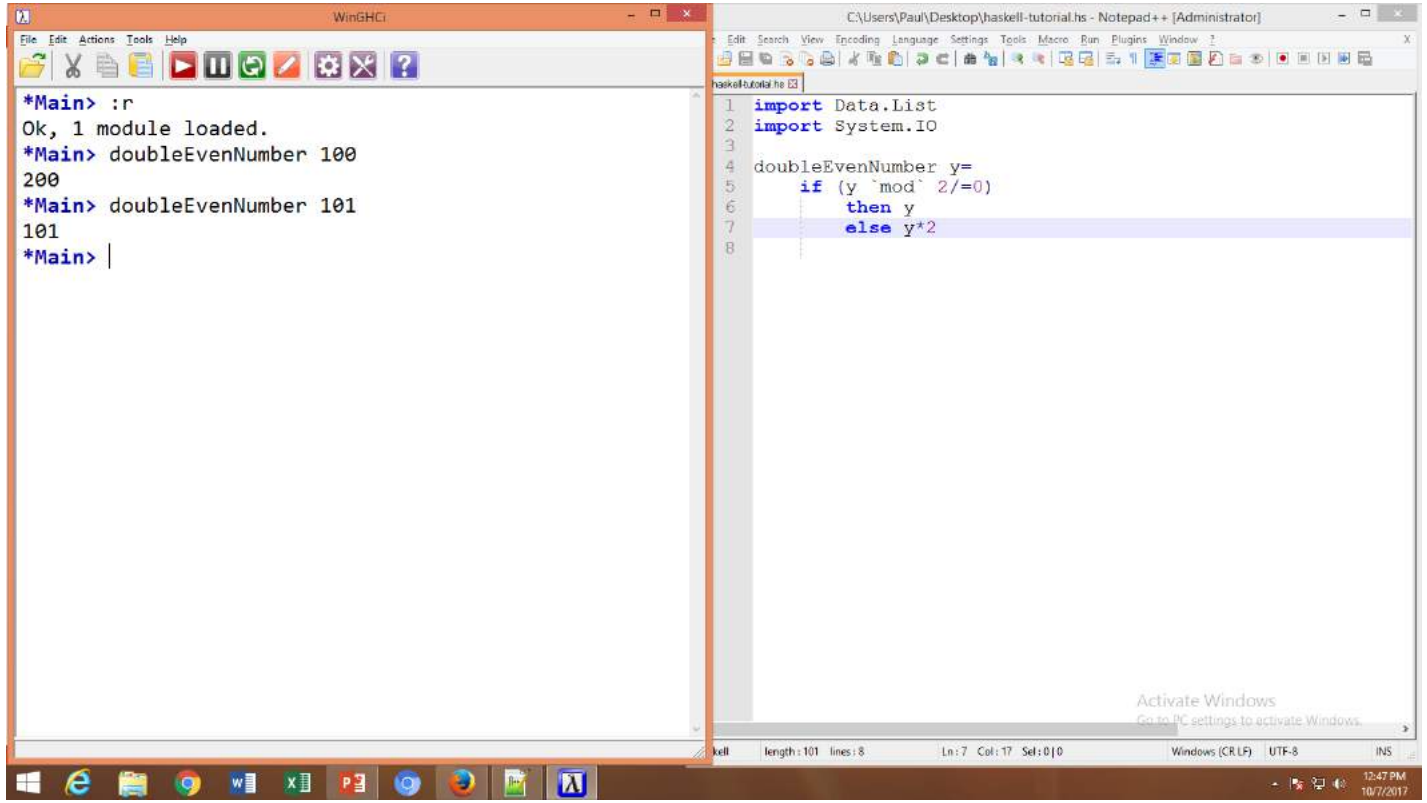
Receive and Return a Function



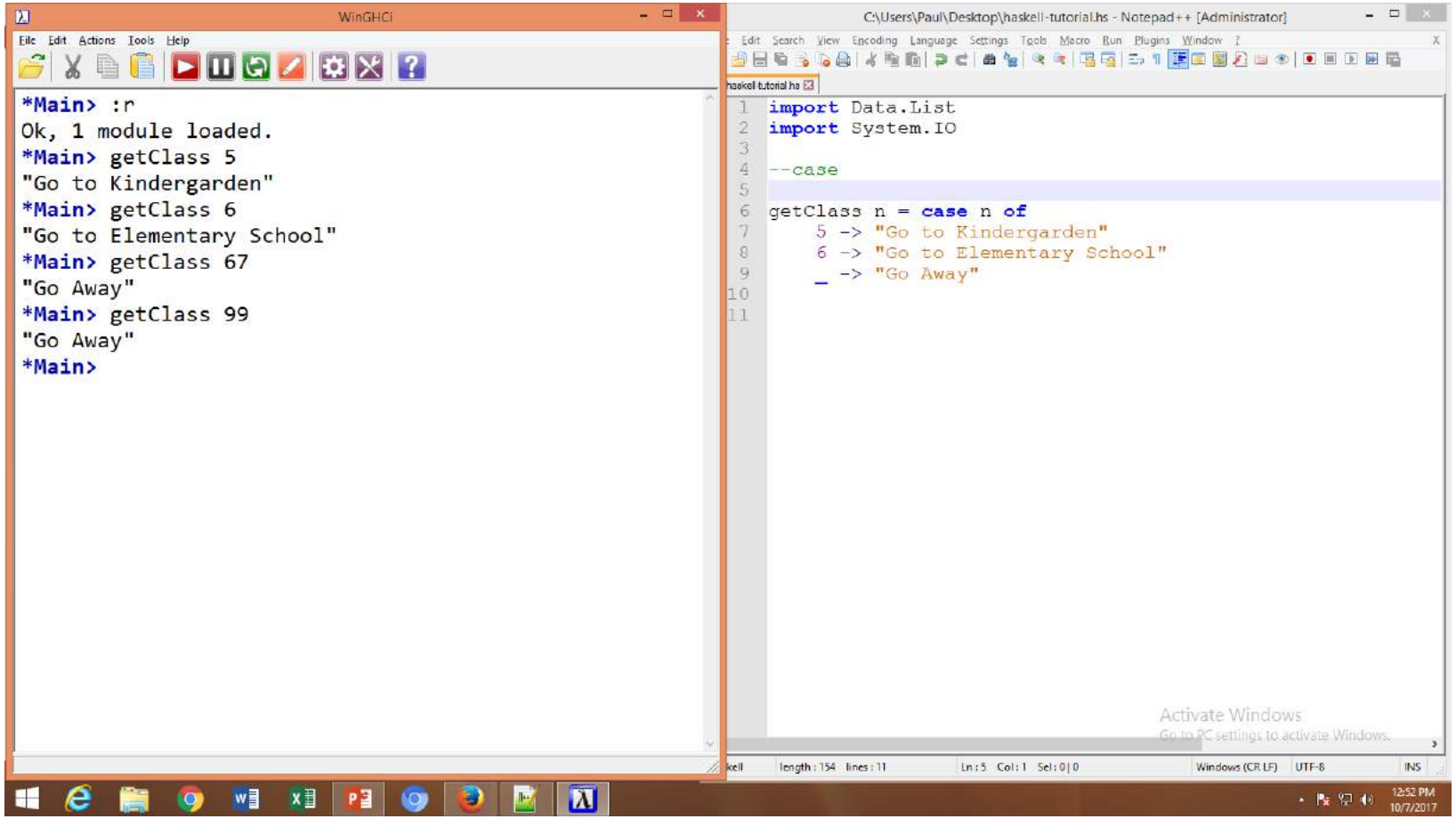
Other Operators

- Comparison
  - `<` --less than
  - `>` --greater than
  - `<=` --less than equal to
  - `>=` --greater than equal to
  - `==` --equal to
- Logical
  - `&&` --AND
  - `||` --OR
  - `not` --NOT

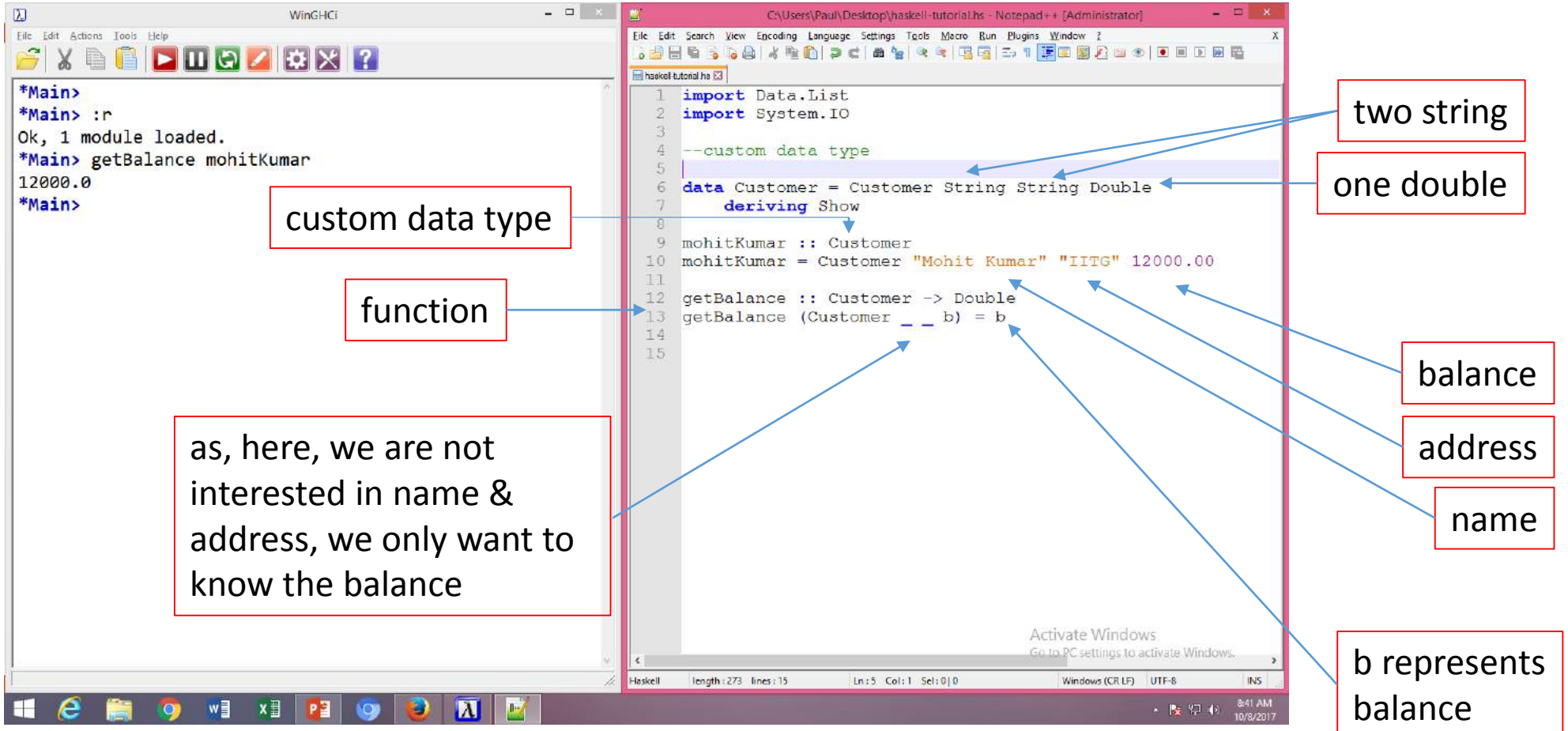
Example



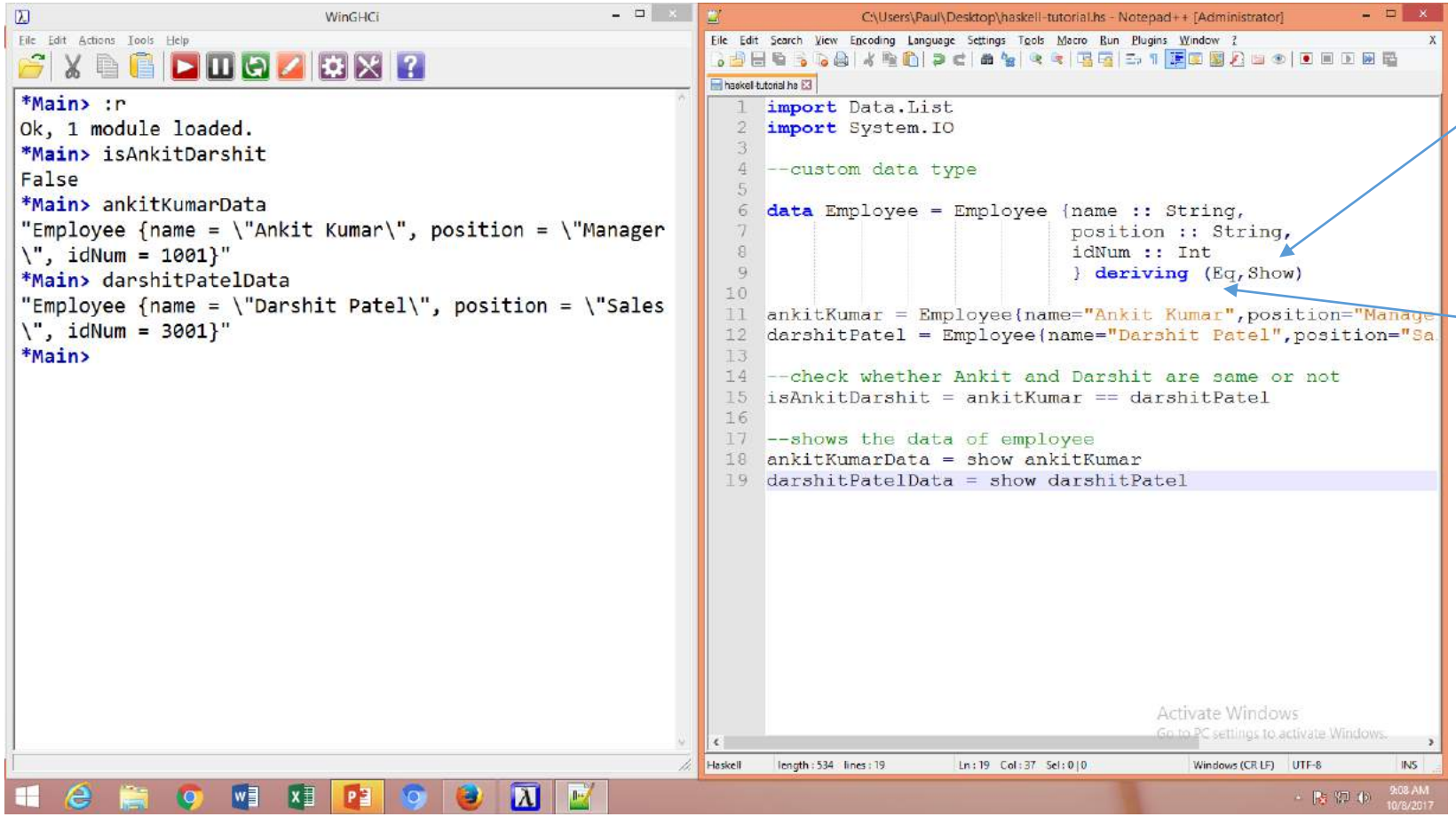
Case


WinGHCi
File Edit Actions Tools Help
*Main> :r
Ok, 1 module loaded.
*Main> getClass 5
"Go to Kindergarden"
*Main> getClass 6
"Go to Elementary School"
*Main> getClass 67
"Go Away"
*Main> getClass 99
"Go Away"
*Main>
C:\Users\Paul\Desktop\haskell-tutorial.hs - Notepad++ [Administrator]
haskell-tutorial.hs
1 import Data.List
2 import System.IO
3
4 --case
5
6 getClass n = case n of
7 5 -> "Go to Kindergarden"
8 6 -> "Go to Elementary School"
9 _ -> "Go Away"
10
11
Activate Windows
Go to PC settings to activate Windows.
length : 154 lines : 11
Ln : 5 Col : 1 Sel : 0 | 0
Windows (CR LF)
UTF-8
INS
12:52 PM
10/7/2017

Custom Data Type



Type Classes



able to show the employee details

able to check for the equality

END OF TUTORIAL

YOU MAY EXPLORE

http://www.learnyouahaskell.com

FOR MORE DETAIL