**Haskell paradigm research**



**Naam:** Vu Le

**Studentnummer:** 592420

**Klas:** ASD klas B, feb 2020

**Docent:** Niek van Diepen & Michel Koolwaaij (2019-2020)

**Assignment:** Onderzoek functionele paradigma’s  
**Course-/semestergegevens:** ASD APP, ASD-semester, jaar 3

**Datum:** 29 oktober 2020  
**Versie:** 1.0

Inhoud

[Introduction 3](#_Toc54792969)

[Basics about Haskell 5](#_Toc54792970)

[Arithmetics 6](#_Toc54792971)

[Conditional statements 7](#_Toc54792972)

[Lists 9](#_Toc54792973)

[Pattern matching 11](#_Toc54792974)

[Looping 12](#_Toc54792975)

[Quicksort 13](#_Toc54792976)

[Map/filter 14](#_Toc54792977)

[Exercises 16](#_Toc54792978)

[Hello, World! 16](#_Toc54792979)

[Leap 17](#_Toc54792980)

[Challenge 18](#_Toc54792981)

[Step 1: Initialize properties and test-edges 18](#_Toc54792982)

[Step 2: Generate graph 19](#_Toc54792983)

[Step 3: Let’s find the shortest path between two vertices 20](#_Toc54792984)

[Subquestions 23](#_Toc54792985)

[What are the basics of Haskell? 23](#_Toc54792986)

[What are the differences between Haskell and Java? 23](#_Toc54792987)

[What concepts learned in the APP-course are frequently used in Haskell? 24](#_Toc54792988)

[Conclusion 24](#_Toc54792989)

[Literature 25](#_Toc54792990)

## Introduction

*(This report is written in English)*

This repository contains all the code and needed images for my research about a functional programming language. This assignment is about learning a new language that interests me and differs from the language that we normally learn at school called Java (which is an object-oriented programming language).

The goal of doing this assignment is to learn the new programming language and spread my gained knowledge about it to fellow students. So the new programming language that I am about to learn is Haskell. Haskell is a purely functional programming language with type interference and lazy evaluation (only executes what is needed, instead only checks for errors). It is also immutable, so once a value is set, it can never be changed again. I chose Haskell because it is one of the most mentioned programming languages in functional programming, so I will be able to learn a lot because there is a lot written and documented online about Haskell.

For this assignment to be a success, I will do a challenge: implementing the shortest-path algorithm (Dijkstra) and demonstrating the algorithm. So obviously, the challenge will only be a success if I successfully implement the algorithm. I will also allow me to implement some learning material of the APP-course to add some difficulty to my research.

A lot of my information will come by this [YouTube-tutorial](https://www.youtube.com/watch?v=02_H3LjqMr8) and [an Haskell course](http://learnyouahaskell.com/). My plan is to follow these guides and some other sources (that will be mentioned in the literature-list at the bottom of this document) to show what I learned about Haskell and to use that knowledge to implement my chosen challenge.

What I will use for this assignment:

* Operating system: Windows 10 64-bit
* Visual Studio Code (with the Haskell Plugin) as my IDE
* Haskell Platform in the terminal (PowerShell)

My main question for the chosen challenge is: **Is it possible to implement the** [**shortest-path-algorithm**](https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm) **in Haskell?**

To help answer my main question, I have a few sub-questions:

|  |  |  |
| --- | --- | --- |
| **Sub-question** | **Research method** | **Explanation** |
| What are the basics of Haskell? | (Dutch) Lab, Bieb | I chose for these research methods because I will do my research mostly on the internet (books, PDF, official Haskell documentation). I will also make some sample code to practice with a few concepts of Haskell, which is why it’s also a Lab-research. |
| What are the differences between Haskell and Java? | (Dutch) Lab, Bieb | I chose for these research methods, because while I’m learning the concepts of Haskell, I will notice the differences in Haskell when reading through the Haskell-literature. But also, while writing code when learning the concepts of Haskell and my chosen challenge for the assignment. I will comment about these differences in both my research (chapter ‘Basics about Haskell’) and when I’m explaining my code in the chapter ‘Challenge’. |
| What concepts learned in the APP-course are frequently used in Haskell? | (Dutch) Lab, Bieb | The same explanation as ‘What are the differences between Haskell and Java?’ |

## Basics about Haskell

*All the code that is mentioned in this chapter, you can find in the file ‘basics.hs’.*

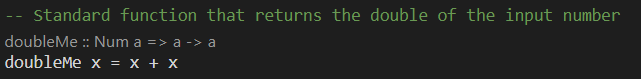
Let’s start with some basics about Haskell. A Haskell program has three fundamental building blocks: values, functions and types. (Haskell for Mac, 2020).  
  
Values are terms, for example a String (‘Hello world’), an Int (9) and a Float (1.45). You can use values to print them or process them through functions. As already, in contrast to Java, you cannot change a value or type in Haskell. It is immutable.   
Types are the properties to belong to a self-defined value. Values are grouped in sets of similar properties. Values like 1, 2 or 3 belong to the type ‘Int’. ‘Hello World’ and ‘This is Haskell’ belongs to the type ‘String’. Other types for example are Double (same as float, but with double precision and eats up more space to store information), Bool (true/false) and Char (‘a’, ‘B’, ‘c’). Below are a few examples to define a type for a value:

1 :: Int

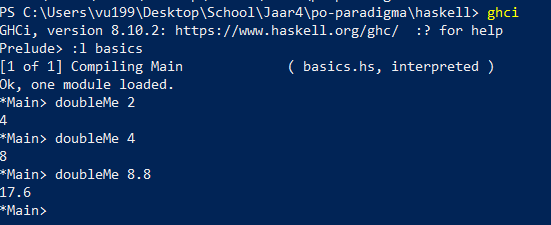
"Hello" :: String

So the value 1 is assigned to the type ‘int’ and the value “Hello” is assigned to the type ‘String’.   
Values and types so far are pretty much the same as Java.

Functions are mappings from input values to output values. A function definition compromises a head and a body separated by an equals sign, for example:

  
*Figure 1: Example of a standard function*

doubleMe is the name of the function, the variable ‘x’ in the head (left side of the equals sign) is the given parameter in the signature of the function and the body of the function is on the right side, where in this case the variable ‘x’ increments with itself. So it’s doubling up, this is outcome in the terminal:

  
*Figure 2: Result of the function in Figure 1 in the terminal*

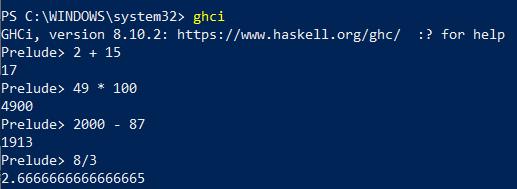
As you can see, if you put in the number ‘2’, this will result in this calculation in the function:

doubleMe 2 = 2 + 2

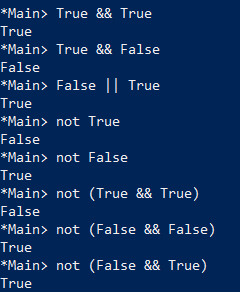
This results in number ‘4’.

### Arithmetics

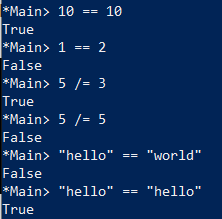
Making standard calculations does as is expected. Several operators can be used on one line, and multiple standard arithmetic rules are obeyed:

  
*Figure 3: Result of performing different arithmetics in the terminal*

Boolean are quite the same as written in Java: **&&** is the logical AND, **||** is the logical OR and **not** denies a True or False:

  
*Figure 4: Result of performing boolean operators in the terminal*

Testing for equality is always done with an ‘==’-operator, like it is defined in an if-statement in Java when comparing 2 values, this will also return either True or False:

  
 *Figure 5: Result of testing equality in the terminal*

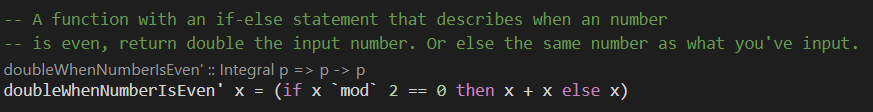
### Conditional statements

Now I’m wondering, are there conditional statements in Haskell, just like in Java? Well, there are:

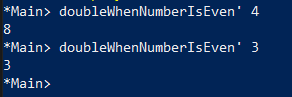
#### If-else statement

Haskell has if-else statements: the ‘else’-statement has become mandatory for Haskell: every expression and function must return something. This is because Haskell is a pure functional programming-language, which its expressions and functions must return something to avoid side-effects. The character ‘ is used to indicate that it is not a lazy function, or a slightly modified version of a function or a variable.

Here is an example of such an if-else-statement, in a function called ‘doubleNumberWhenEven’, so it checks if the input-number is even, if it is even, double up the input number. Else it won’t double up the number when the number is uneven:

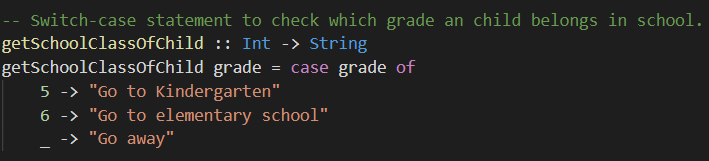
  
*Figure 6: Function to check if the input-number is even*

Here is a sample output when this function is called in the terminal:

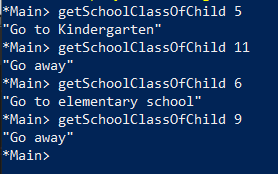
  
*Figure 7: Result of the function in Figure 6 in the terminal*

#### Switch-case statement

You can also use switch-case statements in Haskell (Haskell : case expressions, n.d.)., in the form of:

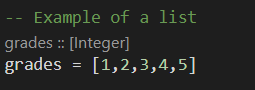
  
*Figure 8: Switch-case statement to check which grade an child belongs in school*

So, as you can see above, we defined in this case, to get the school class of a child, that the grade must be an Int and the description of the grade is a String. If the child is in grade 5, “Go to Kindergarten” will be returned. If a child is in a grade that is not 5 or 6, the child has to go away (“Go away”).   
Below is a sample output of the execution of getSchoolClassOfChild:

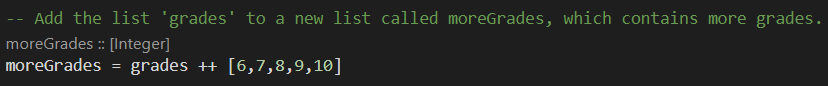
  
*Figure 9: Result of the function in Figure 8 in the terminal*

### Lists

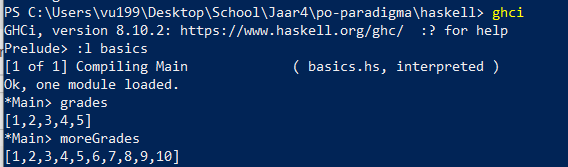
Lists and Haskell are singularly linked: with lists you can model and solve a multitude of problems. You can store multiple elements inside a list, as long each element is from the same datatype (Int, String, Float, Double). For example, you have a list of grades for students:

  
*Figure 10: Example of a list*

As you can see, 6 to 10 is still missing from the grade list. How can you add more grades tot the list? One way is to make a new list and merge the list ‘grades’ with another list (which contains the grades 6 to 10) to a new variable named ‘moreGrades’:

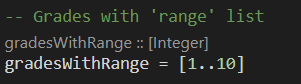
  
*Figure 11: Add existing list to new list*

The output of both lists is as followed:

  
*Figure 12: Output of both functions (Figure 10 + 11) in the terminal.*

Haskell also has functions for operating on lists: head, tail, last, init. Head returns the first element, tail returns everything but the head, last returns the last element and init returns everything except the last element.

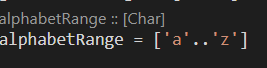
Moreover, Haskell has the ability to create a list with a defined range, for example we want the grades, but now with a range:

  
*Figure 13: Make a list with range (school grades)*

This returns in the terminal:

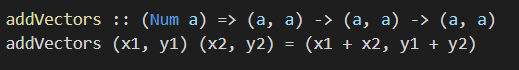
  
*Figure 14: Output of Figure 13 in the terminal*

It also works with characters, so in this example I’ll say that I want the alphabet:

  
*Figure 15: Make a list with range (alphabet)*

returns  
  
*Figure 16: Output of Figure 15 in the terminal*

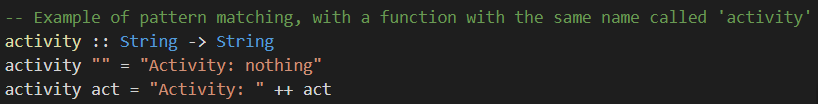
Haskell also has Tuples. Tuples are used when you know exactly how many elements you require to be grouped together. Unlike lists, each element in the list doesn’t have to be of the same type. You can put tuples in a list, but they require the same number of elements. For example:

  
*Figure 17: Example of a tuple, in this case the computation of vectors.*

But something like [(1, 2), (3,6,12), (24, 48)] would return an error.

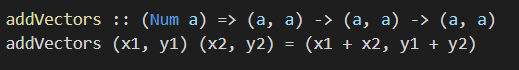
### Pattern matching

Pattern matching is to make and have patterns to which data belongs. First the data is checked to which pattern it belongs and then the right pattern will be called with a different body and output.   
For example, it can allow us to conditionally call a function, with different inputs that are used as like an if-else statement:

  
*Figure 18: Example of pattern matching*

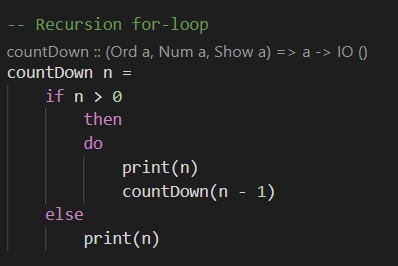
This basically says: if there is no activity (“”) then return “Activity: nothing” or else return the activity with “Activity: “.

Pattern matching can also be used on tuples, the following function shows how to add two vectors together:

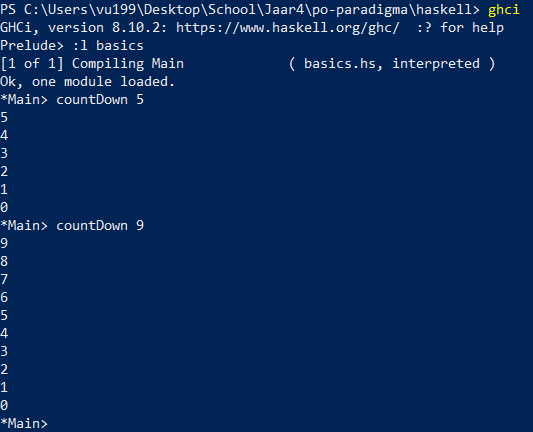
  
*Figure 19: Example of pattern matching with tuples*

### Looping

Haskell doesn’t know looping like most other programming languages do. But you can achieve the same goal by doing recursion. Here is an example of a countdown-function as for-loop:

  
*Figure 20: Recursion to make a ‘for-loop’.*

Sample output in the terminal:

  
*Figure 21: Output of Figure 20 in the terminal*

### Quicksort

One of the sorting algorithms that I’ve learned in the APP-course is called ‘quicksort’. It is a recursive sorting algorithm with basically the following characteristics:

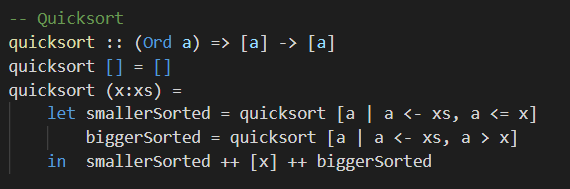
* There is the so-called pivot, that is being used to compare the values to. All the values that are lower than the pivot is on the left side, all the values higher than the pivot is on the right side. So, the original list is going to be divided in two: lower than the pivot and higher than the pivot. The pivot is mostly a random element in the list, the middle or a median (first, middle and last element divided by 3, for example first is 1, middle is 4, last is 2, so (1+4+2) / 3 = 2, 2 is the median).
* Algorithms works best when you could divide the original list in two even parts. So for example, 5 elements on the left and 5 elements on the right.

The type signature of Quicksort in Haskell is:

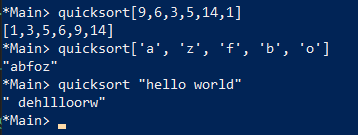
**quicksort :: (Ord a) => [a] -> [a]**

So the main algorithm is that the sorted list is a list has a hierarchy: first all the values smaller than (or equal to) the head (which is the variable ‘x’ in the example below) of the list in front (and those values are sorted), second comes the head of the list in the middle and third comes all the values that are bigger than the head (they're also sorted).

The code:

  
*Figure 22: Example of quicksort (learnyouahaskell.com/recursion)*

This is a sample outcome of the terminal when sorting different lists:

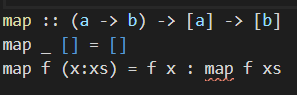
  
*Figure 23: Output of Figure 22 in the terminal*

What is noticeable in the difference between Haskell and Java is that when for the pivot, Haskell automatically chooses the head of the list to be the pivot. In Java, that would cause the worst-case performance (Quicksort, 2008), while in Haskell the reason for that is because it will be easier to get by pattern matching.

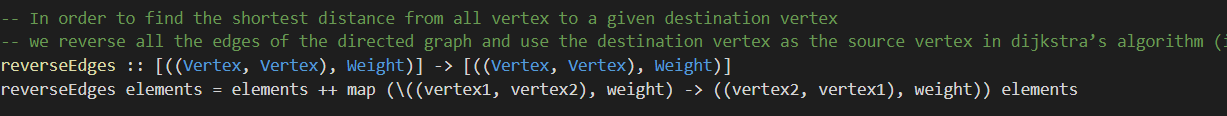
### Map/filter

#### Map

Map executes a function on each element of the given list and returns a new list. Map is a higher-order function (Elliot, 2019): an function that takes one or more functions as argument and returns a function.  
The type signature and implementation for map go like this:

  
*Figure 24: Type signature and sample implementation of the map -function.*

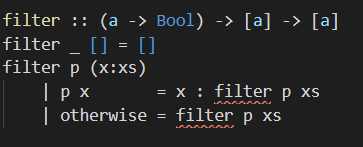
For example, I used the map-function when reversing the edges for generating a graph in the implementation of Dijkstra’s algorithm (chapter ‘Challenge’ -> Step 2: Generate graph):

  
*Figure 25: Map-function in action in implementing the Dijkstra-algorithm.*

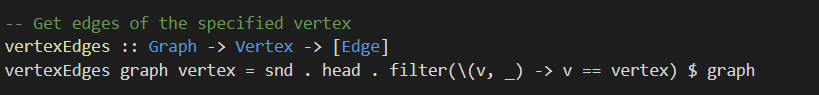
As you can see, it takes the variable elements (which is a function to create the test-edges for the graph) as argument in the map-function, and ultimately returns the edges with reversed vertices.

#### Filter

Filter is a function that takes a predicate (a predicate is a function that tells whether something is true or not, so in our case, a function that returns a Boolean value) and a list and then returns the list of elements when each element returns true at the predicate. The type signature and implementation for filter go like this:

  
*Figure 26: Type signature and sample implementation of the filter -function.*

For example, I used the filter-function to look for the edges for each vertex, after I generated the graph in my implementation of the shortest-path-algorithm (chapter ‘Challenge’):

  
*Figure 27: Filter-function in action in implementing the Dijkstra-algorithm.*

What this says is that we filter the graph, search for the matching vertex, and get the edges inside of it. When the graph is generated, for each vertex, each edge is already linked with a vertex. More explanation about the use of the filter-function is also to find in the ‘Challenge’- chapter -> Step 2: Generate graph & Step 3: Let’s find the shortest path between two vertices.

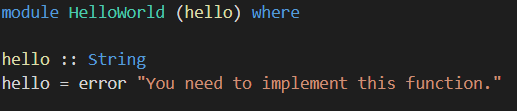
## Exercises

To get more comfortable with Haskell, I have decided to make a few exercises to get used to it.

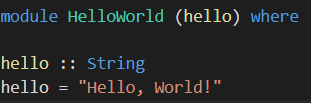
### Hello, World!

My first exercise was to write a function that returns “Hello, World!”.

*Assignment*

  
*Figure 28: Problem in Hello, World!*

*Solution*

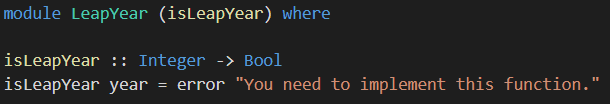
  
*Figure 29: Solution in Hello, World!*

**Explanation**

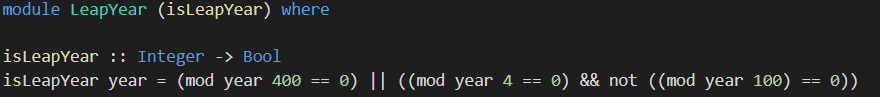
In the module ‘HelloWorld’ is an export function called ‘hello’. In that same function, I changed the body of the function, where I replaced the template error simply with a String (“Hello, World!”).

### Leap

My next exercise was to make a function that would check if the input year was a leap year.  
  
*Assignment*

  
*Figure 30: Problem in LeapYear*

*Solution*

*Figure 31: Solution in LeapYear*

**Explanation**

In the function ‘isLeapYear’, the adjustments that I had to make is to replace the template error “You need to implement this function” with checkers: one check to see if the input year divided by 400 would result in 0 (so it’s residueless), the other to check if the input year divided year would in 0 AND if the input year divided by 100 would NOT result in 0. Both checkers were divided by a logical OR, which means that either check would work.

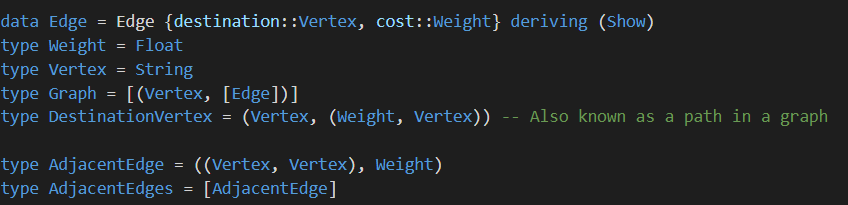
## Challenge

*Implement the shortest-path-algorithm in Haskell.*

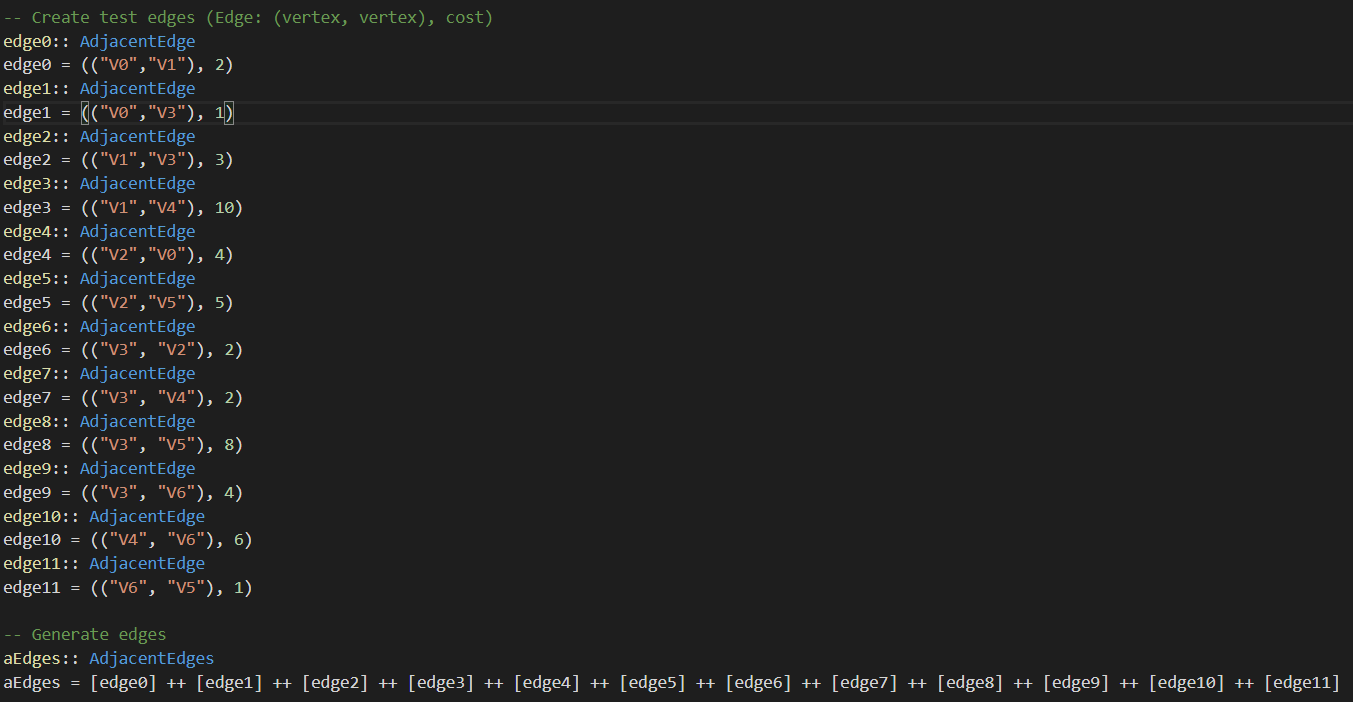
To give a short summary about the shortest-path-algorithm, the shortest-path-problem is the problem of finding a path between two vertices (nodes) in a graph, where the weight of its edges must be minimized.

### Step 1: Initialize properties and test-edges

First, I made a list of all the types (properties) that I needed:

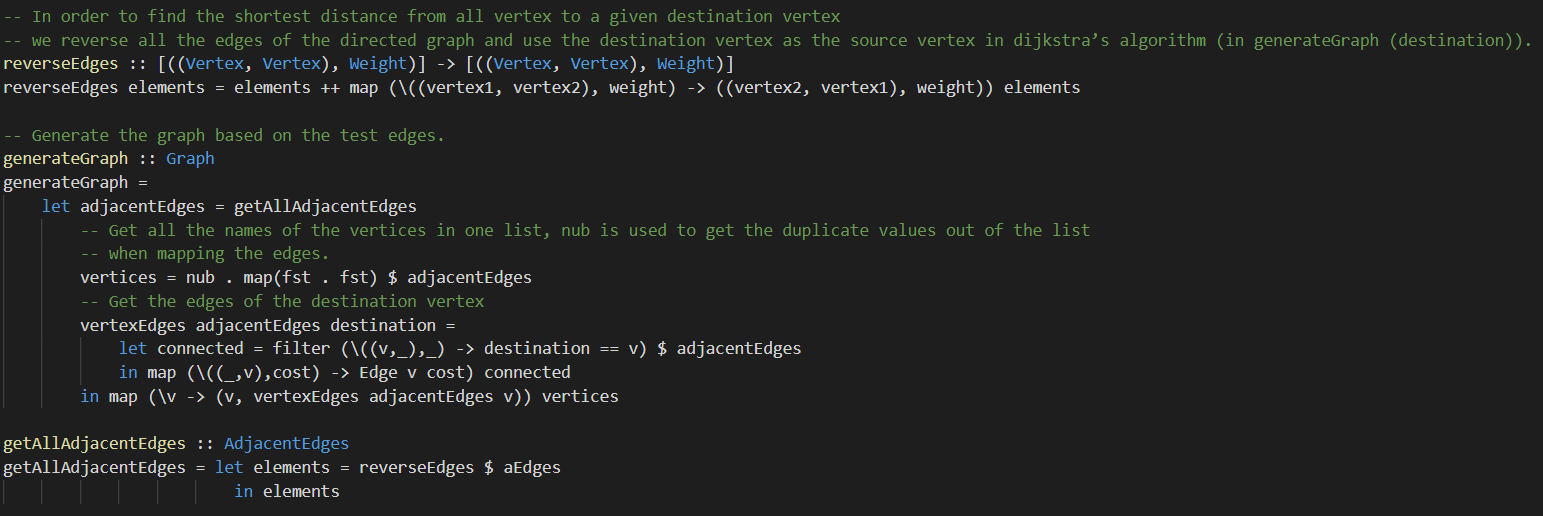
  
*Figure 32: Needed properties for implementing Dijkstra*

Then I made some test-edges to generate a graph:

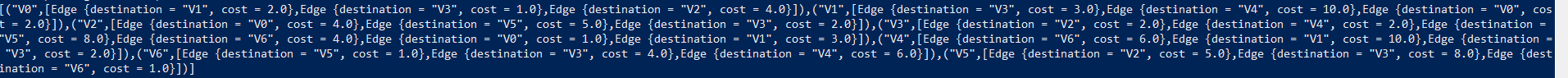
  
*Figure 33: Generate test-edges for generating a graph*

### Step 2: Generate graph

To generate a graph, I also needed to generate reversed edges (reverseEdges-function in the photo below) because in order to find the shortest distance from all vertices to a given destination vertex, we need to reverse all the edges of the directed graph and use the destination vertex   
as the source vertex. In reverseEdges, we update the current list of adjacent edges (elements) with the reversed edges of all the current adjacent edges, by executing a map-function that basically reverses the pair of vertices for each edge:

  
*Figure 34: Generate graph with all adjacent edges*

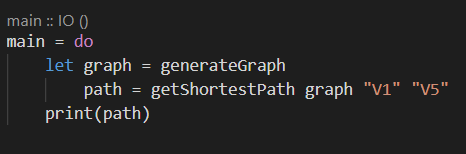
Notice that in the function ‘getAllAdjacentEdges’ that a dollar sign is used, that dollar-sign is basically a replacement for parentheses. So normally in Java, we would write something like: reverseEdges( AdjacentEdges aEdges), or something like that.   
To generate a graph, we get all the adjacent-edges. When we get the adjacent-edges, we get the vertices out of the list of the edges. The vertices are strings, and basically have a name. That’s it!   
But in the adjacent-edges list, we have to get the name of each vertex by mapping into the first element of the current edge from the adjacent-edges and in the first element of the pair of vertices in the current edge. To prevent duplicate vertices in the list, I needed to use the function ‘nub’ to remove duplicate elements from the list.

Then, for each vertex, we find all the destination vertices and the cost (weight) between them by using the filter-function to filter the adjacent-edges. So basically all the edges that belong to each vertex. This results in this, as shown in the terminal:   
*Figure 35: Output of function ‘generateGraph’.*

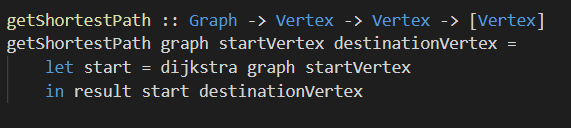
As you can see, all the connected edges with destination vertices and the weight (cost) between each vertex and its destination vertex is shown.

### Step 3: Let’s find the shortest path between two vertices

Now, let’s get to finding a shortest path between vertices. So in this case, I want to find the shortest path between V1 and V5. The main-function is in the ‘main.hs’-file:

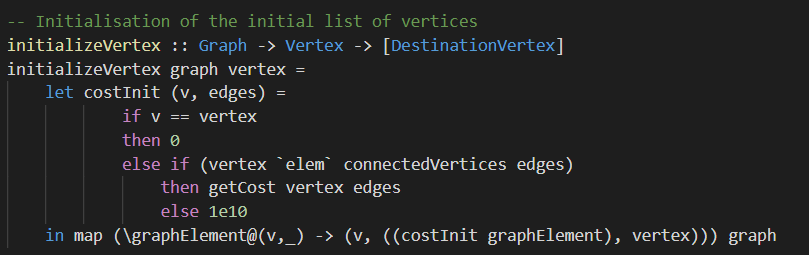
  
*Figure 36: Main-function to start getting the shortest path of the specified vertices*

As you can see, we first have to generate a graph as already explained above. When the graph is created, a function to get the shortest path will be called with some parameters: the generated graph, the start vertex (startVertex) and the destination vertex (destinationVertex).  
This is how getShortestPath looks:

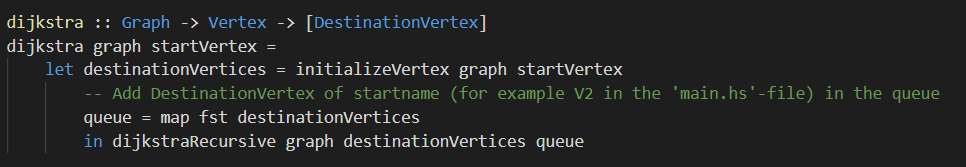
  
*Figure 37: getShortestPath function*

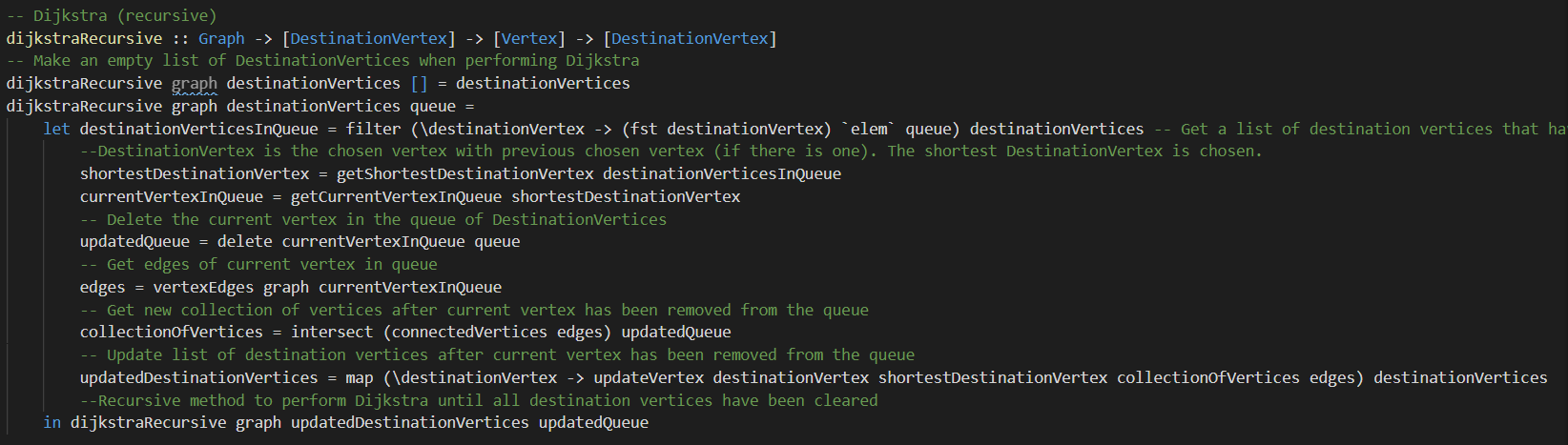
#### Perform Dijkstra’s algorithm

So, as you can see, there is a function called ‘dijkstra’. In that function, first we initialize the initial list of possible destinations of all the vertices by calling ‘initializeVertex’. The given parameters in ‘initializeVertex’ are the generated graph and the start vertex (starting point of the shortest path) which we specified in the main-function:

  
*Figure 38: initialization of a list of initial vertices*

As you can see above, we initialize the cost (weight) for each destination-vertex. Then we map the destination vertices into a list of destination-vertices. Now let’s head back to the ‘dijkstra’-function, where we now make a queue, where all the destination-vertices will be stored (with only their name (V1, V2, V3) as represented):

*Figure 39: Dijkstra function*

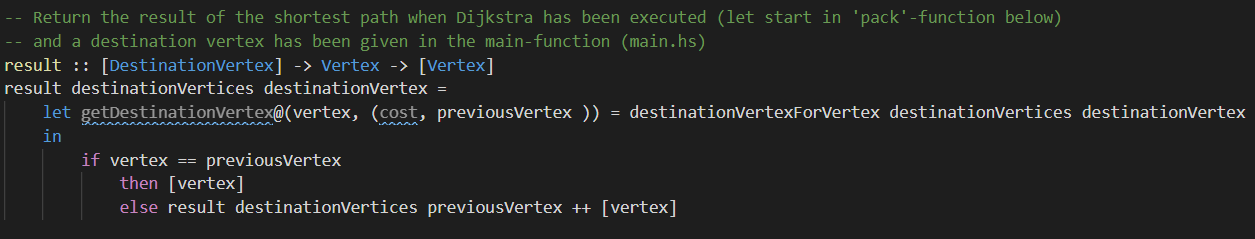
As you can see above, the recursive-method to perform the Dijkstra-algorithm is going to be called, the green comments in the code describes the steps within the function:  
*Figure 40: Recursive function to perform Dijkstra to get the shortest path*  
The algorithm will be called over and over until the list of destination vertices that have been specified in the method ‘dijkstra’ have been cleared.

There are a few observations and functions in this code that need to be clarified:

* function getShortestDestinationVertex: sorts the destination vertices by cost   
  by looping through all the destinationVertices and comparing the cost of a pair of destinationvertices, to pick the destination (path) with the least cost (weight)  
  (cost in ascending order), and then get the first element of the list (the one with the least cost)
* function getCurrentVertexInQueue: gets the first element (the vertex) of the shortest destination vertex that has been chosen.
* Function vertexEdges: gets the edges of the specified vertex.

#### End result

After the recursive function of Dijkstra’s algorithm has been called, in the function getShortestPath, the function ‘result’ will be called:

  
*Figure 41: Generate end result of shortest path*

This function will basically get the information of the destination vertex specified in the main-function, which is V5. It will check in an if-statement if the previous vertex in the shortest path is the same as this destination vertex. If this returns ‘true’, return the vertex. If not, add the vertex to the end result. The end result is obviously the shortest path generated by the recursive method (dijkstraRecursive).   
When it is all said and done, this will be the end result:

  
*Figure 41: Shortest path of V1 to V5 in the terminal*

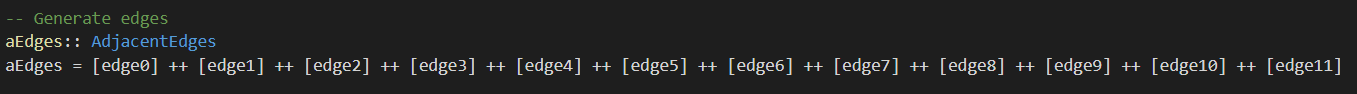
## Subquestions

### What are the basics of Haskell?

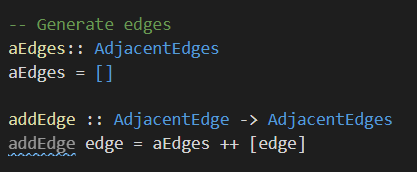
This question is answered by documenting the basics in chapter ‘Basics about Haskell’.

### What are the differences between Haskell and Java?

During the process of implementing my chosen challenge to implement Dijkstra’s algorithm and practicing the concepts of Haskell, I have noticed quite a few differences between Haskell and Java. In this chapter, I will discuss a few important differences and give my opinion on them.

Data structureSo the first difference already is that Haskell has an immutable data structure, so the output of a certain function and data (functions and variables) will be easier to refactor because there will be no side-effects and mutation allowed. In Java, you can for example create a list, but fill that list later with information from a database. If the database doesn’t have the required information, the list will stay empty. If the database has the required information, you can fill the list with that information. In Haskell, you can only set a function or variable once. This also means that you can never change that function or variable again after they are evaluated. It’s like setting the **‘final’** modifier in Java for every variable. This is what I have experienced when for example making the test-edges to generate the graph for Dijkstra’s algorithm:

*Figure 42: Generate test-edges as immutability example*  
The function ‘aEdges’ returns a list of edges to be used to generate a graph. As you can see, I set all the edges in one function. After that, ‘aEdges’ will be evaluated and can never be changed again. I actually tried implementing a function that would dynamically adding edges:

  
*Figure 43: A not working attempt to dynamically add edges*

But you can already tell, this **won’t** work. Because once ‘aEdges’ is set, it will never be changed again. In this case, it will always be an empty list. So no graph can be made if the required edges are not defined.   
In Java, this kind of function would work, because mutation of an existing variable of function is allowed, as earlier explained with the database-example. In my opinion, I like Java in this case better, it allows me to be flexible with initializing functions and variables and being able to modify them.

#### Objects vs Functions

Everything in Haskell can be regarded as a function, while in Java, everything (except for primitive types) are considered to be objects (Fleischer, 2013).

#### Lazy vs Short-circuit evaluation

As mentioned earlier in chapter ‘Introduction’, Haskell has lazy evaluation, which means that unless the programmer or user tells it to, it will not execute functions and calculate variables and things unless these functions and variables are needed to respond with the right information for the user. So an expression is evaluated only when it is absolutely needed. Java has short-circuit evaluation, which means that when evaluating Boolean expressions (logical AND OR), it will stop as soon as you find the condition that satisfies the expression.

### What concepts learned in the APP-course are frequently used in Haskell?

#### Recursion (& Quicksort)

Recursion is a very important concept in Haskell because you do computations based on what something is instead of saying how you get it. There are no while or for-loops in Haskell which is why we use recursion to declare what something is. It is like the replacement for loops in Haskell.  
One of the best examples of recursion is Quicksort, which has become the poster child of sorting in Haskell. The algorithm is naturally recursive because it sorts a list of items by dividing it into smaller sub-lists and then applying the same algorithm (Quicksort) on those. It is called over and over again until the original list has been sorted. More explanation about Quicksort is in chapter ‘Basics about Haskell’ -> Quicksort.

## Conclusion

In the chapter ‘Introduction’, I mentioned that the goal of this assignment is to learn a new language of a new programming paradigm that differs from the language that I am used to (Java). The main question that I formulated was if I could implement the shortest-path-algorithm in Haskell, by using the knowledge that I have gained from learning the basics of Haskell.   
Did I succeed? I think so. I took this challenge head on and documented everything I learned in this research. I used that same knowledge in the challenge, which made it a lot easier. I also many sources (as mentioned in chapter ‘Literature’) to get the information I needed to learn about Haskell and getting the required information to make my chosen challenge a success. This means that the chosen sub-questions were very useful, as I used the knowledge of having answered those sub-questions helped me implement the shortest-path-algorithm.   
I had a lot of fun with Haskell: I wrote code of some basic concepts of Haskell (chapter ‘Basics about Haskell’), made some exercises and got to understand concepts I have learned in the APP-course even better. For example, recursion. Because there are for example no while- and for-loops, I got to understand recursion even better now. Which I formulated in the chapter ‘Subquestions’ -> What concepts learned in the APP-course are frequently used in Haskell? I understood that recursion make a problem simpler.   
I also used recursion to perform Dijkstra to find the shortest-path by collecting all the destination vertices that vertices can go to. If I am going to use a functional paradigm, for example in the DWA-minor, I will definitely consider Haskell as my go-to-option.

## Literature

Haskell for Mac. (2020). *First Steps.* Retrieved October 26, 2020, from

http://learn.hfm.io/first\_steps.html

*Learn You a Haskell for Great Good!* (n.d.). Learn You a Haskell for Great Good! Retrieved October 26, 2020, from http://learnyouahaskell.com/

Wikipedia contributors. (2020, October 21). *Short-circuit evaluation*. Wikipedia. Retrieved October 28, 2020, from https://en.wikipedia.org/wiki/Short-circuit\_evaluation

B. (2015, July 20). *Functional programming vs. imperative programming - LINQ to XML*. Microsoft Docs. https://docs.microsoft.com/en-us/dotnet/standard/linq/functional-vs-imperative-programming#:%7E:text=Functional%20programming%20vs.-,imperative%20programming,functional%20approach%20to%20problem%20solving.&text=With%20an%20imperative%20approach%2C%20a,take%20to%20accomplish%20the%20goa.

*What are the pros and cons of functional programming compared to imperative programming? - Quora*. (n.d.). Quora. Retrieved October 28, 2020, from https://www.quora.com/What-are-the-pros-and-cons-of-functional-programming-compared-to-imperative-programming

Fleischer, J. (2013, June 27). *Bridging the gap between Haskell and Java.* Freie Universität Berlin.

Retrieved October 28, 2020, from http://page.mi.fu-berlin.de/scravy/bridging-the-gap

between-haskell-and-java.pdf

*Haskell : case expressions*. (n.d.). ZVON. Retrieved October 28, 2020, from http://zvon.org/other/haskell/Outputsyntax/caseQexpressions\_reference.html

Elliott, E. (2019, January 10). *Higher Order Functions (Composing Software) - JavaScript Scene*. Medium. Retrieved October 28, 2020, from https://medium.com/javascript-scene/higher-order-functions-composing-software-5365cf2cbe99

*Quicksort: Choosing the pivot*. (2008, October 2). Stack Overflow. https://stackoverflow.com/questions/164163/quicksort-choosing-the-pivot