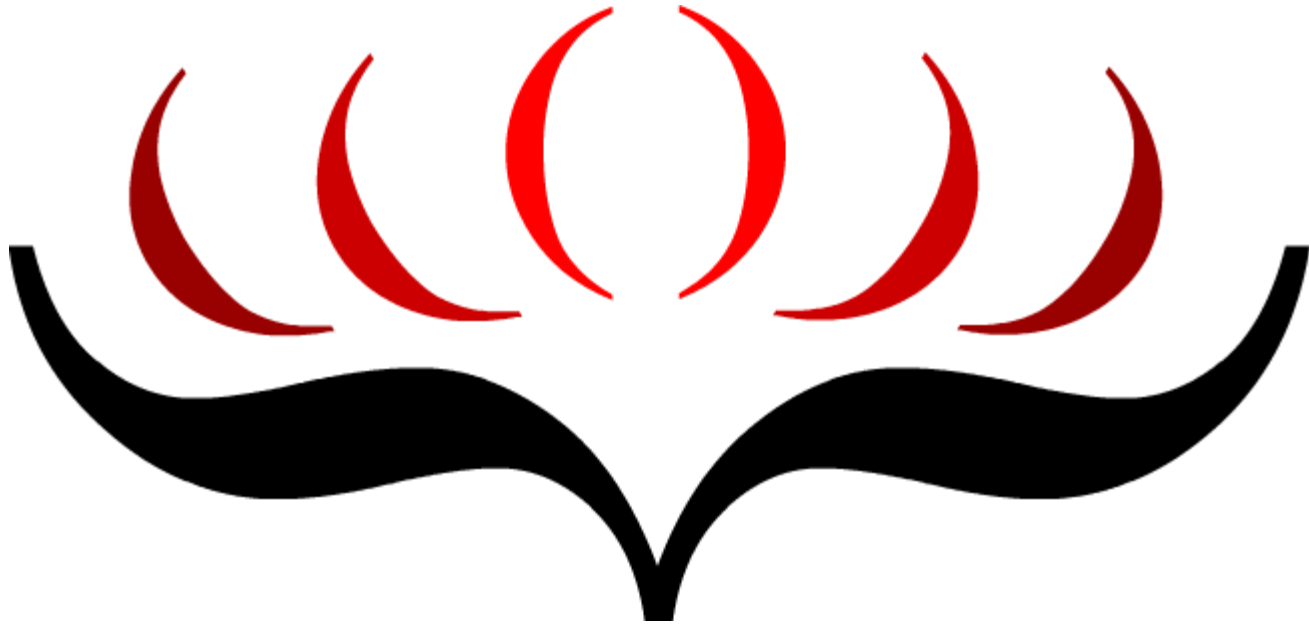


The first steps in Haskell



Vladimir Alekseichenko

Agenda

- Your Haskell environment
- Getting started with ghc*
- Basic types and definitions
- Types and functions
- A first Haskell program

The first steps...



Your Haskell environment

Haskell is a language with many implementations, two of which are widely used:

- Hugs
- GHC

Hugs



Haskell User's Gofer System Hugs



Hugs 98

It's a functional programming system based on Haskell 98, the de facto standard for non-strict functional programming languages.

Hugs 98 provides an almost complete implementation of Haskell 98

Hugs

Version: September

Version: September 2006

Hugs 98: Based on the Haskell 98 standard

Copyright (c) 1994-2005

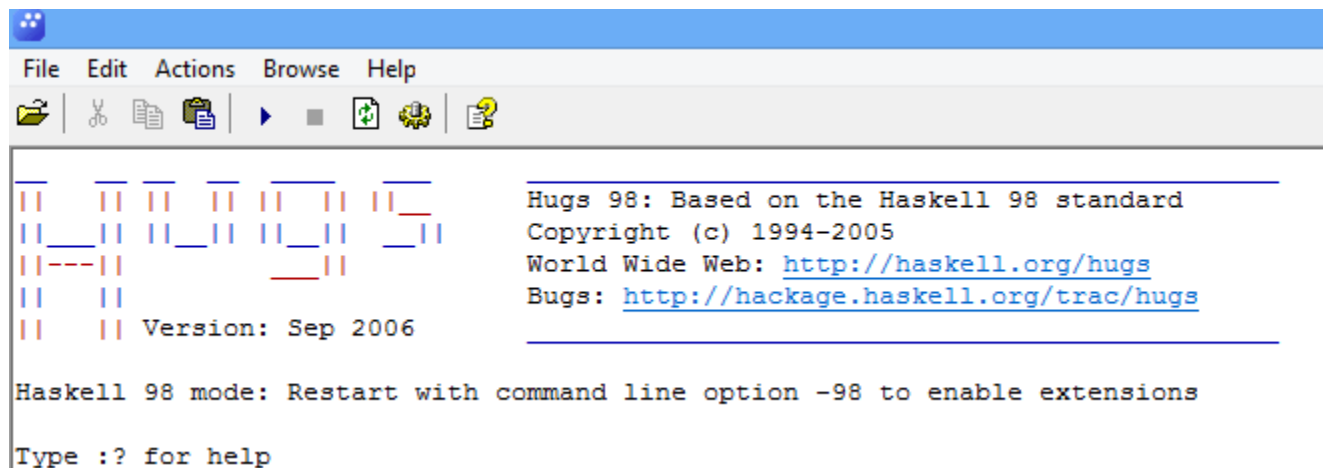
World Wide Web: <http://haskell.org/hugs>

Bugs: <http://hackage.haskell.org/trac/hugs>

```
Haskell 98 mode: Restart with command line option -98 to enable extensions
```

Type :? for help

WinHugs (0)



```
File Edit Actions Browse Help
[Icons]

||  ||  ||  ||  ||  ||  || | |
||__||  ||__||  ||__||  ||__||
||--||      ||
||  ||
||  || Version: Sep 2006

Hugs 98: Based on the Haskell 98 standard
Copyright (c) 1994-2005
World Wide Web: http://haskell.org/hugs
Bugs: http://hackage.haskell.org/trac/hugs

Haskell 98 mode: Restart with command line option -98 to enable extensions

Type :? for help
```

WinHugs (1)

```
Hugs> :? help
```



LIST OF COMMANDS: Any command may be abbreviated to :c where c is the first character in the full name.

:load <filenames>	load modules from specified files
:load	clear all files except prelude
:also <filenames>	read additional modules
:reload	repeat last load command
:edit <filename>	edit file
:edit	edit last module
:module <module>	set module for evaluating expressions
<expr>	evaluate expression
:type <expr>	print type of expression
:?	display this list of commands
:set <options>	set command line options
:set	help on command line options
:names [pat]	list names currently in scope
:info <names>	describe named objects
:browse <modules>	browse names exported by <modules>
:main <aruments>	run the main function with the given arguments
:find <name>	edit module containing definition of name
:cd dir	change directory
:gc	force garbage collection
:version	print Hugs version
:quit	exit Hugs interpreter

WinHugs (2)

```
Hugs> 1
1
Hugs> :set +t
Hugs> 1
1 :: Integer
Hugs> :set -t
Hugs> 1
1
```

GHC



The Glorious Glasgow Haskell Compilation System

It's an open source native code
compiler for the functional
programming language Haskell.

More commonly known as the Glasgow Haskell
Compiler or GHC.

GHC has three main components

- `ghc` - an optimizing compiler that generates fast native code.
- `ghci` - an interactive interpreter and debugger.
- `runghc` - a program for running Haskell programs as scripts, without needing to compile them first.

The Haskell Platform



Windows



Mac



Linux

Comprehensive

The Haskell Platform is the easiest way to get started with programming Haskell. It comes with all you need to get up and running. Think of it as "Haskell: batteries included". [Learn more...](#)

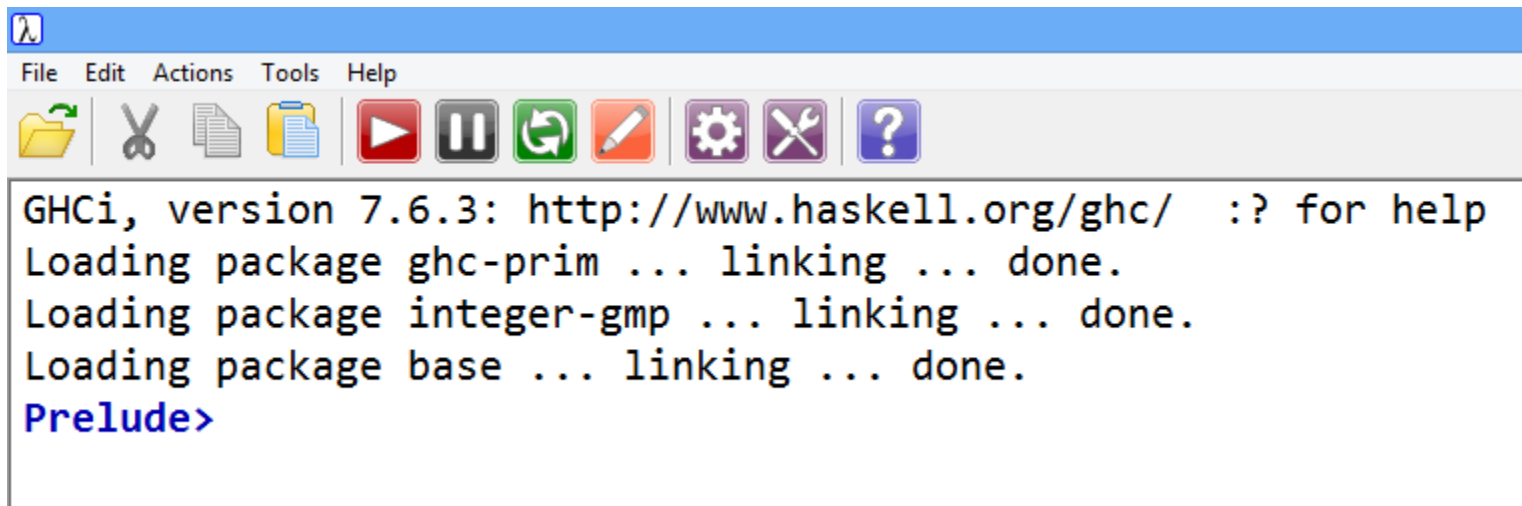
Robust

The Haskell Platform contains only stable and widely-used tools and libraries, drawn from a pool of thousands of [Haskell packages](#), ensuring you get the best from what is on offer.

Cutting Edge

The Haskell Platform ships with advanced features such as multicore parallelism, thread sparks and transactional memory, along with [many other technologies](#), to help you get work done.

WinGHCi (0)



WinGHCi (1)

Prelude> `:? help`



Commands available from the prompt:

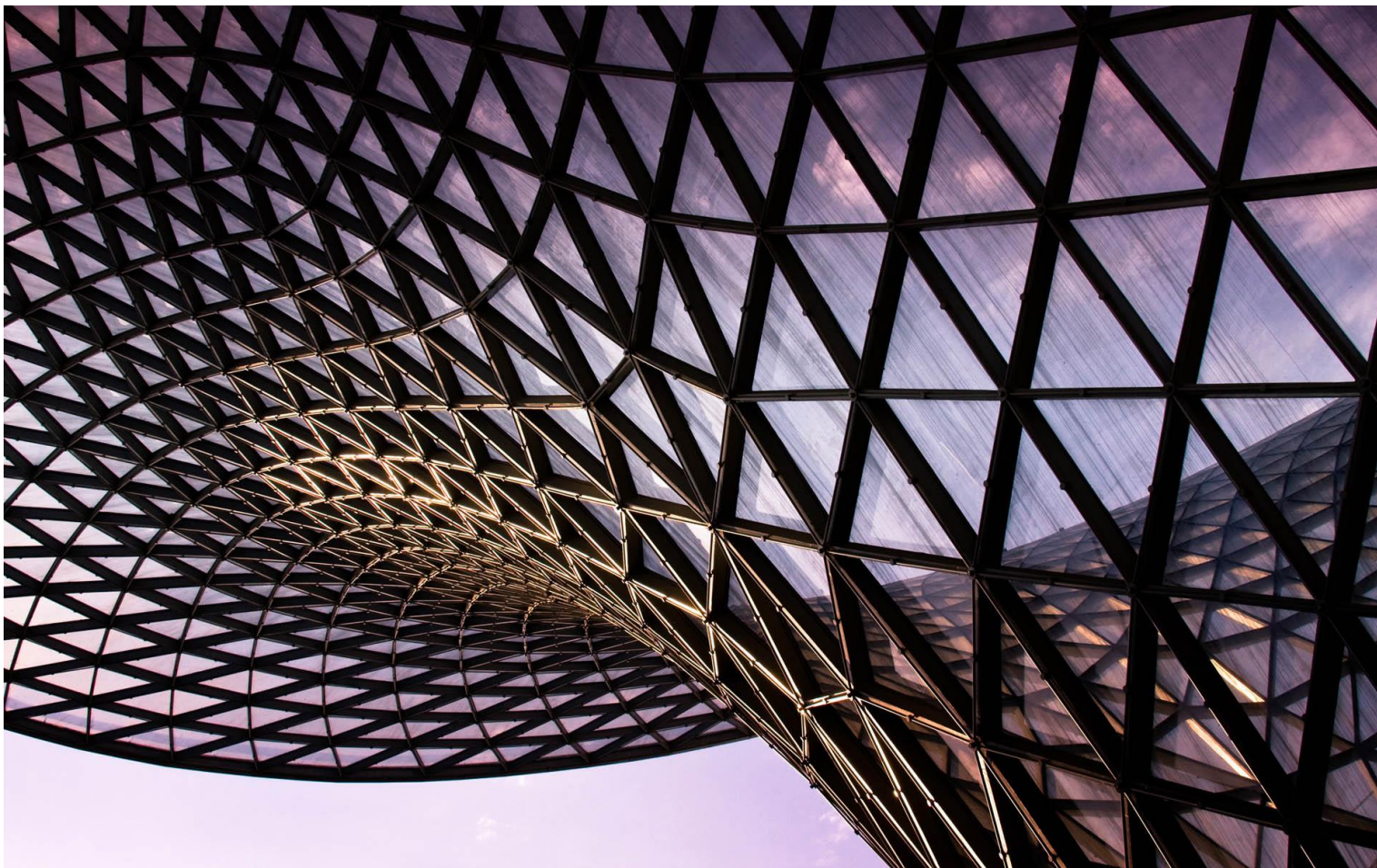
<code><statement></code>	evaluate/run <code><statement></code>
<code>:</code>	repeat last command
<code>:{\n ..lines.. \n:}\n</code>	multiline command
<code>:add [*]<module> ...</code>	add module(s) to the current target set
<code>:browse[!] [[*]<mod>]</code>	display the names defined by module <code><mod></code> (<code>!:</code> more details; <code>*</code> : all top-level names)
<code>:cd <dir></code>	change directory to <code><dir></code>
<code>:cmd <expr></code>	run the commands returned by <code><expr>::IO String</code>
<code>:ctags[!] [<file>]</code>	create tags file for Vi (default: "tags") (<code>!</code> : use regex instead of line number)
<code>:def <cmd> <expr></code>	define command <code>:<cmd></code> (later defined command has precedence, <code>::<cmd></code> is always a builtin command)
<code>:edit <file></code>	edit file

WinGHCi (2)

Prelude> `:i Int`

```
data Int = GHC.Types.I# GHC.Prim.Int#      -- Defined in `GHC.Types'
instance Bounded Int -- Defined in `GHC.Enum'
instance Enum Int -- Defined in `GHC.Enum'
instance Eq Int -- Defined in `GHC.Classes'
instance Integral Int -- Defined in `GHC.Real'
instance Num Int -- Defined in `GHC.Num'
instance Ord Int -- Defined in `GHC.Classes'
instance Read Int -- Defined in `GHC.Read'
instance Real Int -- Defined in `GHC.Real'
instance Show Int -- Defined in `GHC.Show'
```

The basic elements



Lists

- []
- [1, 2, 3]
- 1 : [2, 3]
- 1 : (2 : (3 : []))
- [1..n]
- [2, 4..20]



Bad lists

- `1 : „two” : []`
- `[1::Int, 1::Integer]`
- `[1, 2, „third”]`
- `[1, 5, 9..100]`
- `[1,2..10,20..100]`
- `[(1, ,a'), (,b', 2)]`
- `[[1, 2, 3], [,a', ,b', ,c']]`



Tuples



- (1,2)
- (1,2,3)
- ...
- (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15)
- („a“, 1)
- („a“, [], („a“, „b“), 1)
- ((1, 2), („a‘, „b‘, „c‘), ([1], [2]))

Bad tuples

- (1) – it's not tuple
- (1, (2)) - it's not tuple
- (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16)



Types of lists

- [Int]
- [Integer]
- [Char]
- [(Integer, Char)]
- [[Integer]]

Types of lists

```
Prelude> [1, 2, 3]::[Int]
[1,2,3]
it :: [Int]
Prelude> [1, 2, 3]
[1,2,3]
it :: [Integer]
Prelude> ['a', 'b', 'c']
"abc"
it :: [Char]
Prelude> [(1, 'a')]
[(1,'a')]
it :: [(Integer, Char)]
Prelude> [[], [1], [1..4]]
[[],[1],[1,2,3,4]]
it :: [[Integer]]
```

Types of tuples

- (Integer, Integer)
- (Char, Integer)
- (Char, [Integer])
- (Char, (Integer, Integer), [[Integer]])

Types of tuples

```
Prelude> (1, 2)
```

```
(1,2)
```

```
it :: (Integer, Integer)
```

```
Prelude> ('a', 56)
```

```
('a',56)
```

```
it :: (Char, Integer)
```

```
Prelude> ('a', [1, 2, 3])
```

```
('a',[1,2,3])
```

```
it :: (Char, [Integer])
```

```
Prelude> ('a', (5, 7, 9), [[], [1], [1..3]])
```

```
('a',(5,7,9),[[],[1],[1,2,3]])
```

```
it :: (Char, (Integer, Integer, Integer), [[Integer]])
```

Type

Every expression and function in Haskell has a *type*.

For example, the value `True` has the type `Bool`, while the value `"foo"` has the type `String`.

Some common basic **types**

- Char
- Bool
- Int
- Integer
- Double

Some common basic **types**

```
Prelude> :type 'a'
'a' :: Char
Prelude> :t True
True :: Bool
Prelude> :set +t
Prelude> 1.1
1.1
it :: Double
```

New types

We've seen two ways to introduce new types:

- **data**
- **type**

Type synonyms

Creates an alias for an existing type, with no overhead.

Type synonyms

```
1 module Distance where
2
3 type Coord    = (Double, Double)
4 type Pair a   = (a, a)
5 type Complex = Pair Double
6
7 distance :: Coord -> Coord -> Double
8 distance (x0, y0) (x1, y1) = sqrt( (x1-x0)^2 + (y1-y0)^2)
```

Both **Coord** and **Complex** are **(Double, Double)!**

Distance

```
Prelude> :l "Distance.hs"  
[1 of 1] Compiling Distance          ( Distance.hs, interpreted )  
Ok, modules loaded: Distance.  
*Distance> distance (1.0, 2.0) (2.0, 3.0)  
1.4142135623730951  
it :: Double
```

Data.String

```
type String = [Char]
```

```
Prelude> :t ""
```

```
"" :: [Char]
```

```
Prelude> ['a', 'b', 'c'] == "abc"
```

```
True
```

```
it :: Bool
```

Data types

Creates a new (boxed) type,
adding overhead of a Val
wrapper.

MagicType

```
1 module MagicType where
2
3 data MagicType = AbraKadabra Int Int Int | SimSalabim Int Int
4
5 magicFun :: MagicType -> Int
6 magicFun (AbraKadabra a b c) = a + b + c
7 magicFun (SimSalabim a b) = a + b
```

MagicType

```
Prelude> :l "MagicType.hs"
```

```
[1 of 1] Compiling MagicType          ( MagicType.hs, interpreted )
```

```
Ok, modules loaded: MagicType.
```

```
*MagicType> magicFun (AbraKadabra 1 2 3)
```

```
6
```

```
it :: Int
```

```
*MagicType> magicFun (SimSalabim 1 2)
```

```
3
```

```
it :: Int
```

Distance (0)

```
1 module Distance where
2
3 data Coord = Coord Double Double
4 data Pair a = Couple a a
5 data Complex = Pair Double
6
7 distance :: Coord -> Coord -> Double
8 distance (Coord x0 y0) (Coord x1 y1) = sqrt( (x1-x0)^2 + (y1-y0)^2)
```

Distance (1)

```
*Distance> :l "Distance.hs"
```

```
[1 of 1] Compiling Distance          ( Distance.hs, interpreted )
```

```
Ok, modules loaded: Distance.
```

```
*Distance> distance (Coord 1.0 2.0) (Coord 2.0 3.0)
```

```
1.4142135623730951
```

```
it :: Double
```

```
*Distance> distance (Complex 1.0 2.0) (Complex 2.0 3.0)
```

```
<interactive>:246:11:
```

```
Not in scope: data constructor `Complex'
```

```
Perhaps you meant `Couple' (line 4)
```

```
<interactive>:246:29:
```

```
Not in scope: data constructor `Complex'
```

```
Perhaps you meant `Couple' (line 4)
```


Date (0)

```
1 module Date where
2
3 data Date = Date Year Month Day
4
5 data Year = Year Int
6
7 data Month = January | February | March | April
8             | May | June | July | August
9             | September | October | November | December
10
```

Date (1)

```
11 data Day = Day Int
12
13 data Week = Monday | Tuesday | Wednesday
14           | Thursday | Friday | Saturday
15           | Sunday
16
17 data Time = Time Hour Minute Second
18 data Hour = Hour Int
19 data Minute = Minute Int
20 data Second = Second Int
```

Weekend

```
22 weekend :: Week -> Bool
23 weekend Saturday = True
24 weekend Sunday   = True
25 weekend _         = False
```

Weekend

```
*Date> :l "Date.hs"
```

```
[1 of 1] Compiling Date  
Ok, modules loaded: Date.
```

```
*Date> weekend Monday
```

```
False
```

```
it :: Bool
```

```
*Date> weekend Sunday
```

```
True
```

```
it :: Bool
```

```
*Date> weekend Saturday
```

```
True
```

```
it :: Bool
```

```
( Date.hs, interpreted )
```

Write first program



Types of programs

Haskell programs are of two types:

- executable
- library

Five rules

1. The program consists of modules.
2. One module - one single file.
3. The module name is the file name.
4. The module name begins with a capital letter.
5. File has the extension **.hs**.

Notepad

```
| Prelude Fibonacci> :edit "Fibonacci.hs"
```



Notepad++



Change the WinGHCi editor

```
Prelude Fibonacci> :set editor "C:\\Program Files (x86)\\Notepad++\\notepad++.exe"  
Prelude Fibonacci> :e " Fibonacci.hs"  
'C:\\\\Program' is not recognized as an internal or external command,  
operable program or batch file.  
Prelude Fibonacci> :set editor "C:\\\\"Program Files (x86)\\Notepad++\\notepad++.exe"  
Prelude Fibonacci> :e " Fibonacci.hs"  
Ok, modules loaded: Fibonacci.
```

Sublime Text



Change the WinGHCi editor

```
Prelude> :set editor "C:\\Program Files\\Sublime Text 3\\sublime_text.exe"  
Prelude>
```

Fibonacci number

In mathematical terms, the sequence F_n of Fibonacci numbers is defined by the recurrence relations:

$$F_n = F_{n-1} + F_{n-2},$$

with seed values:

$$F_0 = 0, F_1 = 1.$$

Fibonacci number (0)

```
1 module Fibonacci where
2
3 fibonacci    :: Integer -> Integer
4
5 fibonacci 0 = 0
6 fibonacci 1 = 1
7 fibonacci n = fibonacci (n-1) + fibonacci (n-2)
```

Fibonacci number (0)

```
1 module Fibonacci where
2
3 fibonacci    :: Integer -> Integer
4
5 fibonacci 0 = 0
6 fibonacci 1 = 1
7 fibonacci n = fibonacci (n-1) + fibonacci (n-2)
```

Fibonacci number (1)

```
1 module Fibonacci where
2
3 fibonacci    :: Integer -> Integer
4 fibonacci n = case n of
5             0 -> 0
6             1 -> 1
7             _ -> fibonacci (n-1) + fibonacci (n-2)
```


Fibonacci number (1)

```
1 module Fibonacci where
2
3 fibonacci :: Integer -> Integer
4 fibonacci n = case n of
5     0 -> 0
6     1 -> 1
7     _ -> fibonacci (n-1) + fibonacci (n-2)
```

Fibonacci number (2)

```
1 module Fibonacci where
2
3 fibonacci      :: Integer -> Integer
4
5 fibonacci n | n == 0 = 0
6             | n == 1 = 1
7             | n > 1  = fibonacci (n-1) + fibonacci (n-2)
```

Fibonacci number (2)

```
1 module Fibonacci where
2
3 fibonacci :: Integer -> Integer
4
5 fibonacci n | n == 0 = 0
6             | n == 1 = 1
7             | n > 1  = fibonacci (n-1) + fibonacci (n-2)
```

Fibonacci number (3)

```
1 module Fibonacci where
2
3 fibonacci    :: Integer -> Integer
4
5 fibonacci n =
6     if n == 0
7         then 0
8     else if n == 1
9         then 1
10        else fibonacci (n-1) + fibonacci (n-2)
```

Fibonacci number (3)

```
1 module Fibonacci where
2
3 fibonacci :: Integer -> Integer
4
5 fibonacci n =
6     if n == 0
7         then 0
8         else if n == 1
9             then 1
10            else fibonacci (n-1) + fibonacci (n-2)
```

Loading source files

```
Prelude> :cd D:\src\haskell
```

```
Prelude> :load "Fibonacci.hs"
```

```
[1 of 1] Compiling Fibonacci
```

```
Ok, modules loaded: Fibonacci.
```

```
*Fibonacci>
```

```
( Fibonacci.hs, interpreted )
```

Running the program

```
Prelude> :l "Fibonacci.hs"  
[1 of 1] Compiling Fibonacci  
Ok, modules loaded: Fibonacci.  
*Fibonacci> fibonacci 10  
55  
it :: Integer
```

(Fibonacci.hs, interpreted)

Working with lists

```
head      :: [a] -> a
head (x:_) = x
head []    = error "head: empty list"
```

```
tail      :: [a] -> [a]
tail (x:xs) = xs
tail []    = error "tail : empty list"
```

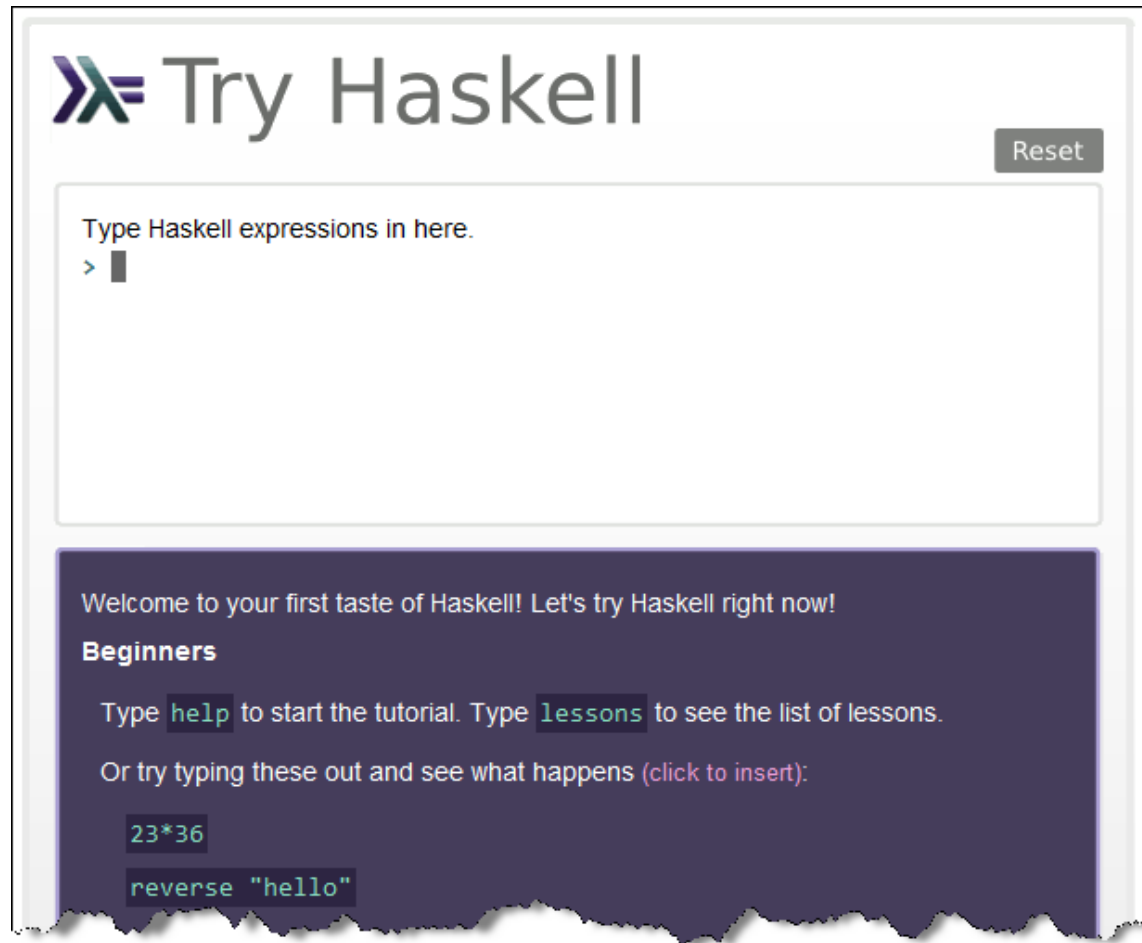
```
length    :: [a] -> Int
length (x:xs) = 1 + length xs
length []    = 0
```

```
null      :: [a] -> Bool
null []    = True
null _     = False
```


Working with lists

```
Prelude> head [1, 2, 3]
1
it :: Integer
Prelude> tail [1, 2, 3]
[2,3]
it :: [Integer]
Prelude> length [1, 2, 3]
3
it :: Int
Prelude> null [1, 2, 3]
False
it :: Bool
```

TryHaskell.org



Books

- **Real World Haskell** by *Bryan O'Sullivan, Don Stewart, John Goerzen*.
- **Learn You a Haskell for Great Good!** by *Miran Lipovača*.
- **Haskell: The Craft of Functional Programming** by *Simon Thompson*.