GHCi: Getting started (1A)

Young Won Lim

Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using OpenOffice.

Based on

Haskell in 5 steps
https://wiki.haskell.org/Haskell_in_5_steps

Interpreter GHCi

```
young@MNTSys-BB1 ~ $ ghci
GHCi, version 7.10.3: http://www.haskell.org/ghc/  :? for help
Prelude> "hello, world!"
"hello, world!"
Prelude> putStrLn "hello, world!"
hello, world!
```

https://wiki.haskell.org/Learn_Haskell_in_10_minutes

Function

```
Prelude> let fac n = if n == 0 then 1 else n * fac (n-1)
Prelude> fac 5
120
Prelude> fac 2
2
Prelude> fac 3
6
Prelude> fac 4
24
Prelude>
```

https://wiki.haskell.org/Learn_Haskell_in_10_minutes

Compiler GHC

```
young@MNTSys-BB1 ~ $ ghc -o hello hello.hs
[1 of 1] Compiling Main             ( hello.hs, hello.o )
Linking hello ...
young@MNTSys-BB1 ~ $ ./hello
hello, world!
```

```
young@MNTSys-BB1 ~ $ cat hello.hs
main = putStrLn "hello, world!"
```

https://wiki.haskell.org/Learn_Haskell_in_10_minutes

Layout

t.hs

```
main = do putStrLn "Type an integer : ?"
          x <- readLn
          if even x
            then putStrLn "even number"
            else putStrLn "odd number"
```

the first non-space character after **do.**

every line that starts in the same column
as that p is in the d**o** block

If you indent more, it is the nested block in **do**

If you indent less, it is an end of the **do** block.

ghc t.hs

./t

ghc –o run t.hs

./t

Multi-line in GHCi

ghci multi-line

```
Prelude> :{
Prelude| main = do { putStrLn "Type an integer: "; x<-readLn;
Prelude| if even x then putStrLn "even" else putStrLn "odd"; }
Prelude| :}
```

https://wiki.haskell.org/Learn_Haskell_in_10_minutes

Types

Int	an integer with at least 30 bits of precision.
Integer	an integer with unlimited precision.
Float	a single precision floating point number.
Double	a double precision floating point number.
Rational	a fraction type, with no rounding error.

Types and Class Types start with capital letters

Variables start with lower case letters

Declaring a type	:: type
Asking which type	:t something

https://wiki.haskell.org/Learn_Haskell_in_10_minutes

Type Classes

```
Prelude> 3 :: Int
3
Prelude> 3 :: Float
3.0
Prelude> 4 :: Double
4.0
Prelude> 2 :: Integer
2
Prelude> :t 3
3 :: Num a => a
Prelude> :t 2.0
2.0 :: Fractional a => a
Prelude> :t gcd 15 20
gcd 15 20 :: Integral a => a
Prelude> :t True
True :: Bool
Prelude> :t 'A'
'A' :: Char
```

class constraint

the type t is *constrained* by the context (Num t), (Fractional t), (Integral t)

(Num t) => the **types** of t must be **Num type class**

(Fractional t) => the **types** of t must be **Fractional type class**

(Integral t) => the **types** of t must be **Integral type class**

3 can be used as any numeric type

2.0 can be used as any fractional type

gcd 15 20 can be used as any integral type

https://wiki.haskell.org/Learn_Haskell_in_10_minutes

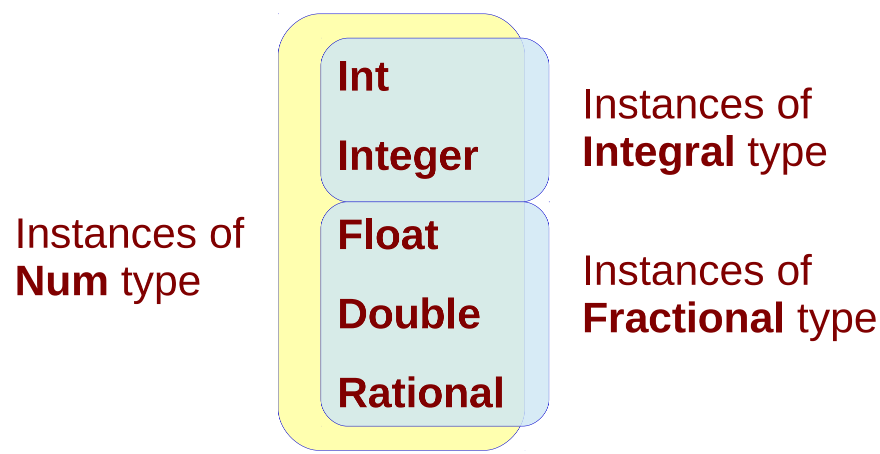
Type Classes

Int
Integer } Instances of **Integral** type

Float
Double
Rational } Instances of **Fractional** type

All five: Instances of **Num** type

Type Class : a set of type (instances)



https://wiki.haskell.org/Learn_Haskell_in_10_minutes

Lists and Tuples

Lists multiple values of the same type

Strings lists of characters.

Tuples a fixed number of values, which can have different types.

The **:** operator appends an item to the beginning of a list

Zip : two lists into a list of tuples.

https://wiki.haskell.org/Learn_Haskell_in_10_minutes

Functions

[1 .. 10]	[1,2,3,4,5,6,7,8,9,10]
map (+ 2) [1 .. 10]	[3,4,5,6,7,8,9,10,11,12]
filter (> 2) [1 .. 10]	[3,4,5,6,7,8,9,10]
fst (1, 2)	1
snd (1, 2)	2
map fst [(1, 2), (3, 4), (5, 6)]	[1,3,5]

~~fst (1, 2, 3)~~

~~snd (1, 2, 3)~~

https://wiki.haskell.org/Learn_Haskell_in_10_minutes

Functions

```
my_sum m n = m+n

main = do putStrLn "Give two numbers: "
          x <- readLn
          y <- readLn
          print (my_sum x y)
```

```
Give two numbers:
10
20
30
```

https://wiki.haskell.org/Learn_Haskell_in_10_minutes

Convenient Syntax

```
secsToWeeks secs = let perMinute = 60
                       perHour   = 60 * perMinute
                       perDay    = 24 * perHour
                       perWeek   =  7 * perDay
                   in  secs / perWeek
```

```
classify age = case age of 0 -> "newborn"
                           1 -> "infant"
                           2 -> "toddler"
                           _ -> "senior citizen"
```

https://wiki.haskell.org/Learn_Haskell_in_10_minutes

Using Libraries

```
import Prelude hiding (lookup)
import Data.Map

employeeDept      = fromList([("John","Sales"),      ("Bob","IT")])
deptCountry       = fromList([("IT","USA"),          ("Sales","France")])
countryCurrency   = fromList([("USA", "Dollar"),     ("France", "Euro")])

employeeCurrency :: String -> Maybe String
employeeCurrency name = do
  dept    <- lookup name    employeeDept
  country <- lookup dept    deptCountry
             lookup country countryCurrency

main = do
  putStrLn $ "John's currency: " ++ (show (employeeCurrency "John"))
  putStrLn $ "Pete's currency: " ++ (show (employeeCurrency "Pete"))
```

https://downloads.haskell.org/~ghc/latest/docs/html/libraries/containers-0.5.7.1/Data-Map-Lazy.html

fromList (1)

fromList :: Eq key => (key -> Int32) -> [(key, val)] -> IO (HashTable key val)

base Data.HashTable

Convert a list of key/value pairs into a hash table. Equality on keys is taken from the Eq instance for the key type.

fromList :: [(Key, a)] -> IntMap a

containers Data.IntMap.Strict, containers Data.IntMap.Lazy

O(n*min(n,W)). Create a map from a list of key/value pairs.

> fromList [] == empty

> fromList [(5,"a"), (3,"b"), (5, "c")] == fromList [(5,"c"), (3,"b")]

> fromList [(5,"c"), (3,"b"), (5, "a")] == fromList [(5,"a"), (3,"b")]

fromList :: [Key] -> IntSet

containers Data.IntSet

O(n*min(n,W)). Create a set from a list of integers.

fromList :: [a] -> Seq a

containers Data.Sequence

O(n). Create a sequence from a finite list of elements. There is a function toList in the opposite direction for all instances of the Foldable class, including Seq.

https://www.haskell.org/hoogle/?hoogle=fromList

fromList (2)

fromList :: Ord a => [a] -> Set a

containers Data.Set

O(n*log n). Create a set from a list of elements. If the elemens are ordered, linear-time implementation is used, with the performance equal to fromDistinctAscList.

fromList :: Ord k => [(k, a)] -> Map k a

containers Data.Map.Lazy, containers Data.Map.Strict

O(n*log n). Build a map from a list of key/value pairs. See also fromAscList. If the list contains more than one value for the same key, the last value for the key is retained. If the keys of the list are ordered, linear-time implementation is used, with the performance equal to fromDistinctAscList.

> fromList [] == empty

> fromList [(5,"a"), (3,"b"), (5, "c")] == fromList [(5,"c"), (3,"b")]

> fromList [(5,"c"), (3,"b"), (5, "a")] == fromList [(5,"a"), (3,"b")]

https://www.haskell.org/hoogle/?hoogle=fromList

lookup (1)

lookup :: Eq a => a -> [(a, b)] -> Maybe b
base Prelude, base Data.List
lookup key assocs looks up a key in an association list.

lookup :: HashTable key val -> key -> IO (Maybe val)
base Data.HashTable
Looks up the value of a key in the hash table.

lookup :: Key -> IntMap a -> Maybe a
containers Data.IntMap.Strict, containers Data.IntMap.Lazy
O(min(n,W)). Lookup the value at a key in the map. See also lookup.

lookup :: Ord k => k -> Map k a -> Maybe a
containers Data.Map.Lazy, containers Data.Map.Strict
O(log n). Lookup the value at a key in the map. The function will return the corresponding value as (Just value), or Nothing if the key isn't in the map. An example of using lookup:

https://www.haskell.org/hoogle/?hoogle=fromList

lookup (2)

```
> import Prelude hiding (lookup)
> import Data.Map
>
> employeeDept    = fromList( [   ("John",  "Sales"),     ("Bob",      "IT")           ] )
> deptCountry     = fromList( [   ("IT",      "USA"),       ("Sales",    "France")   ] )
> countryCurrency = fromList( [   ("USA",   "Dollar"),     ("France", "Euro")        ] )
>
> employeeCurrency :: String -> Maybe String
> employeeCurrency name = do
> dept        <-      lookup name    employeeDept
> country    <-      lookup dept      deptCountry
>                       lookup country countryCurrency
>
> main = do
> putStrLn $ "John's currency: " ++ (show (employeeCurrency "John"))
> putStrLn $ "Pete's currency: " ++ (show (employeeCurrency "Pete"))
```

The output of this program:

```
> John's currency: Just "Euro"
> Pete's currency: Nothing
```

https://www.haskell.org/hoogle/?hoogle=fromList

elem

elem :: Eq a => a -> [a] -> Bool
base Prelude, base Data.List

elem is the list membership predicate,
usually written in infix form, e.g., x `elem` xs.
For the result to be False, the list must be finite;
True, however, results from an element equal to x found
at a finite index of a finite or infinite list.

```
1 `elem` [1, 2, 4] -- True
2 `elem` [1, 2, 4] -- True
3 `elem` [1, 2, 4] -- False
```

https://www.haskell.org/hoogle/?hoogle=fromList

Generator

```
let removeLower x=[c| c<-x, c `elem` ['A'..'Z']]
```

a list comprehension

```
[c | c<-x, c `elem` ['A'..'Z']]
```

c <- x is a **generator**

c is a **pattern**

- to be matched from the elements of the list x
- to be successively bound to the elements of the input list x

```
c `elem` ['A'..'Z']
```

is a **predicate** which is applied to each successive binding of c inside the comprehension

an element of the input only appears in the output list if it passes this predicate.

https://stackoverflow.com/questions/35198897/does-mean-assigning-a-variable-in-haskell

Assignment in Haskell

Assignment in Haskell : declaration with initialization:

You declare a variable;
Haskell doesn't allow uninitialized variables,
so an initial value must be supplied in the declaration
There's no mutation, so the value given in the declaration
will be the only value for that variable throughout its scope.

https://stackoverflow.com/questions/35198897/does-mean-assigning-a-variable-in-haskell

Assignment in Haskell

```
filter (`elem` ['A' .. 'Z']) x
```

```
[c| c <- x]
```

```
do c <- x
   return c
```

```
x >>= \c -> return c
```

```
x >>= return
```

https://stackoverflow.com/questions/35198897/does-mean-assigning-a-variable-in-haskell

Monad Class Function >>= & >>

both >>= and >> are functions from the Monad class.

>>= **passes** the result of the expression on the left
as an argument to the expression on the right,
in a way that respects the context the argument and function use

>> is used to **order** the evaluation of expressions within some context;
it makes evaluation of the right depend on the evaluation of the left

https://www.quora.com/What-do-the-symbols-and-mean-in-haskell

Monad – List Comprehension Examples

```
[x*2 | x<-[1..10], odd x]
```

```
do
   x <- [1..10]
   if odd x
       then [x*2]
       else []
```

```
[1..10] >>= (\x -> if odd x then [x*2] else [])
```

https://stackoverflow.com/questions/44965/what-is-a-monad

Monad – I/O Examples

```
do
  putStrLn "What is your name?"
  name <- getLine
  putStrLn ("Welcome, " ++ name ++ "!")
```

https://stackoverflow.com/questions/44965/what-is-a-monad

Monad – A Parser Example

```
parseExpr = parseString <|> parseNumber

parseString = do
        char '"'
        x <- many (noneOf "\"")
        char '"'
        return (StringValue x)

parseNumber = do
    num <- many1 digit
    return (NumberValue (read num))
```

https://stackoverflow.com/questions/44965/what-is-a-monad

Monad – Asynchronous Examples

```
let AsyncHttp(url:string) =
    async {  let req = WebRequest.Create(url)
             let! rsp = req.GetResponseAsync()
             use stream = rsp.GetResponseStream()
             use reader = new System.IO.StreamReader(stream)
             return reader.ReadToEnd() }
```

https://stackoverflow.com/questions/44965/what-is-a-monad

References

[1] ftp://ftp.geoinfo.tuwien.ac.at/navratil/HaskellTutorial.pdf
[2] https://www.umiacs.umd.edu/~hal/docs/daume02yaht.pdf