Advanced Topics in Programming Languages
Lecture 2 - Introduction to Haskell

Ori Bar El
Maxim Finkel

01/11/17

1 History

Haskell is a lazy, committee designed, pure functional programming language that started development in the late 1980's. Haskell derives its name from Haskell Curry, a mathematician and logician. The goals of the Haskell language were formally defined by its designers, and they stated that Haskell should:

1. Be suitable for teaching, research, and applications, including building
2. Be completely described via the publication of a formal syntax and semantics.
3. Be freely available.
4. Be usable as a basis for further language research.
5. Be based on ideas that enjoy a wide consensus.
6. Reduce unnecessary diversity in functional programming languages.

Development of the language began shortly after the 1988 meeting, but it was not until April 1, 1990 that the first working version of Haskell, Haskell 1.0, was released. From 1990 to 1999 many incremental versions of Haskell were released and in 1999 the Haskell Committee decided that a formalized and stable version of the language needed to be released. They decided to call this standard version of the language Haskell 98. The reasoning behind creating a Haskell standard was so the language could be used in teaching environments and books could be written about it. Haskell 98 did not fully stabilize until 2002.

Haskell recently released a new standardized version called Haskell 2010. This update includes all the changes in the language since 2002 repackaged with up-to-date documentation in the newest version of the Haskell Report. Since the release of Haskell 2010, version numbers are now based on date of release [3].

2 Haskell Interpreter

Haskell is an interpreted language. Two common Haskell interpreters are the *HUGS* interpreter and the *GHC* (Glasgow Haskell Compiler) interpreter. The interpreter loads with a module called Prelude already loaded. Prelude contains a large selection of common functions for list comprehension, common numeric operations and basic I/O. The interpreter has a set of build-in commands for loading external Haskell files and various other operations. The interpreter behaves in a similar fashion to the interpreter in Python. Functions can be declared and evaluated completely from inside the interpreter. This makes it the best place to test small portions of code.

An external Haskell file is called a Haskell script and is denoted by the extension *.hs*. To import other modules or scripts into a Haskell script, Haskell uses the import statement followed by the name of the module to be imported. To load a file into the interpreter, use the : *load* command followed by the path to the script that needs to be opened [3].

3 Haskell Features

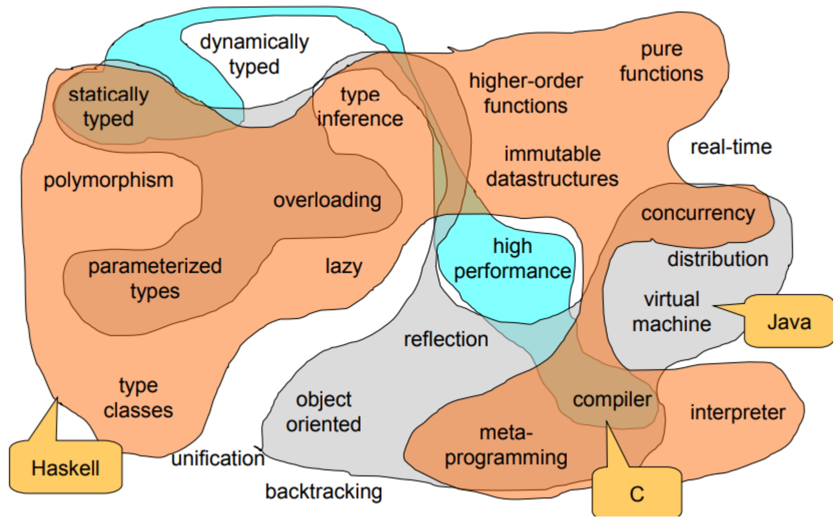

Figure 1: Features of different mainstream programming languages

[5]

All the examples in this chapter can be saved into a Haskell source file and then evaluated by loading that file into GHC. Do not include the "Prelude>" prompts part of any example. When that prompt is shown, it means you can type the following code into an environment like GHCi. Otherwise, you should put the code in a file and run it. [7]

3.1 Types

In Haskell, types are collections of related values. Expression's types are automatically calculated at compile time. This is known as type inference. Because the type of an expression is determined at compile time, errors in type are found at compile time. Because the type of every expression is known at compile time, the code is considered to be safer. The application will not compile if type errors exist. From this we can conclude that Haskell is statically typed. All types in Haskell begin with a capital letter.

In order to view the types of the expressions in the *GHC* interpreter, we will enter the following line:

```
Prelude> :set +t
```

3.1.1 Data Types

Haskell has several basic types. The Bool type consists of the logical values True and False:

```
Prelude> True
True
it :: Bool
```

The Char type is the single character type:

```
Prelude> 'F'
'F'
it :: Char
```

The String type is strings of characters which is basically just a shortcut or alias for lists of Char types:

```
Prelude> "APL17"
"APL17"
it :: [Char]
```

The Int type is a fixed precision integer type, the Integer type is integers of arbitrary precision, the Float type which is real floating point numbers with single precision, the Double type is real floating point numbers with double precision, and the Fractional type is real nembers that can be represented as fractions:

```
Prelude> 42
42
it :: Num a => a
Prelude> 3.14
3.14
it :: Fractional a => a
Prelude> pi
3.141592653589793
it :: Floating a => a
```

Lists in Haskell are homogeneous data structures, or sequences of values which all have the same type. The length of a list is arbitrary, meaning it is not known or defined explicitly. List elements are contained inside of brackets with each element separated by a comma. Each element of a list must be of the same type, lists with elements that are not of the same type will result in a type error. List indexes start at zero. There are a plethora of built in functions for list concatenation and manipulation built into the Haskell language. Because of the power of lists, most applications in Haskell use Lists extensively. The following is an example of a list declaration:

```
Prelude> let list = [1,2,3,4]
list :: Num a => [a]
```

Tuples differ from lists in that they may contain elements of multiple types, but the length of a tuple is determined by its type declaration. Tuple elements are contained in parenthesis with each element separated by a comma. The important differences between lists and tuples is that lists can shrink and grow, but tuples remain the same length. The empty tuple () can only have the value (). Lists and tuples have a unique interaction as you can have lists of tuples and tuples with lists inside of them. By combining the two powerful data types, Haskell can support numerous complex data types. The following are examples of a tuples:

```
Prelude> let tup1 = ()
tup1 :: ()
Prelude> let tup2 = (1,2,3)
tup2 :: (Num c, Num b, Num a) => (a, b, c)
Prelude> let tup3 = (12,3.56,'g',"HELLO",[1,4,7])
tup3 :: (Num a2, Num a1, Fractional b) => (a1, b, Char, [Char], [a2])
```

Functions are defined as mappings of values of a given type to values of another type. These types may be the same. While functions are stand alone entities, they are also types. This is important because in Haskell, functions can be passed as parameters to other functions and also be returned as results. In Haskell, it is perfectly acceptable to have a function that accepts a function as a parameter and returns a function as its return type.

```
Prelude> isPostive x = x > 0
isPostive :: (Num a, Ord a) => a -> Bool
Prelude> factorial n = if n < 2 then 1 else n * factorial (n-1)
factorial :: (Num p, Ord p) => p -> p
```

[3].

3.1.2 Typeclasses

Typeclasses are groupings of types. Because a type belongs to a typeclass, it must support and must implement all the characteristics of that typeclass, similar to how a class in Java must implement all the methods of an interface it uses. Typeclass constraints are denoted with the '=>' symbol. Everything before the '=>' is considered to be a class constraint. Consider the declaration of a plus function "+ :: (Num a) => a -> a -> a", the parameters for this function must be of numeric type. Because Floats, Doubles, Ints, and Integers are all governed by the same typeclass Num, a function that uses the Num typeclass is valid for any numeric data type. This feature allows for polymorphic functions to be created easily in Haskell.

```
Prelude> :i (+)
class Num a where
  (+) :: a -> a -> a
  ...
          -- Defined in ‘GHC.Num’
```

In the previous paragraph, Num was associated with the type variable "a". A type variable is a placeholder variable that begins with a lowercase character. Type variables are not true types, but are instead used to represent parameters with no type attached to them or in conjunction with a typeclass. In this way, they are used for generic parameters for functions, because they are not bound to a specific type or type class without explicitly being assigned to one. Haskell uses the capitalization of the variable to know that Int represents the type Int, and int is just a type variable. Because type variables are used in parameter declaration, they are most commonly simple, one character variables. [3].

3.1.3 Statically Typed Language

A language is *statically typed* if the type of a variable is known at compile time. For some languages this means that you as the programmer must specify what type each variable is (e.g.: Java, C, C++); other languages offer some form of type inference, the capability of the type system to deduce the type of a variable (e.g.: OCaml, Haskell, Scala, Kotlin). The main advantage here is that all kinds of checking can be done by the compiler, and therefore a lot of trivial bugs are caught at a very early stage.

A language is *dynamically typed* if the type is associated with run-time values, and not named variables/fields/etc. This means that you as a programmer can write a little quicker because you do not have to specify types every time (unless using a statically-typed language with type inference). Example: Perl, Ruby, Python. [4]

3.1.4 Algebraic Data Type Declarations

In Haskell, the programmer has the flexibility to create and define new data types and typeclasses. To create a new type, the data keyword is used, followed by the name of the new type, followed by the definition for each type separated by the "|" character. The name for the new type must be capitalized. The following is a definition for a calendar event:

```
Prelude> data Event = Int Int Int
data Event = ...
Prelude> data Event = Date (Int,Int,Int)
data Event = ...
```

In this type declaration, the Event data type contains three separate integers, one for the month, day, and year [3]. We can also define types recursively [5]:

```
Prelude> data Tree = Leaf Int | Node (Int, Tree, Tree)
data Tree = ...
Prelude> let tree = Node(4, Node(3, Leaf 1, Leaf 2),Node(5, Leaf 6, Leaf 7))
tree :: Tree
```

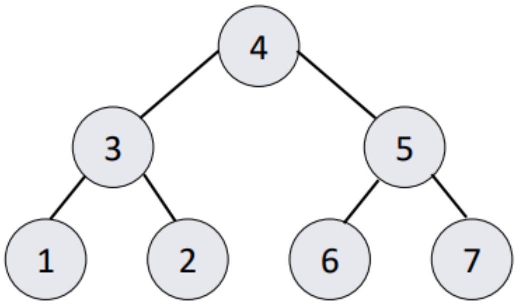

Figure 2: The tree instance that is defined above

The reason these data types are called *algebraic* is because they are created by "algebraic" operations. The "algebra" here is "sums" and "products":

- "sum" is alternation (A | B, meaning A or B but not both).
- "product" is combination (A B, meaning A and B together).

3.1.5 Polymorphism

Polymorphic functions are functions whose type contains one or more different type variables. So any function that implements a typeclass is a polymorphic function and any function that implements a type variable is a polymorphic function. Most built in functions in Haskell are polymorphic functions. Another form of polymorphism is called parametric polymorphism. Polymorphic functions in Haskell do not care about the type of its parameters, it simply treats that type as a completely abstract type. For instance, lets define the following length function that works on lists of arbitrary type:

```
Prelude> length [] = 0; length (x:xs) = 1 + length xs
length :: Num p => [a] -> p
Prelude> length [1,2,3]
```

```
3
it :: Num p => p
Prelude> length ["1","2","3"]
3
it :: Num p => p
```

In addition to parametric polymorphism, Haskell also supports the concept of overloading. In parametric polymorphism, the type of the parameters does not matter, but there are some instances where the type should matter, but still be generic enough to accept multiple parameters types while still being very specific about the exact operation to be performed with each specific type. For example, the equality operator (==) should be able to compare two lists to verify that each element of the list is equal to its corresponding element in another list. However, a list of Integers and a list of Char are very different and the same comparison methods cannot be used for each. Haskell will allow us to overload the equality operator for each instance needed. Haskell will also insure that we cannot compare lists of two different types unless we provide an overloaded function to do so [3].

3.1.6 Type Inference

Haskell's compiler automatically infers types of object. This capability may be used detect bugs. Consider, for example, the function reverse and its most general type:

```
reverseList [] = []
reverseList (x:xs) = reverseList xs
Prelude>   :t (reverseList)
(reverseList) :: [t1] -> [t] -- that doesn't seem right
```

Only by examining the type inference mechanism the programmer can conclude that there is a bug in the function reverseList, since the type of the elements in the reversed list are different from those in the original list. Here is the fixed code:

```
reverseList [] = []
reverseList (x:xs) = reverseList xs ++ [x]
Prelude>   :t (reverseList)
(reverseList) :: [t] -> [t] -- :)
```

3.2 Functional Programming Language

Haskell is a *functional programming language*, meaning a one in which functions are *first-class objects*. In other words, you can manipulate functions with exactly the same ease with which you can manipulate all other *first-class objects* [2]. This means the language supports passing functions as arguments to other functions, returning them as the values from other functions, and assigning them to variables or storing them in data structures[1].

3.2.1 Higher-Order Functions

A *higher-order function* is a function that does at least one of the following:

1. takes one or more functions as arguments
2. returns a function as its result

In the untyped lambda calculus, all functions are higher-order; in a typed lambda calculus, from which most functional programming languages are derived, higher-order functions that take one function as argument are values with types of the form $(\mathcal{T}_1 \to \mathcal{T}_2) \to \mathcal{T}_3$ [8].

For example:

```
twice :: (a -> a) -> (a -> a)
twice f = f . f -- (the . operator stands for function composition)

f :: Num a => a -> a
f = subtract 3

main :: IO ()
main = print (twice f 7) -- 1
```

Or more quickly:

```
twice f = f . f
main = print (twice (+3) 7)  -- 13
```

In addition, mathematical operators (e.g. "+" or "*") are defined using currying instead of as a function of two numbers.

```
Prelude> :i (+)
class Num a where
  (+) :: a -> a -> a
  ...
          -- Defined in 'GHC.Num'
```

Lambda Expressions in Haskell, functions can be created without assigning them a name or explicitly declaring their type by using lambda expressions. The use of lambda expressions comes from lambda calculus. Lambda expressions are useful for defining and evaluating expressions inside of expressions and used to avoid the necessity of giving single use functions a name. The "\" symbol replaces the lambda symbol in the declaration of a lambda function. Lambda expressions are also useful for functions that return functions as their result. For example, we can use a lambda expression in the function that returns a list of the first n odd numbers:

```
   odds :: Int -> [Int]
   odds n =  map (\x -> x*2 + 1) [0..n-1]
   Prelude> odds 5
=> [1,3,5,7,9]
```

3.3 Pure Functions

A language is called *pure* if you can't do anything that has a side effect.

A *side effect* would mean that evaluating an expression changes some internal state that would later cause evaluating the same expression to have a different result. For example, if a function mutated its arguments, set a variable somewhere, or changed behavior based on something other than its input alone, then that function call is not pure. In a pure functional language you can evaluate the same expression as often as you want with the same arguments, and it would always return the same value (i.e. it is idempotent), because there is no state to change. For example, a pure functional language cannot have an assignment operator or do input/output, although for practical purposes, even pure functional languages often call impure libraries to do I/O.

Haskell is a pure language with no side effects other than in I/O.

3.4 Pattern Matching

A powerful tool in Haskell for writing functions is pattern matching. In pattern matching, a sequence of expressions is used to select between a sequence of results of the same type. Pattern matching is not limited to functions of a single argument. In the case of multiple arguments, the patterns are matched in order from left to right for each argument. The underscore character is used to denote a "wildcard", or always True, pattern.

For example, consider this definition of map:

```
Prelude> map _ [] = []; map f (x:xs) = f x : map f xs
map :: (t -> a) -> [t] -> [a]
Prelude> map (\x -> x+1) [1,2,3]
[2,3,4]
it :: Num a => [a]
```

At surface level, there are four different patterns involved, two per equation:

- f is a pattern which matches anything at all, and binds the f variable to whatever is matched.
- (x:xs) is a pattern that matches a non-empty list which is formed by something (which gets bound to the x variable) which was cons'd (by the (:) function) onto something else (which gets bound to xs).
- [] is a pattern that matches the empty list. It doesn't bind any variables.
- _ is the pattern which matches anything without binding (wildcard, "don't care" pattern).

In the (x:xs) pattern, x and xs can be seen as sub-patterns used to match the parts of the list. Just like f, they match anything - though it is evident that if there is a successful match and x has type a, xs will have type [a]. Finally, these considerations imply that xs will also match an empty list, and so a one-element list matches (x:xs) [6].

Let's look at another example, which is a declaration of a tree data type:

```
Prelude> data Tree = Leaf Int | Node (Int, Tree, Tree)
data Tree = ...
Prelude> let tree = Node(4, Node(3, Leaf 1, Leaf 2),Node(5, Leaf 6, Leaf 7))
tree :: Tree
```

We can define a recursive sum function that sums all the nodes' values:

```
Prelude> sum (Leaf n) = n; sum (Node(n,t1,t2)) = n + sum t1 + sum t2
sum :: Tree -> Int
Prelude> sum tree
28
it :: Int
```

Here we see the full power of pattern matching with algebraic types. Haskell matches an instance of a Tree type to the correct pattern: (Leaf n) or (Node(n,t1,t2)), which exactly match the patterns that were used to define the Tree type in the first place. This way Haskell knows how to compute the sum function on the tree. To emphasize this even more, consider the much longer Java code that does the same [5]:

```
public abstract class Node {}

public class Leaf extends Node {
    public int data;
}

public class Binary extends Node {
    public int data;
    Binary left;
    Binary right;
}

int sum(Node n) {
    if n instance of Leaf {
        return ((Leaf) n).data;
    }
    if n instance of Binary {
        Binary b = (Binary) n;
        return b.data + sum(b.left) + sum(b.right);
    }
}
```

3.5 Tail Recursion

A *tail call* is a subroutine call performed as the final action of a procedure. If a tail call might lead to the same subroutine being called again later in the call chain, the subroutine is said to be tail-recursive, which is a special case of recursion. Tail recursion (or tail-end recursion) is particularly useful in order to reduce the space complexity of recursive function.Tail calls can be implemented without adding a new stack frame to the call stack. Most of the frame of the current procedure is no longer needed, and can be replaced by the frame of the tail call, modified as appropriate (similar to overlay for processes, but for function calls). The program can then jump to the called subroutine. Producing such code instead of a standard call sequence is called tail call elimination. Tail call elimination allows procedure calls in tail position to be implemented as efficiently as goto statements, thus allowing efficient structured programming [9].

An example non-tail-recursive function for list summation:

```
   sum [] = 0
.. sum (x:t) = x + sum t
   :t sum
sum :: Num t => [t] -> t
=> it :: Num t => t
Prelude> sum [1,2,3,4,5]
15
it :: Num t => t
=> it :: Num t => t
-- Prelude> sum [1..999999] crashes
```

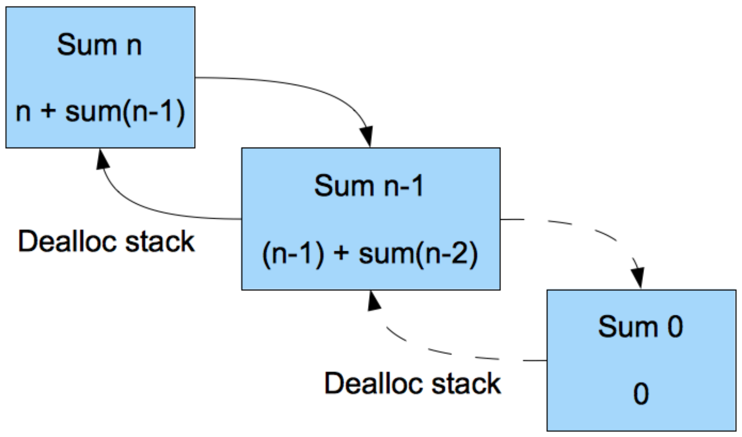

Figure 3: Non-tail-recursive frame calls

An example tail-recursive function for list summation:

```
.. sumIter xs = sumAux xs 0
   sumAux [] c = c
.. sumAux (x:xs) c = sumAux xs (c+x)
..
.. sumTail xs = sumAux xs 0
Prelude>  sumTail [1,2,3,4,5]
15
it :: Num t => t
=> it :: Num t => t
-- Prelude> sum [1..999999] works fine :)
```

The function SumAux is tail recursive because after the recursive call there is nothing else left to do.

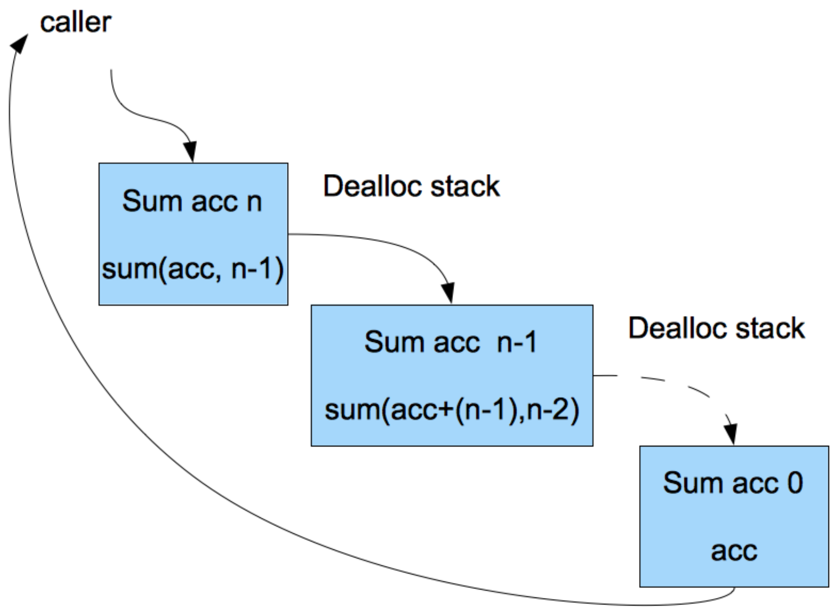

Figure 4: Tail-recursive frame calls

3.6 Lazy Evaluation

Haskell implements an evaluation methodology called lazy evaluation. In lazy evaluation, arguments to a function are only evaluated if and when they are needed in that function. When parameters are evaluated, they are only evaluated just enough to satisfy the needs of the function in the context they are used. If a parameter is passed to a function and never called or used in that function, it is simply never evaluated. If a parameter is used multiple times in the same function, it is shared among each reference to its value without being reevaluated each time. Moreover, using sharing ensures that lazy evaluation never requires more steps than call-by-value evaluation. An infinite list of Integers in Haskell can be defined as [1..]. Because Haskell uses lazy evaluations, we can use this infinite list as a parameter to a function without needing to worry that it will try to evaluate the entire list. For example, the "take" function will return the first x items of a list, so if we execute the following statement: take 10 [1..]. The output will be [1,2,3,4,5,6,7,8,9,10]. While an infinite list of integers is used a parameter, only the first ten items in that list are evaluated. This is a classic example of lazy evaluation.

A useful example of lazy evaluation is the use of quick sort:

```
quickSort [] = []
quickSort (x:xs) = quickSort (filter (< x) xs) ++ [x] ++ quickSort (filter (>= x) xs)
```

If we now want to find the minimum of the list, we can define

```
minimum ls = head (quickSort ls)
```

Which first sorts the list and then takes the first element of the list. However, because of lazy evaluation, only the head gets computed. For example, if we take the minimum of the list [2, 1, 3,] quickSort will first filter out all the elements that are smaller than two. Then it does quickSort on that (returning the singleton list [1]) which is already enough. Because of lazy evaluation, the rest is never sorted, saving a lot of computational time.

Laziness also enables us to define new control flow constructs that have to be built-in in eager languages. For example, laziness can be used for short-circuiting of logical OR as:

```
Prelude> :i (||)
(||) :: Bool -> Bool -> Bool          -- Defined in ‘GHC.Classes’
infixr 2 ||
=> it :: Bool
Prelude>    True || (1+1 /= 2) -- /= stands for "not equal"
True
it :: Bool
=> it :: Bool
Prelude>    False || (1+1 /= 2)
False
it :: Bool
=> it :: Bool
```

3.6.1 Lazy Evaluation and Side-Effects

When an expression is side-effect free, the order in which the expressions are evaluated does not affect their value, so the behavior of the program is not affected by the order. So the behavior is perfectly predictable.

Now side effects are a different matter. If side effects could occur in any order, the behavior of the program would indeed be unpredictable. But this is not actually the case. Lazy languages like Haskell make it a point to be referentially transparent, i.e. making sure that the order in which expressions are evaluated will never affect their result. In Haskell this is achieved by forcing all operations with user-visible side effects to occur inside the IO monad. This makes sure that all side-effects occur exactly in the order you'd expect.

References

[1] Gerald Jay Abelson, Harold; Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, 1984.

[2] ephemient. What is the difference between statically typed and dynamically typed languages? `https://stackoverflow.com/questions/4382223/what-does-pure-mean-in-pure-functional-language/`, 2010. [Online].

[3] Dan Johnson. Haskell - csc 415. `campus.murraystate.edu/academic/faculty/wlyle/415/2011/Johnson.doc`, 2011. [Online].

[4] NomeN. What is the difference between statically typed and dynamically typed languages? `https://stackoverflow.com/questions/1517582/what-is-the-difference-between-statically-typed-and-dynamically-typed-languages/`, 2009. [Online].

[5] Mooly Sagiv. Introduction to haskell. `http://www.cs.tau.ac.il/~msagiv/courses/apl17/HaskellIntro1.pdf`, 2017. [Online].

[6] Wikibooks. Haskell/pattern matching, 2017. [Online].

[7] Wikibooks. Haskell/variables and functions, 2017. [Online].

[8] Wikipedia. Higher-order function, 2017. [Online].

[9] Wikipedia. Tail call, 2017. [Online].