Haskell Tutorial Notes

Alexandru Hambasan
(with some minor edits by Jürgen Schönwälder)

Jacobs University Bremen

September 3, 2019

Haskell - Introduction and Motivation

- Why this programming language?
  - It's fun to write and reason about
  - Will most likely change the way you think about programming
  - It's safe (we will see later what this refers to)
  - Seasoned Haskell users can read it easily
  - It's pure! (we will also see what this means later on)

Haskell Environment

- GHC (Glasgow Haskell Compiler)
  - used for ”real work”
  - supports parallel execution
  - provides performance analysis and debugging tools
  - ghc $\rightarrow$ compiler for generating fast native code
  - ghci $\rightarrow$ interpreter and debugger
  - runghc $\rightarrow$ program for running Haskell programs as scripts

Note: the last three represent the main components of GHC

Lists, Characters and Strings

- Lists
  - they are surrounded by square brackets and elements are separated by commas
  - empty list $\rightarrow$ [ ]
  - all elements must be of the same type
  - Haskell supports enumeration notation
    i.e. [1..10] = [1,2,3,4,5,6,7,8,9,10]
  - concatenate lists using the ++ operator
    e.g.: *[1, 2, 1] ++ [1, 2] = [1, 2, 1, 1, 2]*
  - the cons (short for construct) operator adds an element to the front of a list
    symbol for cons $\rightarrow$:
    e.g.: *1 : [2, 3] = [1, 2, 3]*

Note: only the construction < *element* > : < *list* > is allowed

Lists, Characters and Strings

- Characters
  - Haskell follows the conventions established by C
  - single characters are enclosed in single quotes
  - '\n' → newline character
  - '\t' → tab character
- Strings
  - strings are enclosed in double quotes
  - they are a list of characters

    e.g.:

    ```
    ghci> let x = ['s', 't', 'r', 'i', 'n', 'g']
    ghci> x
    "string"
    ```
  - putStrLn function prints a string

Haskell's Type System

Haskell is a strongly, statically typed programming language where types can be inferred.

- strongly typed
  - Haskell doesn't automatically cast one type from another; in contrast with C
  - the benefit: catches real errors in the code before they cause problems
- statically typed
  - types are known at compilation time by the compiler/interpreter
  - you can do dynamic typing in Haskell, even though is not as easy as in other programming languages (say Ruby)
  - static typing, in combination with strong typing, makes type errors impossible to occur at runtime
- types can be inferred
  - types can (almost always) be deduced, so you don't have to specify them (but you can!)

Basic Types in Haskell

- *Char* $\rightarrow$ character
- *Bool* $\rightarrow$ boolean value
- *Int* $\rightarrow$ on a 32-bit machine, usually it represents a 32 bit signed integer. On a 64-bit machine, it is usually a 64 bit signed integer (and so on)
- *Integer* $\rightarrow$ a signed integer of unbounded size
  - this type is expensive!
  - this type gives more reliably correct answer (because it doesn't silently overflow).
- *Double* $\rightarrow$ used for floating point representation, usually 64 bits wide

Note: *Float* type also exists, but it is much slower

Lists and Tuples

We have already introduced lists in the slide *Lists, Characters and Strings*, but now let us see how we can work with them.

- we will make use of them very often during this course
- note that to apply a function in Haskell, one would write the name of the function, followed by the arguments (no parentheses needed)

```
ghci> odd 5
true
ghci> compare 1 2
LT
```

- function application has higher precedence than the operators, so writing *compare 1 2 == LT* is the same with writing *(compare 1 2) == LT* and both expression will evaluate to *True*

Lists and Tuples

- *head* function returns the first element of a list:

```
ghci> head [1, 2, 3]
1
ghci> head "string"
s
```

- *tail* function returns the list without its first element:

```
ghci> tail [1, 2, 3]
[2, 3]
ghci> tail "string"
tring
```

Please note how a string is just a list of characters

Lists and Tuples

- *length* function returns the number of elements in a list
- *null* function returns true if a list is empty and false otherwise
- *last* function returns the last element of a list
- *init* function returns the list without the last element

  Problem: Call head function on empty list (i.e.: *head []*). What happens? How can we avoid this issue?

- **Naive Idea:** Check if there are elements in the list by writing something like: *length xs* $>$ 0 *then head xs else 'Q'*
  - the length of a list is not stored explicitly somewhere, so the function has to go through the entire list and count the elements $\Rightarrow$ takes more time than necessary
  - Haskell allows us to create infinite lists, so by using length function careless we might end up in an infinite loop

Lists and Tuples

- **Better Approach:** use function null instead
  - constant time
  - makes our code indicate better which property of the list we really care for

  e.g.: *if not (null xs) then head xs else 'Q'*
- *concat* function takes a list of lists and concatenate them, returning one list
- *reverse* function takes a list and reverse all the elements within it
- *and* and *or* functions can be applied to a list of bools

```
ghci> or [False, False, True, False]
True
ghci> and [False, True, True]
False
```

- *all* and *any* functions return True if the predicate (taken as an argument by the function) succeeds on every (or any, respectively) element of the list

Lists and Tuples

- *take* function takes a list as an argument and returns a list containing the first k elements from it
- *drop* function takes a list and returns it without the first k elements
- *splitAt* function splits a list at the given index and returns a pair containing the two sublists
- *elem* function returns True if a given value is in the list and false otherwise

```
ghci> elem 1 [3, 1, 2]
True
```

- *filter* function filter a list.

```
ghci> filter even [1..10]
[2, 4, 6, 8, 10]
```

- *zip* function takes two lists and ”zips” them into a single list of pairs.

```
ghci> zip [1, 2, 3] "test"
[(1,t), (2, e), (3, s)]
```

Lists and Tuples

- A tuple is a fixed size collection of values
  - note that the values in a tuple can have different types
- *fst* function returns the first element of a pair (a 2-tuple)
- *snd* functions returns the second element of a pair

Functions

- Haskell is a functional language $\Rightarrow$ functions play a major role
- defining a function can be as simple as:

```
multiply :: Num a => a -> a -> a
multiply x y = x * y
```

- the first line represents the function signature
  - we will see how to interpret this later when we will be talking about currying, don't panic if you don't understand it now
  - you can think at it as: "As long as type *a* is a numeric type, the function takes two arguments of type *a* and return something of type *a*"
- the second line represents the function definition
  - as we will see, the definitions can become more complex (i.e.: can have *patterns*, *let constructions*, *where constructions*, etc)

Functional Programming in Haskell

Now we will learn how to think from a functional programming point of view and how imperative languages features (say loops) maps to different functional features (recursion, folds, lazy data structures). The following topics will be discussed:

- how to think while doing functional programming
- pattern matching
- lambda functions
- recursion
- lazy evaluation
- call-by-value vs call-by-name vs call-by-need
- using Haskell features instead of loops
- monads

Think before you code

If you have previously worked with C, C++, Java, etc, you might find the functional programming style to be (at least in the beginning), in a sense, weird and difficult to learn. Here are a few tips on how to think while doing fp:

- don't think about a program as a sequence of operations
- think about the relationship between the input and the output
- try to drive simplicity to the maximum.
- think in terms of composition and not in terms of inheritance
- think of side-effects (we will see later on what this refers to)

Pattern Matching

- specifying some patterns to which some data should conform
- check to see if it does
- deconstruct the data according to that pattern
- pattern matching leads to neat code that is easy to read

E.g.:

```
whoAmI :: Int -> String

whoAmI 1 = "One"
whoAmI 2 = "Two"
whoAmI x = "I am not One or Two"
```

- please note that the patterns are checked from top to bottom
- patterns should catch all possibilities, otherwise an exception will be raised for non-exhaustive matching

Pattern Matching

Example: Let us implement functions for extracting the first and second element of a 3-tuple:

```
myFirst :: (a, b, c) -> a
myFirst (x, _, _) = x

mySecond :: (a, b, c) -> b
mySecond (_, y, _) = y
```

- Patterns also work with lists
- the pattern x:xs is used a lot in practice. Here, *x* binds to the head of a list and *xs* binds to the tail of the list.

Example: Here is a function that sums elements from a list:

```
sumElements :: Num t => [t] -> t
sumElements [] = 0
sumElements (x:xs) = x + sumElements xs
```

Guards

- resemble an if statement in C
- test for some condition

Example:

```
whereAmI :: Int -> String
whereAmI age
   | age <= 4 = "You are at home"
   | age <= 15 = "You are at primary school"
   | age <= 19 = "You are at high-school"
   | age <= 23 = "You are at university"
   | otherwise = "You are at work!"
```

Note: *there is no = after function name and its parameters*

Where constructions

- the *where* clause allows us to introduce local variables and functions (another mechanism that allows us to do this are *let expression*, see next slide)

Example:

```
addFourNumbers :: Int -> Int -> Int -> Int -> Int
addFourNumbers x y z k = a + b
                 where a = x + y
                       b = z + k
```

Let bindings

- syntax: let $<$ *bindings* $>$ in $<$ *expression* $>$
- let bindings are local (i.e. they cannot be used across guards)

Example:

```
ghci> let (a, b, c) = (10, 20, 30) in a+b+c
60
```

- note that it is possible to do pattern matching with let bindings (as we did above)

Case expressions

- syntax:

```
case expression of pattern_1 -> result_1
                   pattern_2 -> result_2
                   pattern_3 -> result_3
                   . . .
```

- *expression* is matched against the given patterns

Case Expressions

Example: Let us see in this example how we can use case expressions instead of pattern matching on parameters in function definitions. Note that both samples of code do the same thing!

```
tail' :: [a] -> [a]
tail' [] = error "No tail"
tail' (x:xs) = xs

tail' :: [a] -> [a]
tail' xs = case xs of [] -> error "No tail"
                      (x:xs) -> xs
```

Lambda Functions

- note that Haskell is entirely build on lambda calculus
- also called anonymous functions
- they are helper functions without a name
- syntax:

```
\<argument> -> <function body>
```

- the backslash character is pronounced lambda and it is used because of its visual resemblance with the Greek letter lambda (λ)
- an example of a lambda function:

```
Prelude> (\x -> x^2) 2
4
```

- lambda functions behave (although there are some restrictions on how to define them, see next point) like functions with names
- a lambda function can only have a single clause in its definition

Lambda Functions

- as we cannot have multiple clauses with lambda functions, we must be sure that our pattern covers all cases, otherwise runtime errors will occur

example: Let us implement an unsafe tail function

```
tail' :: [t] -> [t]
tail' = \(_:xs)->xs
```

Please remark that if we would call our tail'function on the empty list, we will get a *non-exhaustive patterns in lambda definition*.

Note: *the ' in the end of a function is read prime and is a common notation for different versions of a function*

- be mindful when you use them!
- we will rarely use lambda functions in this course, instead we will concentrate more on partial function (see next slide)

Partial Functions and Currying

$\boxed{\textit{In Haskell, all functions take only one argument}}$

Question: How can this statement be valid? We have seen that some functions (like *take* function) take 2 arguments.
Answer: In order to understand how the functions like *take* actually behave, we have to understand the concept of currying.

Let's have a look at *take* type signature:

```
take :: Int -> [a] -> [a]
```

- notice that *take* has in its signature only one parameter (i.e. an Int)
- after the Int argument is applied, a function is returned that takes a list as the argument.

Partial Functions and Currying

Let us implement a function that returns the first 3 elements of a list:

```
take3' :: [a] -> [a]
take3' = take 3

Prelude> take3' [1..]
[1,2,3]
```

- notice that *take3'* $< list >$ and *take 3* $< list >$ mean the same thing and return the same result

Partial Functions and Currying

From a mathematical point of view, currying a function expression means successively splitting away arguments from the right to the left. I.e.:
$f(x_1, x_2) = f'(x_1)(x_2)$

- here, f' takes one argument (i.e.: x_1) and returns a function that takes as argument x_2

In the general case of currying an m -ary function expression:
$f(x_1, x_2, ..x_m) = f^{m-1}(x_1)(x_2)..(x_m)$

This is exactly what Haskell does!

Map Function

- *map* takes a function and applies it to every element of a list
- because it takes a function as its argument, we refer to it as a higher-order function

Example:

```
Prelude> map (\x->x*x) [1..10]
[1,4,9,16,25,36,49,64,81,100]
```

Let us see how we can implement our own version of *map*:

```
myMap f [] = []
myMap f (x:xs) = f x : myMap f xs
```

Left Fold Function

- the *foldl* function takes
  - a "step" function
  - an initial value for its accumulator
  - a list
- what *foldl* is doing is call the step function on the current accumulator and an element of the list, then the new accumulator value is passed to itself recursively to consume the rest of the list.

Example: Let us implement our *sum* function (see slides about recursion) using foldl

```
foldSum xs = foldl step_funct 0 xs
    where step_funct accumulator x = accumulator + x
```

When using foldl, you should think about:

- what should be the initial value of the accumulator
  - in this case, it was 0
- how to update the accumulator
  - in this case we used the function *step-funct*

Right Fold Function

- called *foldr*
- same as foldl, but instead folding from left, it folds from the right of a list
- in practice one would nearly always use *foldl'* instead of *foldl* in order to avoid memory leaks caused by the non-strict evaluation that Haskell provides.

Defining New Datatypes

Sometimes we might want to construct new data types for specific purposes.
Example: a datatype that represents the basic characteristics of a person

```
data PersonInfo = Person Int Double String [String]
                  deriving (Show)
```

- to define a new datatype, use the keyword *data*
- PersonInfo is the name of our new type (it is called a **type constructor**)
- after the type is defined we use its type constructor to refer it
- Person is the **data constructor**
  - it's used to create a value of the PersonInfo type

Note: *Both the type constructor and the data constructor must start with a capital letter*

Defining New Datatypes

- the Int, Double, String and [String] (that follows after Person) are the **components/fields** of the type
  - if you are familiar with an object-oriented language, these components serve the same purpose as fields for a class
- in this case the Int represents the age of a person, the Double represents the height, the String field is the name of the person and [String] are the names of his/her parents
- the meaning of *deriving (Show)* will be explained later, so for now just remember that you need to write it into a type declaration in order to be able to print the value within ghci

Defining New Datatypes

- to create a new value of type PersonInfo, we treat the data constructor as a function.

Example: Let us create a new value of type PersonInfo that represents a 49 years old person whose name is Eliot, his parents are Alice and Bob and is 181.49 meters high.

```
newPerson = Person 49 181.49 "Eliot" ["Alice", "Bob"]
```

- in this particular case, the name of the type constructor (PersonInfo) and the value/data constructor (Person) are different only for you to see which is which.
- it is common practice to give them both the same name
- there is no ambiguity because:
  - the type constructor's name is used only in type declaration/ type signature
  - the value constructor's name is used in the actual code (in writing expressions)

Defining New Datatypes

- it's possible to introduce *synonyms* for an existing type at any time
- it might give the type a more descriptive name

Example: Consider the fields from the example above. It is not very clear what the Int, Double, String and [String] mean, therefore we should try to make things as clear as possible.

```
type Age = Int
type Height = Double
type Name = String
type ParentsNames = [String]

data PersonInfo = Person Age Height Name ParentsName
                  deriving (Show)
```

Algebraic Datatypes

- an example of an algebraic datatype is the Bool

```
data Bool = True | False
```

- notice that the Bool datatype has 2 value constructors, *True* and *False*, separated by a | (this symbol can be read as *or*)
- when a type has more than 1 value constructor, these are referred as alternatives/cases.
- the value constructors of an algebraic datatype can take zero or more arguments.

Algebraic Datatypes

Example: Consider (and please excuse the macabreness) the following case, where we can have an alive person (then we need to store all of his/her properties), a dying person (we only need his/her age) or a dead person (we need to know nothing):

```
type Age = Int
type Height = Double
type Name = String
type ParentsName = [String]

data PersonInfo = AlivePerson Age Height Name
                | DyingPerson Age
                | DeadPerson
                  deriving (Show)
```

Parameterized Types

- we have seen that lists have a polymorphic type (i.e.: the elements of a list can be of any type)
- to achieve this when we construct our own datatypes, we make use of a type called *Maybe*

```
data Maybe a = Just a
             | Nothing
```

- in this case, *a* represents a type variable!
- therefore, *Maybe* takes another type as its parameter
- you can create specific types such as *Maybe [Char]*, *Maybe Int*, etc..

Typeclasses

- a very powerful feature of Haskell
- they are not a type themselves, but rather describe a set of types that have a common interface
- if you are familiar with an OOP language, type classes are for types what classes are for objects
- the methods in a typeclass correspond to virtual functions in C++, meaning each instance of the type class might have its own implementation of that method

In the following example, we will try to implement our own equality test function (i.e. our own version of ==).

```
class BasicEq a where
    isEqual :: a -> a -> Bool
```

Note: *the example is taken from a Real World Haskell book*

Typeclasses

- **class BasicEq a where** simply means we define a typeclass called *BasicEq* where *a* is the type variable.
- on the second line, we declared the function
  - it is not mandatory to also define the functions, but you can do so
  - you can declare/define several functions

Now, let us look at the type of our function. To do so, we load the code in ghci and check for its type.

- to load a file in ghci, open ghci and type *:load theFileName*
- to check for a type of a function of variable, type *:type functionName*

```
*Main> :type isEqual
isEqual :: BasicEq a => a -> a -> Bool
```

- everything that is before the symbol $\Rightarrow$ is called a *type constraint*
- you can read it as: "As long as type *a* is an instance of *BasicEq*, the function takes two arguments of the same type and return a Bool"

Typeclasses

Let's define a new data type and then see how we can make it an instance of *BasicEq* type class.

```
data Color = White | Black
            deriving (Show)

instance BasicEq Color where
  isEqual White White = True
  isEqual Black Black = True
  isEqual _ _ = False
```

- since we did not provide a default implementation (i.e. defining the function within the typeclass) for the function isEqual, we had to write one when we declared type Color as being an instance of typeclass BasicEq
- the advantage of having a typeclass is that we can, for example, use our *isEqual* function on any other type we declare to be an instance of *BasicEq* typeclass

Show and Read

- both *Show* and *Read* are built-in typeclasses

Show

- this typeclass is used to convert values of another type (usually the numbers) to their String representation
- most important function in the *Show* typeclass is the *show* function.
- examples of the *show* function:

```
Prelude> show True
"True"
Prelude> show 10
"10"
Prelude> show [(1,2), (1,5), (2,1)]
"[(1,2),(1,5),(2,1)]"
```

- you can easily define a *Show* instance for your type:

```
instance Show Color where
  show Black = "Black"
  show White = "White"
```

Show and Read

- you can also derive an instance for your type by using *deriving (Show)* as we had in the first example
  - Haskell supports automatic derivation for the following typeclasses: *Show*, *Read*, *Bounded*, *Enum*, *Eq*, *Ord*
  - automatic derivation is not always possible even for these typeclasses

Read

- most important function in the *Read* typeclass is the *read* function
- takes a string, parses it and returns the data of any type that is a member of *Read*
- some examples of using *read*

```
Prelude> (read "12")::Int
12
Prelude> (read "12")::Double
12.0
Prelude> (read "[1,2,3]")::[Int]
[1,2,3]
```

Recursion

- a program technique in which a function calls itself
- one of the most powerful techniques in functional programming
- characteristics:
  - Base (or Terminating) Step: a version of the problem that is simple enough that the routine can solve it and simply return a value, without calling itself again
  - Recursive Step: the function calls itself and while doing so, it solves a smaller problem
- it is the programming equivalent of mathematical induction

Example: Let us look at a simple Haskell implementation of a recursive function that computes the sum of the first n numbers.

```
sum :: Int -> Int
sum 0 = 0
sum n = n + sum(n-1)
```

Tail Recursion

In functional programming (so also in Haskell) recursion replaces the well-known *for* and *while* loops that imperative languages have.

Problem: In contrast with the loops from an imperative language, a recursive function would require linear space instead of constant space. The reason behind this is that a recursive function allocates some space each time it applies itself, in order to know where to return.

Idea: Use tail recursion!

Question: What is tail recursion?

- a recursive function where the final result of the function itself is final result of the recursive call
- Haskell detects tail recursive calls and makes them run in constant space (TCO)

Note: *TCO stands for Tail Call Optimization*

Tail Recursion

Let us implement the above recursive function as a tail recursive function.

```
sum2 :: Int -> Int
sum2 x =
  tailSum x 0
  where tailSum 0 a = a
        tailSum n a = tailSum(n-1)(a+n)
```

- Note that sum2 wraps a call to the function tailSum that is tail recursive.
- The tailSum function has another parameter *a* that is called an accumulator and which holds the values from previous calls.

Mutual Recursion

- often useful to define functions simultaneously, each of them calling the other (and maybe itself) in order to compute the result
- such functions → mutually recursive

Example: Let us implement a function for testing whether a (positive) number is odd or even. Usually, in practice, one would test if the number is congruent to 0 mod 2, but for the sake of example, we will not do that here.

```
odd :: Int → Bool
odd 0 = False
odd n = even (n-1)

even :: Int → Bool
even 0 = True
even n = odd (n-1)
```

Lazy Evaluation

- expressions are evaluated only at their (first) use
- it's a trade-off
  - the code is more modular
  - it can become confusing how evaluation proceeds in the program
- in order to fully understand how lazy-evaluation works in Haskell, the following sub-topics have to be understood:
  - graph reduction
  - normal form and weak head normal form
  - evaluation order
  - space and time usage using lazy-evaluation

Note: *the subtopics above will be discussed on the whiteboard*

Call-by-Value vs Call-by-Name vs Call-by-Need

The relationship between call-by-value and call-by-name can be explained by the following theorems

- **Church Rosser (1):** For a purely functional language, if call-by-value evaluation and call-by-name evaluation both yield a well-defined result then they yield the same result.
- **Church Rosser (2):** If a well-defined result exists for an expression then the call-by-name evaluation strategy will find it where, in some cases, call-by-value evaluation will not.

- Haskell uses call-by-need by default. Arguments are not evaluated when the function is called, but when the arguments are needed. These evaluations are also stored, such that the next time they are required, the cached values can be looked up.

Call-by-Value vs Call-by-Name vs Call-by-Need

- in contrast, functions that use call-by-name substitute the arguments in the function body in an unevaluated form $\Rightarrow$ the same expression might end up being evaluated multiple times
- functions that use call-by-value form of evaluation, evaluate the arguments once the function is called.

Please note:

- some languages that are eager evaluated (meaning they use call-by-value form of evaluation, C for example) use short-circuit evaluation in some cases.

  Take for example the short-circuit operator && and the boolean C expression *(A && B)*. B will be evaluated, only if the first argument is not enough to determine the overall boolean value (i.e.: if A is true). This resembles, in a sense, the call-by-need form of evaluation.
- Haskell can be forced to do call-by-value evaluation.

Side effects

- a side effect allows the behavior of a function to change the global state of a system
- changing the global state of a system might mean:
  - printing something on the terminal
  - writing data on the disk
  - changing the value of a global variable
- if a function's output depends entirely on the input we explicitly provide, we say that the function is **pure**
  - this means that the function not only has to have no side-effects, but has to also be idempotent
- in Haskell, we can only look at a type signature of a function and see if it is pure or impure
  - an impure function will have the type of the result starting with an IO

```
putStrLn :: String -> IO ()
```

Pure vs. Impure

Question: Should you choose to write code in a pure language or an impure one?

Answer: It very much depends on what you are trying to achieve and what context you are in.

- impure functional programming offers a set of features such as exceptions, state and continuations
- in pure functional programming it is easier to reason about the correctness of your code and you have the benefit of lazy-evaluation
- don't forget that Haskell has deep roots into mathematics and sometimes when writing code you will be put in the position to think from a mathematical point of view
- if you want to modify your program that is written in a pure functional programming, you might need to restructure much of your code, where in a impure functional programming you could ease the change by using its features

Exceptions

On the last slide, we mentioned exceptions as being part of the features an impure programming language provides.

Question: What are exceptions anyway?

Answer: Consider the mathematical case of *1 / 0*.

- you want a way to say: *Hey, this shouldn't have happened*
  - in the context of programming, this means throwing an exception
- you want a way to handle the case, now that has happened
  - this means handling an exception

Exceptions

Are exceptions pure or impure? Do the output of the functions that use them depend only on the explicit input?

- throwing an exception is pure
  - because throwing the exception doesn't make the output of the function dependent of something else $\Rightarrow$ the function is still pure
- catching an exception is impure
  - consider the case where we would return 1 if the computations within the functions succeed and 0 if they don't. Notice that in this case, the function's output depends on outside circumstances (i.e.: not only on the input) such as memory runout, the other programs that are running on the machine, etc

Note: Because of the time and space constraints, I have tried to keep the explanations as brief as possible, but there is more to argue about this!

Monads

The topic of *monads* is sometimes deemed to be hard to understand because it is a very abstract concept.

Don't worry if you don't fully understand the following slides, but it is important to get an overall idea!

Motivation: Why do we need monads?

- we would like to program only using functions
  - but how can we establish the function executing order?
- how to we deal with functions that fail? exceptions are not functions and we've seen that handling an exception is not pure
- code reuse
- eliminate boilerplate (same section of code included in more places within the program) code

IO Monad

Programs process the data from the outside and return the results back to the outside world ⇒ the I/O system is very important for a program.

Consider the following program:

```
main = do
       putStrLn "Please enter your name"
       name <- getLine
       putStrLn $ "Hello " ++ name ++ "!"
```

- putStrLn is used for printing a string followed by a *newline character*
- getLine reads a line from the standard input
- the operator

  ```
  <-
  ```

  binds the value from an **IO action** to a name

IO Monad

Let us look at the signatures of putStrLn and and getLine

```
putStrLn :: String -> IO()

getLine :: IO String
```

What is IO() meaning? What about IO String?

- anything of the type *IO t* is called an IO action
  - note that *t* can be any type
- the *()* means an empty tuple and in this case it indicates that the function putStrLn has no return value
  - it is pronounced *unit*

Actions

What are actions?

- anything of type IO is an action
- an action is something we do that modifies the state of the real world
  - print something to the terminal
  - read input from the console
  - establish a network connection
  - many others
- an action can be created, assigned and passed anywhere, like numbers, strings or functions
- they produce an effect when performed, but not when evaluated
- by executing an action of type *IO t* we will perform an I/O and get a result of type *t*
- you can perform an I/O action only from within other I/O actions
- *main* is an I/O action with type *IO ()*

IO Monad

- the execution of a Haskell program begins with *main*
- all I/O in Haskell programs is driven from the top at *main*
- this mechanism allows to isolate side effects
  - perform I/O (which is not pure) within IO actions
  - call pure functions from there
- *do* is how you define a sequence of actions
  - indentation matters!

Why is it important to isolate impure code?

- unanticipated side-effects and the misunderstanding of the circumstances in which functions might return different results for the same input cause many errors in programs
- isolating impure code (i.e. code that has side effects) allows the programmers to know which part of the code might alter the state of the system or the world
- because of this, some compilers might be able to provide a level of automatic parallelism (don't worry if you don't know what this is)

I/O - Files

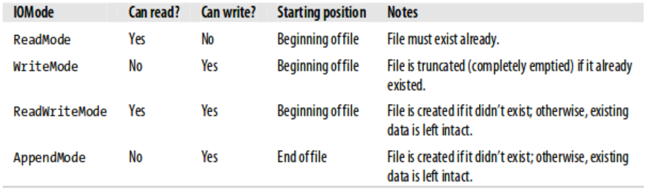
```
import System.IO
import Data.Char(isLetter)

main :: IO ()
main = do
       inputHandle <- openFile "input.txt" ReadMode
       outputHandle <- openFile "output.txt" WriteMode
       main_loop inputHandle outputHandle
       hClose inputHandle
       hClose outputHandle

main_loop :: Handle -> Handle -> IO()
main_loop inH outH =
  do inEOF <- hIsEOF inH
     if inEOF
        then return ()
        else do inCh <- hGetChar inH
                if (isLetter inCh)
                   then do hPutChar outH inCh
                           main_loop inH outH
                   else main_loop inH outH
```

I/O - Files

- the code above represents a simple program that reads the characters from a file and prints all characters that are letters in another file
- the program begins by using openFile which returns a file *handle*
- in the end of the program hClose is used in order to close the *handle*
- you need to import the System.IO module when working with files because these functions (openFile, hClose, etc..) are defined there
- in the *main loop* function we check if we reached the end of the file, if not we proceed with reading a character and checking if it is a letter (for that we need *isLetter* function defined in Data.Char module). If it is, we print it in the output file. We continue recursively to process the file.

I/O - Files

- openFile function takes as arguments a String (i.e. the file name) and a *IOMode* (it specifies how the file will be managed)
- the following table (taken from *Real World Haskell* book) shows the possible values of a IOMode



IOMode	Can read?	Can write?	Starting position	Notes
`ReadMode`	Yes	No	Beginning of file	File must exist already.
`WriteMode`	No	Yes	Beginning of file	File is truncated (completely emptied) if it already existed.
`ReadWriteMode`	Yes	Yes	Beginning of file	File is created if it didn't exist; otherwise, existing data is left intact.
`AppendMode`	No	Yes	End of file	File is created if it didn't exist; otherwise, existing data is left intact.

- if you want to open a binary file, use *openBinaryFile* function instead

I/O - Files

- to close file handles use *hClose* function
- **do not forget** to close the opened files!
- when you read and write to a file, a record of the current position is maintained and once you do a reading this position is incremented
- hTell function returns the current position
- hSeek function allows you to change the current position

Note: *we will not discuss hSeek and hTell as we will not make use of them*

- if you will ever need Haskell for real world projects, you should know that there is also a Lazy I/O approach which you might want to read about

Monads

On previous slides we discussed the IO monad, but we focused more on the I/O actions and how to perform them, we haven't really defined what monads are. We will firstly have a look on what a monad is from a mathematical perspective.

- a monad is a triple: *(M, unitM, bindM)*
  - M is a *type constructor* (see the slides where we introduced new data types)
  - unitM and bindM are polymorphic functions with the following signatures:

```
unitM :: a -> M a
bindM :: M a -> (a -> M b) -> M b
```

  - unitM takes a value of type *a* into its corresponding monadic representation
- `bindM` function simply makes the result from the previous computation to become the argument of the next one; it is a way to sequence actions and its correspondent in Haskell is the operator $>>=$ (we will talk later about sequencing in Haskell)

Monads

- the triple has to satisfy 3 laws:
  - left unit
  - right unit
  - associativity

Note: As I believe a further discussion about monads from a mathematical point of view would be beyond the scope of this tutorial, I would encourage you (in case you want to want to know more on this) to read *The essence of functional programming* which is a very fine paper on monads written by the one who introduced them to Haskell.

Monads

Before explaining what is a monad, let us simply have a look on the monads we have seen so far (maybe, until now, you didn't know they are monads):

- lists
- Maybe
- IO monad

The monads are simply type classes that supports **at least**:

- a binding operation (represented by $>>=$)
- a wrapping operation (represented by *return()*)

If we look into $>>=$ signature:

```
(>>=) :: Monad m => m a -> (a -> m b) -> m b
```

- *m a* is called a monadic value; *a* is the old type (i.e.: an Int, a Bool, etc) wrapped by the monad *m*
  - it is exactly the meaning of IO Int, IO Bool, etc..
  - people usually see the monads as containers, boxes

Sequencing

When we talked about *do* on slide 59, we said that it is just a way to sequence actions. We can also chain actions by using the operators $>>$ and $>>=$.

- $>>$ operator sequences two actions together: the first action is performed, its result is discarded and then the second action is performed
- $>>=$ operator sequences two actions together: the first action is performed, its result becomes the argument for a second function that returns an action

Consider the program given as an example on slide 56, let us try to write it without using *do*:

```
main =
    putStrLn "Please enter your name" >>
    getLine >>=
    (\name -> putStrLn ( "Hello " + name + "!"))
```

Return()

- in Haskell, *return* does not have the same meaning as in other languages you might be familiar with
- *return* is used in Haskell to wrap data in a monad
- it is the equivalent of the `unitM` function that we saw earlier
- it does not abort the execution and can be used anywhere in the program
- it is, in a sense, the opposite of the operator $<-$ which is used to pull out the data from a monad

If you write a function whose result depends on the I/O, you must wrap it in a IO monad.

Example

```
isRaining :: IO Bool
isRaining =
     do putStrLn "Is it raining outside? (y/n)"
        input <- getChar
        return (input == 'y')
```

References

- *Real World Haskell* by Bryan O'Sullivan, Donald Stewart, John Goerzen
- *The essence of functional programming* by Philip Wadler