Notes - 6008 CEM

Saad Iftikhar

2022

Learning Haskell: The coding parts

Running Haskell code

Interpreter

Running haskel code via interpreter.

```
stack ghci
```

This can be ended using : quit or ctrl

When in compiler:

Running the code file in the interpreter.

```
:l <filename>.hs
```

Or use the following code to reload a pre loaded file

```
:r <filename>.hs
```

Compile Source Code to Executable

1. Type the following code (it creates an excutable):

   ```
   stack ghc <filename>.hs
   ```

   This runs the code in the terminal

   ```
   ./<filename>
   ```

2. Or the code can be run and compiled at the same time using the following code:

   ```
   stack runhaskell <filename>.hs
   ```

Basics: Numbers, Strings, Booleans

When subtracting numbers use brackets (….) so Haskell doent get confused by using the - sighn for arithmetic.

Numbers

For example: `5*(-3)`

There also buit in constants in Haskell.

For example: `pi`

The following code does not work as haskell can't add ints to floats:

(123::Int) + (123::Float)

Strings

Strings in Haskell can be concatenated using ++

For example: `"Hello" ++ "World"`

Booleans

`True`

`False`

Comparison operators

Less than: `<`

Greater than `>`

Less than or equal to: `<=`

Greater than or equal to: `>=`

Equal to: `==`

Not Equal to: `/=`

Built-in Functions in Haskell

code	does	output
succ 8	Gives **succeeding** number	9
min 44 33	Gives **min** of two numbers	33
44 \`min\` 33	The above function in **infix**.	
:type True	chekcs type for operators.	True :: Bool
:type 1	checks type of variables	1 :: Num p => p
:type max	checks type of function	max :: Ord a => a -> a -> a
:info Num	checks the type of a type. Gives the actions that can be performed on that type. Gives the type of the type.	class Num a where (+) :: a -> a -> a (-) :: a -> a -> a (*) :: a -> a -> a negate :: a -> a abs :: a -> a signum :: a -> a fromInteger :: Integer -> a {-# MINIMAL (+), (*), abs, signum, fromInteger, (negate | (-)) #-} -- Defined in 'GHC.Num' instance Num Word -- Defined in 'GHC.Num' instance Num Integer -- Defined in 'GHC.Num' instance Num Int -- Defined in 'GHC.Num' instance Num Float -- Defined in 'GHC.Float' instance Num Double -- Defined in 'GHC.Float'
:info Fractional	The first line of output: Tells that if **a** is **fraction** then it belongs to **Num**.	 **class Num** a => **Fractional** a **where**
:t	does the same thing as **:type**	
:info String	Gives type. Implies a string is a list of characters.	type String = [Char] -- Defined in 'GHC.Base'

Functions in Haskell

Example: `double x = 2*x`

Syntax: <Function_name> <parameter> = <something> <action> <parameter>

Two arguments can also be used: `adder a b = a+b`

The above functions can be called by using the following:

```
double 4
adder 3 6
```

$

```
Code:    double x = 2*x
Use1:    double 5+5
Output1: 15
Use2:    double$5+5
Output2: 20
```

Example 2:

```
Code:    map ($ 3) [(4+), (10*), (^2), sqrt]

Output:  [7.0,30.0,9.0,1.7320508075688772]
```

Composition

```
Code: map (negate . abs) [22,-33]
```

The above code allows using two functions together on a variable

Currying

The following function:

```
add a b = a+b
```

could be used as:

```
add 3 4
```

output: 7

but it can also be used as:

```
x = add 3
x 4
output: 7
```

As haskel performs the action in the fowllowing way.

```
(add 3) 4
```

The above code code be used in a simpler way.

```
add4 = (+4)
```

```
add4 3

7
```

Creating functions using prefix notation

```
(+) 3 5

Output: 8

Could be used as the following to create a function:

add a b = (+) a b
```

Or simply:

```
add = (+)
```

Null data

Null data in haskell gives a error and is called `undefined`

Higher Order Functions

Code: applyTwice `f x = f (f x)`

```
The above function takes a function as aparameter

Code:

divideByTen = (/10)
Code complined:
applyTwice divideByTen 100

Output: 1.0
```

Lambda functions in Haskell

Code: `filter (\ x -> x `mod` 2 == 0 ) [1..40]`

```
The above code could be read as.

Function parameter: \ x

Indicates what function does: ->

From the list: [1..40] filter the even numbers
```

Output: `[2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40]`

Lambda Pattern Matching

Code: `map (\(a,b) -> a + b) [(1,2),(3,5),(6,3),(2,6),(2,5)]`

Output: `[3,8,9,8,7]`

Command line arguments

```
import System.Environment
import Data.List

main = do
   args <- getArgs
   progName <- getProgName
   putStrLn "The arguments are:"
   mapM putStrLn args
   putStrLn "The program name is:"
   putStrLn progName
```

Importing Libraries in Haskell

import <moduleName>

Example: **import** Data.List

Example only imporing one or two functions.

import Data.List **hiding** (**nub**)

import Data.List (**nub**, **sort**)

Making your own modules

module <moduleName>
(<**sequenceOfNames**>) **where**
<function definitions>

Example:

```
module MyFirstListFunction
( sumFirstFive ) where
sumFirstFive aList = sum (take 5 aList)
```

Lists in Haskell

Lists in Haskell can only contain the same type of data.

Examples:

```
l1 = [1,2,3,4,5]
l2 = [True, False, True]
```

Raw list of charactes: `["H","e","l","l","o"]`

`Gives : "Hello"`

`A list of character: ["H","e","l","l","o"]`

`Gives: ["H","e","l","l","o"]`

`Or`

Raw list of charactes: :t ['h']

['h'] :: [Char]

`A list of character:` :t ["h"]

["h"] :: [[Char]]

Or by applying on just strings

:t "h"

Gives a list of characters: "h" :: [Char]

:t 'h'

Gives a character: 'h' :: Char

List comprehensions

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

The above code is read as:

x such that x is a list from 1 to 10 and x == x * 2

output: `[2,4,6,8,10,12,14,16,18,20]`

The above code using multiple statements:

```
[2*x | x <- [1..50], x^2 >= 25, x^3<300]
```

Outputs: `[10,12]`

Using multiple variables:

```
[ x++" "++y | x <- ["Hello","Goodbye"], y <- ["World", "Bob"]]
```

```
["Hello World","Hello Bob","Goodbye World","Goodbye Bob"]
```

Normal operations in lists

code	does	output
`[1..10]`	creates list from 1 to 10	`[1,2,3,4,5,6,7,8,9,10]`
`['a'..'z']`	Gives letters from a to z	`"abcdefghijklmnopqrstuvwxyz"`
head [1,2,3,4,5]	gives the first element	1
`[1..10]`	creates list from 1 to 10	`[1,2,3,4,5,6,7,8,9,10]`
`[1,2..]`	gives infinite list	
`"Hello"!!0`	Indexing. Gives first element.	`'H'`
`[1,2,3] ++ [4,5,6]`	contanation.	`[1,2,3,4,5,6]`
`2 : [1]` `1:2:3:[]`	Prepending	`2 : [1]` `[1,2,3]`

Built-in functions for lists in Haskell

code	does	output
`length [1,2,3,4,5]`	gives length of list	5
tail [1,2,3,4,5]	gives everything except first entry	[2,3,4,5]
`null [1,2,3,4,5]`	checks if a list is empty	`False`
`reverse [5,4,3,2,1]`	reverses elements in a list	`[1,2,3,4,5]`
`take 3 [5,4,3,2,1]`	takes first three elements	`[5,4,3]`
`sum [5,4,3,2,1]`	returns a sum of a list	15
`product [5,4,3,2,1]`	returns a product of a list	120
`elem 5 [5,4,3,2,1]`	return bool element is present in a list or not.	`True`
``5 `elem` [5,4,3,2,1]``	the above in infix.	
`and [True, False, True, True]`	returns True if all values are True otherwise False.	False
`or [True, False, True, True]`	returns True if one value is True otherwise False.	True
`any (==4) [1,2,3,4,5,6]`	cheks if any element is 4.	True
`all (==4) [1,2,3,4,5,6]`	checks if all elements are 4.	False
`splitAt 3 "GCUGCC"`	splits elements at 3rd index.	`("GCU","GCC")`
`partition (>3) [1,3,5,6,3,2,1,0,3,7]`	Partations elements given a condition.	`([5,6,7],[1,3,3,2,1,0,3])`
`take 10 (repeat 5)`	repeats 5 10 times	`[5,5,5,5,5,5,5,5,5,5]`
`cycle [1,2,3]`	repeats 123 infinite times	
`map (+3) [1,2,3,4,5]`	applies a function to all elements in a list	`[4,5,6,7,8]`
`filter (>3) [1,5,3,2,1,6,4,3,2,1]`	filters elements based on a condition.	`[2,4,6,8,10]`
`filter even [1..10]`	Another example of above used with function	`[5,6,4]`

maximum [100,220,33]	returns max num from list	220

Fold

Code: eg1 xs = foldl (-) 0 xs

Function used: eg1 [1,2,3]

Output: -6

What happens:

```
(((0-1)-2)-3)
= ((-1-2)-3)
= (-3-3)
= -6
```

Code: **sum'** = foldl1 (+)

Function used: sum' [1,2]

output: 3

Data.List Functions in Haskell

import Data.List

code	does	output
nub [1,2,3,1,2,3,4,5,3,2,7,4]	removes duplicates	[1,2,3,4,5,7]
4 `elemIndex` [1,2,3,4,5,6]	Tells the index of element in list.	Just 3
10 `elemIndex` [1,2,3,4,5,6]	Same as the above.	Nothing
' ' `elemIndices` "Where are the spaces?"	Gives index of required element.	[5,9,13]
"bat" `isInfixOf` "batman!"	Checks if element is present in a list.	True
find (>4) [1,2,3,4,5,6]	Finds the first elemnt suting a given condition.	Just 5
sort [8,5,3,2,1,6,4,2]	sorts numbers	in acending order
group [1,1,1,1,2,2,2,2,3,3]	groups similar elements together	[[1,1,1,1],[2,2,2,2],[3,3]]
insert 4 [3,5,1,2,8,2]	inserts element where the corresponding element is equal to or greater then selected element.	[3,4,5,1,2,8,2]

Data.Char Functions in Haskell

import Data.Char

code	does	output
isUpper 'A'	checks if a character is upper case	True
isLower 'A'	checks if a character is Lower case	False
isSpace ' '	checks if a character is a space	True
isAlpha 'a'	checks is character is alphabet	True
isLetter 'a'	checks if letter is alphabet	True
isDigit '1'	checks if character is number	True
isAlphaNum '!'	chcecks if character is a number or a alphabet.	False
generalCategory 'a'	Gives category of character	**LowercaseLetter**
toUpper 'a'	converts characters to uppercase	A
toLower 'A'	converts characters to lowercase	a

Tuples in Haskell

Example 1:

Code: t = (1,2,3)

Output: (1,2,3)

Example 2: different data types:

Code: t = (1,"a", True)

Output: (1,"a",True)

Tuples are of fixed length in haskell and cannot be appended to.

Example 3 pairs:

```
Code:   fst ("one", "two")
Output: "one"
Code:   snd ("one", "two")
Output: "two"
```

Dictionaries / Maps

```
Code:    myDict = [ ("one", 1), ("two", 2), ("three", 3) ]
Output:  [("one",1),("two",2),("three",3)]
```

To look at elements in a dictionary:

```
Code:   lookup "one" myDict
Output: Just 1
```

```
Code:   lookup "four" myDict
Output: Nothing
```

Zip function:

```
Code:        zip [1 .. 5] ["one", "two", "three", "four", "five"]
Output:      [(1,"one"),(2,"two"),(3,"three"),(4,"four"),(5,"five")]
```

Takes two lists and zips them together in tuple pairs or creates maps.

Maybe

Maybe data has two outputs

Just
Nothing

Example:

```
Code1:                m1 = Just 4
Code2:                m2 = Nothing
Function applied1:    :t m1
Output1: m1 :         : Num a => Maybe a
Function applied 2:   :t m2
Output2:  :           : Maybe a
```

Extracting data from maybe

```
Library import:   import Data.Maybe

Code:             myDict = [ ("one", 1), ("two", 2), ("three", 3) ]

Using function:   fromJust (lookup "one" myDict) + fromJust (lookup
                  "two" myDict)
Output:               3
```

If statements in Haskell

Code:

```
doubleSmallNumber x = if x > 100
                        then x
                        else 2*x
```

Explanation:

In haskell if statements can only have one else statement to do more then that use nested if statements.

In haskell **else** is mandantory with an **if**.

Could also be used as:

```
doubleSmallNumber' x = (if x > 100 then x else x*2) + 1
```

nested if statements:

```
whatSize x = if x<10 then "small"
             else (if x>100 then "big"
                   else "medium"
                  )
```

Case Expressions

```
digitToWord n = case n of 1 -> "one"
                          2 -> "two"
                          3 -> "three"
```

Pattern Matching

```
describeList xs = "This list is " ++
                    case xs of []    -> "empty."
                               [x]   -> "a singleton."
                               [x,y] -> "a pair."
                               xs    -> "too darn long."
```

In the code above:

Xs: is a traditional variable name for a list in Haskell

X: is used for a singleton

```
myHead someList = case someList of [] -> undefined

                                   (x:xs) -> x (x:xs)
```

Pattern Matching: Function Bodies

```
Example:
sayMe :: (Integral a) => a -> String
sayMe 1 = "One!"
sayMe 2 = "Two!"
sayMe 3 = "Three!"
sayMe x = "Too Big"
```

In the above example:

`sayMe :: (Integral a) => a -> String`

`variable name: sayMe`

`Input given: Integral`

`Output Expected: String`

Example2:

```
myLength :: (Num b) => [a] -> b
myLength [] = 0
myLength (_:xs) = 1 + myLength xs
```

In the code above:

`Xs:` is a traditional variable name for a list in Haskell

`_: is used for an element not required.`

`Example 3`

```
(&&) :: Bool -> Bool -> Bool
True  && True = True
True  && False = False
False && True = False
False && False = False
```

Could simply be written as:

```
(&&) :: Bool -> Bool -> Bool
True && True = True
 _   && _    = False
```

`Meaning everting other then the first statement with && given two bols is False.`

`Example4 using @ symbol`

```
capital :: String -> String
capital "" = "Empty string, whoops!"
capital all@(x:xs) = "The first letter of " ++ all ++ " is " ++ [x]
```

code used: `capital "Haskell"`

`Output: "The first letter of Haskell is H"`

The @ symbol breaks the sequence of a pattern but still has reference to the sequence.

Function Body Guards

Code:

```
degreeClass :: (Num a, Ord a) => a -> String
degreeClass score
    | score >= 70 = "First Class"
    | score >= 60 = "Upper Second"
    | score >= 50 = "Lower Second"
    | score >= 40 = "Third"
    | otherwise = "Fail"
```

Explanation:

```
degreeClass :: (Num a, Ord a) => a -> String
```

input: is type **Num** and is **Ordnial**.

Output should be **String**.

Where Claus

```
classification listOfGrades
    | av >= 70 = "First Class"
    | av >= 60 = "Upper Second"
    | av >= 50 = "Lower Second"
    | av >= 40 = "Third"
    | otherwise = "Fail"
    where
        average xs = (sum xs) / (genericLength xs)
        av = average listOfGrades
```

Let Expression

```
Code: 4 * (let a = 9; b=10; c=11 in a + b + c - 1) + 2
```

```
Output: 118
```

Another example:

```
cylinderSurfaceArea radius height =
    let sideArea = 2 * pi * radius * height
        topArea = pi * radius^2
    in  sideArea + 2 * topArea
```

Introduction to I/O IN Haskell

When writing in a interpreter.

Code: putStrLn "Hello World"

Output: Hello World

But if the code needed to be added to a script, then it should use the **main** keyword and then compile and run the code.

```
main = putStrLn "hello world"
```

Output: Hello World

To show numbers

```
Code:   show 6008
Output: "6008"
```

Numbers are converted to strings and then shown. As seen above.

To show numbers without quotes numbers

```
Code:    print 6008

Output:  6008
```

To show strings in quotes

```
Code:    putStrLn (show "hello")

Output:  "hello"
```

Or by using

```
print "hello"
```

Getting input

```
getLine
```

Making input and output interactive in Haskell

Do-Blocks

```
main = do
         putStrLn "What is your surname?"
         name <- getLine
         putStrLn "What is your title"
         title <- getLine
         putStrLn ("Hello " ++ title ++ " " ++ name ++ ".")
         putStrLn ("Nice to meet you!")
```

Getting pure output from Haskell

```
main = do
  putStrLn "What is your surname?"
  name <- getLine
  putStrLn "What is your title"
  title <- getLine
  let initial = head name
  putStrLn ("Hello " ++ title ++ " " ++ [initial] ++ ".")
  putStrLn ("Nice to meet you!")
```

Converting strings to integers

```
main = do

 putStrLn "How old are you?"

 ageStr <- getLine

 let age = read ageStr :: Integer

 let days = age*365

 putStrLn (show days ++ " days")
```

If statements with do

```
import Data.Char

main = do
   putStrLn "AI: What do you want to say to me?"
   fromUser <- getLine
   let corrected = correct fromUser
   if fromUser==corrected
      then do
          putStrLn "AI: Sorry - I cannot help."
          putStrLn "AI: But nicely written!"
      else do
```

```
            putStrLn "AI: Learn to write properly please."
            putStrLn "AI: You should have typed:"
            putStr ">>> "
            putStrLn corrected

correct :: String -> String
correct sentence
     | endChar `elem` ".?!"  = withCap
     | otherwise = withCap++"."
     where withCap = capitaliseFirst sentence
           n = length sentence
           endChar = sentence!!(n-1)

capitaliseFirst :: String -> String
capitaliseFirst (hd:tl) = (toUpper hd) : tl
```

Return statement

Haskell uses **return stements** to make **pure function impure** by adding **a wrapper** around them.

This can be done to help us **return impure** data types.

These **wrappers** are called **monads**

Example: return ()

Haskell return is the opposite of <-

So if data is obtained using <- it can be inseted using return

See the code below for understanding:

```
act :: IO (String)
act = do
         putStrLn "Title?"
         tl <- getLine
         putStrLn "First Name?"
         fn <- getLine
         putStrLn "Last Name?"
         ln <- getLine
         return (makeAddress tl fn ln)

makeAddress tl fn ln = tl ++ " "  ++ [fn!!0] ++ ". " ++ ln

main = do
    address <- act
    putStrLn "\nThe correct address to use in the letter is..."
    putStrLn address
```

code	does
putStrLn	prints a string to a line.
putStr	prints a string to a line without the new line character added at the end.
putChar	is like `putStr` but for a single character.
print	takes a value of any type that is an instance of
Show	calls show to make it a string and the uses `putStrLn` to send that string to the terminal.
getLine	reads a string from the terminal until a new line character is found.
getChar	reads a single character from the input.

code	does	output
sequence	takes a list of I/O actions and returns an I/O action that will perform those actions one after the other, like `do`	could be anything user enters
mapM	takes a list of I/O actions and returns an I/O action that will perform those actions one after the other, like `do`	1 2 3 [(),(),()]
mapM_	takes a list of I/O actions and returns an I/O action that will perform those actions one after the other, like `do`	1 2 3
getContents	is an I/O action that reads everything from the standard input until it encounters an end-of-file character.	
cat **cat <inputFileName>.txt | runhaskell <codeFileName>.hs**	could be used in a compiler to do the above	
interact main **= interact <functionName>**	takes a function of type `String -> String` and returns an I/O action that will take some input, run that function on it and then print out the function's result.	

Working with files

Getting data from files

```
import System.IO

main = do
    fileHandle <- openFile "<fileName>.txt" ReadMode
    contents <- hGetContents fileHandle
    putStr contents
    hClose fileHandle
```

In the above code the open file action has the following modes:

- ReadMode,
- WriteMode,
- AppendMode,
- ReadWriteMode

Example 2:

```
main = do
    contents <- readFile "<fileName>.txt"
    putStr contents
```

with the above code the following actions can be used.

- `writeFile` – It takes a path and a string and returns an I/O action that will write the string to the file at the path. If the file already exists, is it overwritten.
- `appendFile` is similar but it adds to the end instead of overwriting.

Example of append file:

```
import System.IO

main = do
    todoItem <- getLine
    appendFile "todo.txt" (todoItem ++ "\n")
```

Example 3:

Appending using a command line argument.

```
import System.IO
import System.Environment

main = do
    args <- getArgs
    let fname = head args
    todoItem <- getLine
```

```
    appendFile fname (todoItem ++ "\n")
```

Random numbers will ot be covered in these notes as they are not a part of Haskell standard library and have to be self-generated.