UNIVERSITY OF TARTU
Institute of Computer Science
Software Engineering Curriculum

Kaarel Tinn

Developing a Course on Teaching Functional Programming in JavaScript

Master's Thesis (30 ECTS)

Supervisor(s):
Vesal Vojdani
Margus Niitsoo

Developing a Course on Teaching Functional Programming in JavaScript

Abstract:

This thesis describes the design and development process of a course teaching functional programming in JavaScript. The course is targeted at third-year BSc or first-year MSc students who are familiar with at least the basics of JavaScript. The course will be taught at the University of Tartu. This paper discusses the motivation behind the creation of the course. It also reviews problems with learning functional programming and argues how learning functional programming could be made more efficient by focusing on practical examples. The topics covered in materials - lecture slides and home assignment descriptions - are explained in depth. The overall course structure and delivery process is described. The thesis serves as helpful documentation to the course's future lecturer.

Keywords: Functional Programming, JavaScript, Course Design, Education

CERCS (Common European Research Classification Scheme):

S281 - Computer-assisted education

P175 - Informatics, systems theory

Kursuse "Funktsionaalne programmeerimine JavaScriptis" arendamine

Lühikokkuvõte:

Käesolev uurimistöö kirjeldab kursuse "Funktsionaalne programmeerimine JavaScriptis" disaini ja arendus protsessi. Antud kursus on mõeldud kolmanda aasta bakalaureuse või esimese aasta magistri tudengitele. Kursus on mõeldud õpetamaks Tartu Ülikoolis. Uurimistöö kirjeldab, mis asjaolud motiveerisid sellise kursuse loomist. Uurib tausta, et miks on funktsionaalset programmeerimist keeruline õpetada ja arutleb, et keskendudes rohkem praktilistele näidetele muudaks see õppimisprotsessi efektiivsemaks. Uurimistöö kirjeldab detailselt kursusel käsitletud teemasi ja annab ülevaate töö raames valminud toetavatest materjalidest. Kirjeldatakse üleüldist kursuse struktuur oja esitamise protsessi. Töö eesmärk on olla dokumentatsioon abistamaks tulevast kursuse lektorit.

Keywords: Funktsionaalne programmeerimine, JavaScript, Kursuse koostamine, Haridus

CERCS (Common European Research Classification Scheme):

1. S281 - Arvuti õpiprogrammide kasutamise metoodika ja pedagoogika
2. P175 - Informaatika, süsteemiteooria

Table of Contents

1. Introduction

This thesis describes the design and development of the course about teaching functional programming. Functional programming is a programming paradigm and it differs from more conventional paradigms like imperative and object-oriented. Functional programming restrains programmers by encouraging use of pure functions, handling data structures immutably, avoiding side-effects. Learning functional programming means unlearning skills that are used in other paradigms. Often, functional programming is taught using purely functional programming languages like Haskell or F#. The course in question is meant to be an introduction to functional programming that uses the popular programming language JavaScript. The course focuses on the practical application of functional programming. This means the examples that are used in the lectures are inspired from real-world problems and exercises in the home assignments are constructed so that they simulate common real-life problems.

Chapter 2 covers the background of JavaScript and functional programming. It explains the motivations behind creating such a course and argues about the relevancy of functional programming in the software engineering industry. It is followed by defining the course's goals and target audience. Finally, related works are reviewed which have influenced the creation and the content of the course. Chapter 3 describes the main design principles that were followed during the development process. It explains the course's learning objectives and divides them into main and secondary objectives. The chapter ends with a high-level overview of the course - describing administrative considerations, the structure of lectures, the home assignments, a final project, and the course's grading scheme. Chapter 4 describes the materials that were developed to support running the course. Firstly, it lists resources used in the course. Secondly, it gives an in-depth description of each lecture and its topics. Thirdly, the exemplary home assignments are shortly described to give a sense of what students must solve to pass the course. After this, both non-functional and functional requirements are listed together with potential final project ideas. The final project is meant to put all of the skills learned during the course into practice. Lastly, topics that were considered for inclusion in the course materials are examined. An explanation is provided as to why these topics were excluded.

2. Background

Functional programming is a programming paradigm that relies on pure functions that manipulate immutable data and avoids side-effects as much as possible. The foundations of functional programming go back to the 1930s, when Alonzo Church published multiple papers on lambda calculus. The first actual functional programming language, LISP, was created in the 1950s. Lisp was based on lambda calculus notation. Later more modern functional programming languages were invented, including ML dialects and Haskell. These languages included features that made programming more practical: a static type system, polymorphism, pattern matching, algebraic data structures, lazy-evaluation, etc (Tan, 2004).

2.1 Why does functional programming matter?

Multiple sources claim that functional programming tends to increase the abstraction level of the codebase (Miyata, 2017; Braithwaite, 2013; *Major Programming Paradigms*, s.a.). This section covers how functional programming achieves this and why it is relevant in front-end application development.

2.1.1 Increasing use in practice

Functional programming has been getting more popular recently. The most significant increase has been in the domain of user-facing interface development for internet browsers. Technologies like React and Redux have been promoting functional programming to a broader audience and functional programming concepts in JavaScript are getting more attention (*React*, s.a.; *Redux*, s.a.). However - it has not always been like this. Object-oriented programming has been (and still is) the mainstream paradigm of the software engineering industry. This is mainly due to the programming language Java which gained most of its popularity in the 1990s. JavaScript was not considered a serious language for a long time, but as web programming evolved and web applications started to get more complex - the use of Javascript grew enormously (Elliot, 2017). This has led to an understanding that the way JavaScript programs were written before - mostly using an imperative coding style - is not sustainable in the long term. Although, being a multi-paradigm language that supports both an object-oriented and functional programming style, JavaScript is not an excellent language to write programs following object-oriented programming concepts. The latest additions to the language have been trying to relieve that as EcmaScript6 added syntax for classes.

On the other hand, functional programming is considered hard to learn and understand, especially among software engineering students who will professionally write code. The software engineering industry highly values code quality and simplicity. Functional programming makes it possible to write code that is declarative, easy to reason about and test. A possible issue for widespread adoption is that functional programming concepts are introduced academically and too abstractly. This means that in the future students will not recognize that they are using any functional programming concept and can not extend the code to leverage the power of functional programming concepts or can not apply the functional programming concepts on their own. This is especially true in modern front-end development since many new libraries are using functional concepts - like React and Redux.

Modern web development depends heavily on JavaScript and as the solutions are getting more complex the amount of JavaScript which is required for this is also increasing. This means JavaScript codebases are getting larger and functional programming and its techniques significantly reduce the related maintenance costs. Finally - application state handling is increasingly often done in the front-end instead of solely being handled server-side. Because of this - immutability, which will also be thoroughly covered in the course, is becoming a useful technique that professional software engineers are required to be aware of.

2.1.2 Strict conventions

The debate about which paradigm is better - object-oriented or functional programming - is never-ending and often contains a lot of subjective opinions about why one is better than the other. This section will not concentrate on why functional programming is better than object-oriented programming. Neither will it try to convert people to functional programming. Instead, it will try to give an objective overview on why functional programming concepts matter in software engineering and how following those can lead to better programs.

All programming paradigms tend to restrict programmers in one way or another. This strictness is useful because it prevents using harmful practices that are possible in other languages. For example, in functional languages, immutability is forced upon the data. This means that data structures cannot be changed in place. For example, to change elements inside the array, a new array has to be created and the changes will be applied to elements during the initialization of the new array. This concept is beneficial when it comes to parallel computing and prevents one process from changing the state of another process. A modern computer has more than one core and immutability makes it easier to leverage the power of them all. Another positive aspect of immutability is that it makes it harder to accidentally change data without the programmer being aware of it. This makes it possible to implement time-traveling tools for inspecting data during the lifecycle of the program. JavaScript does not have built-in immutability support, but knowing the concept, programmers can treat the data as immutable. For example, Redux suggests never mutating the state (although it is possible) and instead return a totally new representation of the state. Some libraries make it possible to simulate immutability in JavaScript, but they often add too much technical overhead to a project and using them might not be worth the effort.

2.1.3 Declarative thinking

Functional programming prefers a declarative style over an imperative approach. The declarative style focuses on “what” whereas imperative on “how”. The imperative style controls the flow of the data and builds the end result incrementally by doing data manipulations at every step. In the declarative style the computation is defined by what needs to change at every step and the end result is an aggregation of those changes. React is a good example of a declarative programming technique. In React, the component’s structure is defined first. Later, when the component is used, it is initialized with data (Mundy, 2017). Following the declarative style, developers can focus on what is important rather than getting lost into low-level details. This is even more important as a project grows and matures. This

is why modern front-end heavy applications are not written in the imperative library jQuery and why the declarative library React has become heavily popular in industry (Copes, 2018).

2.1.4 Competitive advantage

Another positive aspect of functional programming in the author's opinion is that the adoption of functional programming in a company's technology stack can give it a competitive advantage as functional programming can lure technical talent on board. On the other hand, knowing functional programming might increase developers' value in the employer's eyes. Functional programming has gained popularity in recent decades. Already mature object-oriented programming languages have been adopting functional programming constructs to its native toolkit. Languages like Scala, F# were created as functional counterparts to Java and C# (Bandt, 2019). Functional programming is more suitable for in-demand tasks like machine learning and data analysis (Moutafis, 2020).

2.2 Problems learning functional programming

This section describes aspects that make learning functional programming difficult and motivate the development of the course under the question. Firstly, it covers why functional programming is so difficult to learn compared to other paradigms. Secondly, it argues that the way functional programming is being taught currently is too academic. The learning process could be more efficient if introduced in a familiar language and based on practical examples.

2.2.1 Steep learning curve

Functional programming is considered difficult to learn in the software engineering industry. In the author's opinion, this is a misconception and exists for multiple reasons.

Universities and other institutions are teaching programming by introducing either imperative or object-oriented paradigms. They teach concepts that are very different from functional programming principles - variables, loops, mutable states, objects, classes. Object-oriented programming, for example, encourages the developer to hide the structure of the data, which is controversial in functional programming. Once students start to learn functional programming, they understand that their previous knowledge is not applicable in this situation. To achieve something, for example looping, they have to learn a whole new concept (recursion) since the language simply may not support looping (Sinclair, 2019). This causes frustration and leads to opinions about the complexity of functional programming.

2.2.2 Unfamiliar programming languages

Another reason which makes functional programming challenging to learn is that usually the first language used to teach the paradigm to the newcomers is either Haskell or some other languages from ML family ("*ML (Programming Language)"*, s.a.). The learning curve of Haskell is quite steep and the struggle of learning a new language may overshadow grasping the ideas behind core concepts of functional programming. The difficulty can be reduced if functional concepts would be introduced in a more forgiving language or in a language that the students already know. JavaScript is a good candidate because it has familiar syntax for

those coming from Java, C, or C++ backgrounds. It runs in the web browser and thus is easy to try out concepts without the additional overhead of installing and setting up a development environment.

Many object-oriented programming languages nowadays contain functional concepts like lambdas, comprehension. Haskell has a strong type system. Although being very robust, expressive, and powerful, it can scare away newcomers. Generally, strong-typing is considered useful since it allows developers to discover bugs faster and describes the code's intention more clearly than dynamic typing (Chiusano, 2016). It is arguable whether it is helpful to discuss this when introducing functional programming concepts to students since it would most likely considerably steepen the learning curve.

2.2.3 Lack of real-world examples

Often, when learning functional programming, it seems that it strays away from the real world and instead focuses too much on the theoretical aspects, making it quite abstract. In the author's opinion, aspiring programmers want to see how technology is used in action rather than to listen to tedious theoretical lectures. As in the object-oriented programming patterns, there exist patterns in functional programming which are often used in real-world projects. React's higher-order components are a good example of composition and JavaScript Promises are a neat way to handle values that may not exist in the future and illustrate a practical use case of monads (Sinclair, 2016). Based on that, new techniques of teaching functional programming that target experienced developers more familiar with object-oriented programming principles have been slowly starting to emerge (Petricek, 2012).

2.3 Teaching functional programming with JavaScript

This section covers how learning functional programming could be made more efficient by teaching it in JavaScript. A short introduction to JavaScript is made as well as how the language's features and popularity can positively affect learning functional programming fundamentals. This is followed by an explanation of the course's goals and a description of the target audience of the course.

2.3.1 Why JavaScript?

JavaScript is a general-purpose programming language and is used on several fronts in the software engineering industry. According to the StackOverflow Developer Survey 2020 (*Most Popular Technologies: Programming, Scripting, and Markup Languages*, 2020). JavaScript is the most popular language and it is used by 67% of developers who participated in the survey. JavaScript has been able to retain this position for 8 successive years. In the author's opinion, the reason behind that is the emergence of web-based software solutions as JavaScript is the only high-level language supported by modern browsers. JavaScript enables adding interactivity to web applications - achieving behavior that was once possible only for desktop applications.

JavaScript supports language features like higher-order functions and anonymous functions which are necessary to practice functional programming. In 2015 EcmaScript6 (a

cross-browser JavasScript standard) brought new syntax and features to the language. These include arrow functions and the spread operator. These additions make it easier to write in a functional style.(Eich & Wirfs-Brock, 2020, 1-189).

JavaScript's accessibility and popularity are the main reasons it was chosen as the programming language of this course. There are some secondary reasons. First - learning functional programming in a language that is already familiar (or similar to already familiar languages) positively affects the ability to understand functional programming concepts in the author's opinion. JavaScript enables us to build tangible web applications with ease and this makes the learning process more exciting.

2.3.2 Course goals

The main goal of the course is to teach the basics of functional programming in more practical ways. After the completion of the course, students can explain and understand different functional programming concepts like higher-order functions, composition, currying, and partial application, etc. They can tell the difference between mutability and immutability and why immutability plays a significant role in producing robust programs. They will be introduced to algebraic data structures and how those can be put into practice. Students should understand the benefits of functional programming as well as the drawbacks. It is important to stress that JavaScript is only a tool to teach functional programming concepts and that the bigger goal is to teach the fundamentals of functional programming so that they will be applicable also in other languages.

Another goal of the course is to broaden students' minds and improve overall programming skills. Functional programming requires a totally different mindset to approach the problems and this greatly benefits overall programming skills. After course completion, students will be more able to recognize different patterns in code and use functional programming techniques in the code they are writing.

2.3.3 Target audience

The course is primarily targeted towards 3rd year Bachelor students in the Computer Science curriculum or 1st year Master students in the Software Engineering curriculum. Basic programming skills are expected - this means students can already solve non-trivial programming problems using at least one modern programming language. JavaScript knowledge is not expected but will be beneficial since basics will be covered only very briefly. Resources for individual learning and getting up to speed with JavaScript basics will be provided.

2.4 Related work

This section covers related resources that have influenced developing the course materials. It takes a look into similar courses taught at the University of Tartu. This is followed by an overview of massively open online courses (MOOCs) that teach functional programming in one way or another. Finally, it covers a series of articles and other resources that multiple authors have published and are freely accessible on the Internet.

2.4.1 Related courses at the University of Tartu

There have been several courses over time teaching functional programming at the University of Tartu. In this section, I will describe three courses that are currently actively taught at the UT and how they are related to the new course.

“Programming Languages” (MTAT.03.006) is a course targeted at 3rd-year bachelor students. The first part of the course focuses on introducing functional programming concepts in Haskell. Haskell is a strongly-typed functional programming language with a strong emphasis on being completely pure in the sense that all side-effects must be handled explicitly. The second part of the course teaches multi-paradigm programming and introduces Scala. The main focus is on showing that different paradigms could be used together to solve a problem and on teaching students how to apply them in real-world scenarios. In the first part, the assignments are smaller and independent from each other, whereas in the second part they are more about solving a bigger task at hand. From the course feedback, it can be seen that students mostly like the course since it teaches them to be better programmers and it shows that there are other ways to write a program other than the object-oriented programming paradigm. Starting from the 2019/2020 academic year this course is no longer mandatory in the Computer Science BSc curriculum and is replaced by “Functional Programming” (LTAT.03.019), which will be taught in Idris (*Idris*, s.a.). It will still maintain the classical functional programming content previously taught in Haskell but will replace the Scala parts with topics that are more indicative of where programming language research is currently headed. It is a bachelor-level course, so it will only begin to show how more powerful type systems are used to prove properties about programs, but the idea is to make it more about fundamental ideas and less about practical applications. This is why there is now room for a more practical course on functional programming.

“Agile Software Development” (MTAT.03.295) is not strictly about functional programming itself rather than the process of producing software in an agile manner. It uses Elixir, which is a dynamically typed functional programming language. This course is mandatory for the Software Engineering curriculum and since in this curriculum students have different backgrounds - there might be students that have not seen functional programming before. This can cause difficulties since the pace of the course is rather high. Overall the course does a good job of showing how functional programming can be used in writing modern web applications.

“Interactive Frontend Development” (MTAT.03.313) is focused on how to build highly interactive front-end heavy applications. In this course, basic JavaScript skills are required. It introduces the React front-end library and Redux for state management. Since those two technologies are both influenced by functional programming - this course slightly covers topics like higher-order functions and currying. This course is not compulsory in any curriculum.

“Programming Languages” teaches functional programming, but it does not touch the practical part of it and merely focuses on the functional programming concepts. “Agile Software Development” and “Interactive Front End Development” are more oriented to real-world problems, but they do not focus on functional programming explicitly and connections have to be drawn independently.

The overview provided shows there is an unfilled space for a course similar to what this thesis describes: a course that uses real-world problems as an inspiration to teach fundamental functional programming concepts.

2.4.2 Related online courses

In this section, we will go over the freely available courses found on the Internet. These courses do not use JavaScript. It will try to understand how they approach teaching functional programming: what concepts they introduce and in which order, how the exercises are designed, etc.

"Programming Languages" is provided by the University of Washington via the online learning platform Coursera. It tries to improve the programming skills of students by introducing three different programming languages: ML, Racket, and Ruby (*Programming Languages, Part A*, s.a.). It is not only about functional programming itself but has a strong emphasis on the topic. The main goal of the course is to develop skills useful to learn new programming languages more easily and to use language concepts in various settings. The first part of the course (part A) touches on different functional concepts such as higher-order functions, recursion and pattern matching, etc. The course has also a part B and C. Part B is about comparing different type systems and introduces laziness and memoization techniques. Part C compares functional programming to object-oriented programming and is generally more focused on object-oriented programming. The assignment descriptions are detailed and technical. There are assignments at the end of every week. The solutions are auto-graded as well as peer-reviewed. One thing to take away from this course is how the assignments are structured and presented to the students.

"Functional Programming Principles in Scala" is provided by the École Polytechnique Fédérale de Lausanne (*Functional Programming Principles in Scala*, s.a.). The course syllabus includes topics like higher-order functions, data and abstraction, types, pattern matching, lists and collections. It uses the multi-paradigm language Scala. The course requires previous programming knowledge and familiarity with at least one object-oriented programming language. The course is six weeks long and the main goal is to introduce functional programming principles clearly and concisely. The course uses a series of videos introducing new concepts. There are optional quizzes and a programming assignment that counts towards the final grade at the end of every week. This course is a good example of how to introduce functional programming principles using multi-paradigm languages.

2.4.3 Related resources on the web

Many resources on the web found using the terms "functional programming" and "javascript" are related to functional programming in JavaScript. Many of them are blog posts written by programmers. It is difficult to distinguish good content from mediocre ones in this vast amount of articles. In this section, we will describe some notable books and article series about the topic.

One of the best resources in the author's opinion regarding functional programming in JavaScript is "Mostly Adequate Guide to Functional Programming" (Lonsdorf, 2015). It is a concise introduction to functional programming concepts and is freely available online. The first chapters teach basics like higher-order functions, functional purity, currying, and

composability. The later chapters are more about advanced concepts like algebraic data structures and category theory. The examples in the book are original and to the point. The book states that JavaScript is probably the world's most popular functional programming language in its introductory chapter. It is likely that developers reading the book already use functional programming concepts in his/her everyday work. The content and theme of the book strongly relate to the course being developed and will be an influence for the course.

Another good resource about functional programming in JavaScript is the article series "Composing Software" by Eric Elliot (Elliot, 2019). In these articles, the author looks at the big picture of how functional programming has been used in the past and why it has resurged in relevance again. He also states that software engineering is all about composing things together. The developers do it all the time and that it is better to understand what it means. Otherwise, mistakes could be made, leading to poorly designed software. In the series, the following topics are covered: composability and what it means, the history of functional programming, immutability, pureness, higher-order functions, currying, functors, monads, abstraction and composition. The theme of those articles is more general, also the examples are a bit superficial - despite that they are still useful to illustrate the point. Ideas in these articles can be used to present more higher-level concepts of functional programming to students.

Tom Harding's article series "Fantas, Eel and Specification" is a good explanation of Fantasy Land Specification algebras (*Fantasy Land Specification*, s.a.). Fantasy Land Specification describes a set of operations and laws objects must have and obey to comply with the specification. Library authors mainly use the specification to make sure their libraries comply with the same rules and those different libraries could be used together without worrying about interoperability problems. Algebras defined in Fantasy Land Specification are similar to Haskell's type classes. All Fantasy Land Specification does is provide naming conventions for algebraic structures. The article series goes over most of the algebras defined within Fantasy Land Specification and tries to explain intuition behind every algebra. It contains semi-practical examples written in vanilla JavaScript (Harding, 2017).

There are also other good resources, but those listed above are the ones the author trusts the most. This is mainly because the community recommends them and the authors are well-known. They are also freely available.

3. Course design

This section goes over the main principles that were considered in the design of the course. Main and secondary learning objectives are examined and justified. Finally, a high-level overview of the course structure will be provided and the course's materials will be described.

3.1 Main design principles

The main design principle for the course is to be as practical as possible. Multiple resources state that functional programming is more difficult to understand and learn than other paradigms (Scalfani, 2020; Sinclair, 2019). Using practical examples and exercises helps keep interest in the topic and allows the students to see non-academic applications for the paradigm. In the course, no web frameworks will be used. Instead, a small front-end architecture with the help of small helper libraries will be built and used to render HTML web pages with interactive capabilities. In the author's opinion, learning and building applications from scratch help with understanding how web frameworks or libraries work behind the hood. Knowing the design principles of a framework makes it easier to use it in practice.

The course's delivery process and materials have been influenced by the rules of good textbook design (*Wikibooks:Textbook Considerations*, s.a.). These rules describe what a good textbook should follow to be well-structured and present its topics formally. These rules also apply for designing a course since the purpose of textbooks and courses are similar - only the format of delivery varies. These rules are used to set a structure for the materials and delivery process. Next, each rule will be discussed in terms of course design.

Rule of Frameworks

It is easier to acquire new knowledge if there is a clear and familiar structure for how the learning materials are presented. Different topics are presented using a similar structure and it should be low effort to navigate within the materials. For example, there is usually an introductory slide on the lecture slides followed by slide(s) having code examples to consolidate the knowledge.

Rule of Meaningful Names

According to the textbook the human mind tries to remember various things by giving names to them. If multiple names exist for the same thing - it will require mental power to extract enough information from the surrounding context to understand what is being described. Having concrete, fixed and distinct names for concepts makes it easier to acquire new knowledge. There has been an attempt to follow this rule and give a single distinct name for each concept taught. Where applicable, it has been described that there are other names for the concepts as well. For example, in Lecture 5, the confusion around names in Fantasy Land Specification is explicitly addressed. Also, algebraic data structure's methods have many aliases used in other resources and those are listed on Lecture 5 slides.

Rule of Manageable Numbers

This rule concerns keeping the number of new concepts introduced each week as low as possible. To achieve this - there have been topics that have been discarded. For example, it was considered to teach recursion and closures in Lecture 2 and 3, respectively. For that same reason, the use of TypeScript as a course's programming language was considered but discarded in favor of vanilla JavaScript.

Rule of Hierarchy

This rule states that the mental learning model is hierarchical. It is easier to absorb new information when it is linked to already known concepts. Topics in this course are ordered roughly from easiest to most complicated.

Rule of Repetition

This rule states that mastery requires repetition. For example, pure functions are being used throughout the course as well as other fundamental concepts.

3.2 Learning objectives

A learning objective is a concept the student should be familiar with after finishing the course. In this course, the objectives are divided into main and secondary learning objectives. The main learning objectives are about the fundamental concepts of functional programming. Main learning objectives are explained and emphasized through-out the course, whereas the secondary objectives might not be. Secondary objectives might be more detailed ideas or techniques that are still important in the light of functional programming. Reason behind this distinction is that if students go away knowing only about main learning objectives then it gives them a strong basis for discovering functional programming on their own. Functional programming is a broad topic and learning everything at once is impossible. This distinction will also be communicated with them in the first lecture.

3.2.1 Main learning objectives

As mentioned, main learning objectives are the fundamental concepts of functional programming and knowing them is a must for every functional programmer. Main learning objectives are function purity, composition and immutability. In this section each will be briefly described and justified.

Functional programming has its roots in mathematics and thus the concept of a pure function is a central idea in this paradigm. The ability to discover pure functions and differentiate them from impure functions belongs in every functional programmer's toolkit. This ability allows us to construct more robust programs. Separating pure from impure code is a good way to create structure in the code base and eases the discovery of bugs. In order to build valuable and practical programs - side-effects are necessary. Functional purity allows us to draw a line between the two worlds despite the paradigm we might be using. The end goal for every

software developer and programmer is to write code that is correct functionally and maintainable in the longer term. Pure functions are one tool to achieve this.

Composition plays a big part in functional programming. Functional purity allowed us to divide our programs between pure and impure functions (grand scale). Composition enables us to divide one function into smaller functions (granular scale) making code more reusable and easily testable. Composition allows us to break problems down into smaller chunks and tie them back together. Composition is not only applicable to the functional programming paradigm. Composition could easily be used in other paradigms as well. For example, so-called dot-chaining is also a form of composition and is used heavily in non-functional languages.

Data is the blood of the programs and thus it is crucial to handle it with care. Immutability helps to achieve this by giving programmers a framework and guidelines to work with data structures. An entire group of bugs can be eliminated if the data in the program is handled by following the rules of immutability.

3.2.2 Secondary learning objectives

There are many secondary learning objectives in this course. Listing all of them here is not reasonable. Secondary learning objectives include theoretical concepts like higher-order functions and currying and practical techniques like Elm architecture and reactive programming.

After successfully completing the course, students should be able to explain the concepts described by the main learning objectives using their own words and without the help of external resources. Secondary learning objectives are not meant to be studied by heart - rather knowing about their existence is enough to proceed. Once the need arises to solve problems, then it is only a matter of reminding them. Also, some secondary learning objectives include complex topics. Getting proficient in them requires hard work. The course's size is 3 EAP so there is not enough time to cover some other complex topics. Category theory and algebraic data structures are good examples of this. There exist specialized courses to study them. This course covers these topics without going too much into theoretical details and tries to be as practical as possible.

3.3 High-level overview of the course

This section is about giving a high-level overview of the course.

3.3.1 Administrative considerations

The course will be worth 3 EAP on successful completion. It means that the amount of work required by the student to get a positive result would be 78 hours. There will be eight lectures and take a total of 12 hours. The individual work portion should be done in 66 hours. Initially, there are no practice sessions planned. In the lectures, both theoretical fundamentals, as well as practical assignments will be discussed.

Lectures might take place biweekly or weekly depending on when the course starts and what is the duration of the course. Two weeks are considered to be the time window within which

both home assignments and the project should be completed. Announcing the grades for each assignment should be done within one week.

3.3.2 Lectures

In the lectures, new topics are introduced and explained. Functional programming concepts are introduced with code examples on slides to illustrate and give intuition on problems that otherwise might seem too abstract. Most of the examples on the lecture slides are contrived and meant mainly for illustrative purposes but still try to mimic real-world problems. Code examples are designed to be small enough to fit on the slides. Some of the code examples from lectures (from slides or practical sessions) will be used in the home assignments where students have to extend or simply use predefined functions. In the thesis text, some code examples from the slides are brought out in Section 4.2.

3.3.3 Home Assignments

Access to home assignments is granted at the end of lectures when they are briefly described. Home assignments are designed to be as practical as possible. In some weeks, students have to build small working applications. In some weeks, there are individual exercises. Exercises can depend on each other. Techniques used in previous weeks might be used in the following weeks.

3.3.4 Final Project

The final project aims to use the knowledge gained throughout the course to build a larger and functionally working web application. In the home assignments, small applications were built using various functional programming techniques. In the project, the same ideas and techniques will be used, but on a larger scale. Section 4.4 lists the proposed project ideas and requirements - both functional and non-functional - that the application must have to get maximum points for the solution.

The total score for a project is 20 points. Ten points for correct implementation of features (functional requirements) and 10 points for satisfying all non-functional requirements. It is possible to earn two bonus points by implementing a bonus feature.

3.3.4 Assessment

There are seven home assignments and a final project. The first week's home assignment is introductory and does not contribute to the total score as much as other home assignments and is worth 2 points. Oher home assignments are worth 8 points each. No threshold needs to be exceeded in order to get positive results for home assignments. A general formula for computing the final score for the course is following:

2 pts + 6 x 8 pts + 20 pts = 70 pts

The grade is computed using the following table:

Grade	Points	Pass / Fail
A	70 - 63	Pass
B	62.99 - 56	Pass
C	55.99 - 49	Pass
D	48.99 - 42	Pass
E	41.99 - 35	Pass
F	< 35	Fail

Table 1. Grading system.

4. Course materials

As described in the introduction section - this thesis' primary goal is to develop a course about functional programming in JavaScript. It includes teaching material: slides, optional reading resources, teaching plan for the lectures, assignment descriptions, project setup and automatic test suites for the assignments.

This section covers materials created as a part of the thesis. It goes over the delivery process of the course, explains topics described in the lectures and gives an overview of home assignments and specifies a final project requirements.

4.1 Delivery process

As described in Section 3.3, the course format is traditional in the sense that there are lectures for explaining topics and home assignments for assessing and checking the comprehension of the concepts introduced. Details discussed in this section are illustrative and in practice may change.

Table 2 describes resources and documents used in the course. References to developed materials are noted in the Appendices.

Resource	**Description**
Lecture Slides (Google Slides)	Slides to use in the lectures.
Home assignment source-code (Github)	Home assignments for each week can be found from a single Github repository. Each home assignment will have its branch (i.e., `week-2-assignments`).
Course Wiki (Github)	It contains all necessary information about the course and links to course resources.
Home assignment submission form (Google Forms)	It is used to submit home assignment solutions. A solution in .zip format and Student ID are required for a valid submission.
Grades (Google Sheets)	This file will contain grades for each home assignment and project. It will be used to publish the grades and updated after each home assignment has been graded. It also contains feedback for the home assignment solutions and final grades of the course.

Table 2. Course documents.

4.2 Lecture materials

This section describes developed materials and gives an overview of topics covered in lectures and learning objectives associated with each lecture. The order of topics may change, and the lectures' content could change as additional topics might be added to the course. The

lecture slides are the main assets of the course. The slides were created using Google Slides software. The descriptions of lecture materials are best to read together with lecture slides to get the full context needed to follow the text. There are multiple references to code examples presented on the slides, although some examples are brought out also in the text. Slides are subject to change, but things presented in the text are meant to stay intact and not change drastically. Links to lecture slides are given in Appendix 1.

Lecture 1: Introduction

The first lecture is about introducing the course, describing the format, goals and administrative details such as how the assignments are submitted and assessed and how the final grade is computed. Also, a brief overview of the history of functional programming and motivation about learning functional programming using JavaScript is given. The distinction between main and secondary learning objectives is described. It is stressed that in order to be successful, it is required to have basic programming skills. A small introduction to the most heavily used JavaScript language features is given, including defining functions using arrow-syntax and `function` keyword. One of the main goals of the lecture is setting up a development environment for the home assignments and a project. Students should have downloaded the source code at the end of the lecture and successfully started the application.

Learning objectives:

- Understands background behind functional programming.
- Understands how functional programming differs from imperative and object-oriented programming paradigms and what are its strengths but also weaknesses.
- Understands what is considered a good programming style in functional programming.
- Knows how to access course resources and how to set up the course development environment.
- Knows how the course is structured and what is the course's success of criteria.

Although it might make sense to start defining a pure function and describing the nature of functions in the first lecture, it was decided to introduce those concepts in the following week. The idea is to gradually introduce new concepts - in this lecture, introduce different ways to define functions in JavaScript. The following lecture builds upon that with the pure function definition.

After the initial introduction has been made, the lecturer describes the importance of functional programming in modern software development. Since the courses' programming language is JavaScript, it is worth explaining why industrial front-end development is affected by the emergence of functional programming practices.

In the standard computer science curriculum, functional programming is considered an advanced topic and is taught after students have already experience in imperative and object-oriented paradigms. In functional programming, there are different concepts that require students to unlearn previously acquired principles. Explaining differences to students is vital to align the assumptions about the learning process.

Since JavaScript knowledge is not a prerequisite in the course, various ES6 concepts - arrow functions, destructuring, modules, the difference between `let` and `const` - are covered in

this lecture. Students' skills may vary and for those who feel they are not experienced enough, the lecturer provides links to resources to learn about JavaScript basics.

In practical assignments, JavaScript is used to create and render HTML elements on the web page. In order to add styling to those elements with minimal effort, the lecturer suggests the TailwindCSS library, which is also included in the starter project template (*TailwindCSS*, s.a.). However, it is not required to add any styles in the home assignments. TailwindCSS library is briefly introduced in the lecture.

Lecture 2: Functional Programming Basics

The second lecture lays the groundwork for the whole course. It introduces pure functions and addresses the importance of the function in terms of functional programming. Some ideas (no loops, no mutation) might seem off-putting at first glance, but it is important to understand that this strictness comes with the value of constructing programs with fewer moving parts.

Learning objectives:

- Can explain the following functional programming concepts and use them in practice:
  - Higher-order functions
  - Pure functions
  - Currying and partial application
- Knows how to program without loops using `map`, `reduce`, `filter`.
- Knows what side-effects are and make a difference between pure and impure functions.

A pure function is a function that does not perform any side-effect and simply accepts an input, does some work and returns an output. Having no side-effects is one requirement for a function in order to be pure. A side-effect is an action that happens during the execution of the function and can be non-deterministic. A side-effect can be mutating a state, performing IO operations, accessing a database, fetching remote data, or throwing an error and halting the program's execution. Any program that needs to communicate with the outer world cannot be side-effect free. In functional programming handling side-effects explicitly is considered good practice even if not all functional languages restrict them. For example, fetching a record from the database is considered a side-effect since it cannot be known beforehand if the record exists in the database or not. In Lecture 2 only the definition of side-effect is given. How to handle them functionally will be explained in Lecture 5 and Lecture 6.

As a small exercise to test the understanding a couple of functions defined in JavaScript are shown to the students and they have to decide whether a function is pure or not. After which, pure counter-parts of impure functions are presented to the students.

A higher-order function is a function that takes a function as an argument or returns function as a return value. It is important to emphasize that in functional programming, a function is a value like any other - in other words, it is a first-class citizen in the language and can be treated as a value. This means functions can be passed around like variables. They can be saved to variables, passed to, or returned from other functions. The three most common higher-order functions for arrays are introduced - `map`, `filter` and `reduce`. Here it is stressed that these functions belong to `Array` and work purely on `Array` instances. These

three functions belong to every functional programmer toolkit and are the primary tools to avoid imperative coding style.

Lastly, the concept of currying is explained. Curried functions accept arguments one by one, every time returning a new function waiting for the next argument until all arguments are provided. In most functional programming languages, all functions are curried by default but not in JavaScript. In JavaScript, it is possible to simulate currying - either by defining unary function returning new function or using the helper function which turns regular functions into curried. Techniques on how to curry functions are presented and explained why currying is helpful in the functional programming paradigm. Data as the last argument principle is introduced and explained how this works well together with currying.

Lecture 3: Function Composition

Composition is introduced and an example of how to refactor a function written in imperative style into declarative style is shown. It is argued why functional programming favors declarative style and how it is beneficial for code quality. Definition of `compose` for two unary functions are given. The point-free programming concept is introduced with its pros and cons discussed. It is explained that dot-chaining is also a form of function composition. The concept of currying is introduced and how to leverage this to write more compact code and the difference with partial application.

Learning objectives:

- Can explain the following functional programming concepts and use them in practice:
  - Function composition
  - Point-free programming
  - Currying and partial application
  - Dot-chaining
- Understands difference between imperative and declarative coding styles and why to favor declarative style over imperative in most cases.
- Can explain why function composition is useful and compose functions using `compose` and `pipe`.

In functional programming, function composition is in the central position. Composition is used to piece together small functions to form new functions. It can be used to break down the program into smaller, more digestible and understandable parts which are easier to test and work with. Composition term is used throughout the course in various contexts and students must understand its importance and usefulness.

Composition is explained using Lego bricks as an example. Lego bricks express composition in the real world, one lego brick can be composed to another to form a new Lego brick. Composition is to compose two (or more) things to create another, the same type of thing (Lonsdorf, 2015, ch05).

After that simple definition of `compose` for two unary functions is given (Figure 1).

```
const compose = (f, g) => x => f(g(x))
```

Figure 1. Definition of compose function.

This definition is only used for explaining the concept of composition. In practice (and in home assignments) a `compose` function from Ramda library is used (*Ramda*, s.a.).

In the lecture, the above definition of `compose` is used to explain, step by step, how to transform imperative `sumEven` function (Figure 2) to declarative one (Figure 3).

```
const sumEven = xs => {
  const evens = xs.filter(x => x % 2 === 0)
  return evens.reduce((acc, num) => acc + num, 0)
}
```

Figure 2. Imperative sumEven function.

```
const sumEven = compose(sum, getEven)
```

Figure 3. Declarative sumEven using compose.

In Figure 3, definitions of `sum` and `getEven` function are omitted for the sake of thesis brevity. However, in the lecture slides they are presented and it stressed that using small, reusable and easily testable functions to compose other larger functions is useful in practice. It is important to pay attention to this since at first it seems refactoring to use a `compose` produces more code than before.

Dot-chaining is derived from the example above which is also a form of function composition (Figure 4). Dot-chaining is explained since it is used in practice and might be more natural to read (top-down, left-to-right evaluation) and is made easy in JavaScript. However, drawbacks of dot-chaining are explained: not being polymorphic and dependent on the instance methods of the underlying object (e.g. `xs` is `Array`). After this, the definition of `pipe` is given which is essentially a `compose`, but the evaluation of function is reversed (first `f`, then `g`).

```
const sumEven = xs =>
    xs
    .filter(isEven)
    .reduce(add, 0)
```

Figure 4. Dot-chained version of sumEven.

`sumEven` definition in Figure 3 is point-free. Point-free means defining a function without naming data it is meant to work on. For counter-example `sumEven` in Figure 4 accepts an array of numbers which is named `xs`.

Functional programming has its roots in mathematics and because of that the functions obey mathematical laws. Students are presented with different mathematical laws that functions and other structures must obey to be mathematically (and functionally) sound throughout the course. In this lecture, the composition's associativity law (Figure 5) and distributivity of `map` over `compose` (Figure 6) are introduced.

```
compose(f, compose(g, h)) === compose(compose(f, g), h)
```

Figure 5. Associativity of compose.

Associativity guarantees that it does not matter how we compose functions internally, so it is safe to compose functions however is more reasonable.

```
compose(map(f), map(g), map(h)) === map(compose(f, g, h))
```

Figure 6. Distributivity of map over compose.

The distributivity law tells us that we can safely extract calls to `map` inside the `compose`.

These laws make it easier to change programs since it is possible to rely on proven properties that those operations hold. It is hard to find similar laws in object-oriented programming.

There is a necessity to have the possibility to debug functions using `compose`. At first sight, it seems complicated to put a logging statement into the composition chain to be used for debugging purposes. It is not a problem in imperative programming since programs are made of statements and there is a "space" to insert additional statements. Functional programs are mostly made of expressions and focus on what should be done instead of how. The solution is to turn the debugging statement into an expression to plug into the composition chain (Figure 7). This is also a good example of the differences between imperative and functional programming.

```
const log = curry((tag, x) => (console.log(tag, x), x))
const sumDoubleEven = compose(
    sum,
    log('after double'), // after double [ 4, 8, 12 ]
    map(double),
    log('after getEven'), // after getEvent [ 2, 4, 6 ]
    getEven
)([1, 2, 3, 4, 5, 6])
```

Figure 7. Debugging expression.

The lecture is concluded with a summary stressing that composition is the most important design pattern in functional programming. The composition has many beneficial side-effects it brings to programs and those will be addressed in conclusion.

Lecture 4: Data & State

In this lecture, the immutability of the data is discussed and the reasons it matters are described. Different types of immutability are discussed with examples. Native JavaScript techniques for immutably handling JavaScript data structures are presented. Also, how to work with deeply nested object literals using spread operator and functional lenses. JavaScript is mainly used for adding interactivity to web applications. State handling is important and for any non-trivial application a necessary part of the architecture. The model-view-update (MVU) pattern is described to manage the application state.

Learning objectives:

- Understands what is immutability and what are its strengths.
- Knows what are problems with mutable data and application state.
- Knows how to handle data structures immutably.
    - Using native JavaScript methods and spread operator.
    - Using functional lenses.
- Understands model-view-update pattern and knows how to use it in practice.

Functional programming pays much attention to how data and the state of the program are handled. The main purpose of this lecture is to give students an understanding of how important it is to handle data with care and teach techniques for that.

The concept of immutability is essential in functional programming. In many functional programming languages, data structures are immutable by default and mutating them is made hard or is even impossible. Immutability is a property of data and means that it is impossible to modify the contents of data structure after its creation. In order to change, the initial data structure has to be copied and apply changes during the creation process.

In JavaScript, all primitive data types are already immutable. Primitive data types in JavaScript are `number, string, bigint, boolean, undefined, symbol`. It is called primitive immutability and this fact helps to keep programs immutable since it is possible to be sure the most primitive values cannot be changed.

In JavaScript, there also exists assignment immutability. In ES6 the `const` keyword was added to the language. The `const` keyword prevents binding a variable to a new value after it is already bound to something. This creates a shallow effect of immutability which may confuse beginners and thus it is important to explain this. Also, in the course code examples, `const` is being used heavily.

After explaining primitive and assignment immutability, the value immutability is introduced. It is stressed that in functional programming it is important to treat data as immutable to keep functions pure and free of side-effects. One of the side-effects is mutating the global state. Treating data structures immutably it is possible to ignore one type of side-effect and source of possible bugs. Examples of methods that are immutable are given. Reasons why immutability is an important concept to follow in functional programming, are listed and each reason is explained in more detail.

After knowing the benefits of immutability different practical techniques are reviewed on how to achieve immutability in programs and how to work with data structures in an immutable way.

ES6 added spread operator (`...`) to the language. Using this is a native way to update data so that it is immutable. The spread operator works both with arrays and the object literals. Small and illustrative examples are given on the lecture slides using the spread operator to update, add or remove attributes on the object literal (Figure 8).

```
const todoItem = {
  title: 'Change tires',
  status: 'undone',
}

// Updating attribute
{ ...todoItem, status: 'done' }

// Adding attribute
{ ...todoItem, dueDate: '2020/11/05' }
```

Figure 8. Updating and adding attributes using spread operator.

Problems with the spread operator are addressed after it is introduced. Firstly, it is easy to add a new attribute or element to an object or array and update the value under the key in the object on the top level. It becomes cumbersome to work with deeply nested data and makes reading and changing code hard, leading to errors. Secondly, the spread operator is language-specific. Although there may exist similar operators in other languages, it is still not a universal approach. Examples are given how using the spread operator for nested data structures quickly becomes hard to read and modify.

As a solution to address problems with the spread operator, the concept of lenses is introduced. Lenses give the possibility to "zoom" into the specific part of a data structure, apply an arbitrary number of operators on that part without losing the overall structure of the whole data structure. Lenses are expressed as functions and this makes them easily composable and reusable (Elliot, 2018).

Examples of how to use lenses using lenses implementation from the Ramda library is given. It is shown how to add and update data inside deeply nested data structures and how to compose lenses (Figure 9).

```
const R = require('ramda')
const todoItem = {
```

```
  title: 'Change tires',
  status: 'undone',
  tags: ['urgent', 'car'],
}

const tagsLens = R.lensProp('tags')
R.view(tagsLens, todoItem) // [ 'urgent', 'car' ]
R.set(tagsLens, [], todoItem)


// { title: 'Change tires', status: 'undone', tags: [] }
R.over(tagsLens, R.map(R.toUpper), todoItem)
// { title: 'Change tires', status: 'undone', tags: [ 'URGENT',
'CAR' ] }
```

Figure 9. Lenses example.

In modern front-end development, reacting to state changes has become a central design principle. This far into the course, pure functions have been used to render HTML elements on the web page and in order to see how changes to data affect the view, it was necessary to do a full browser reload. Since browsers run JavaScript natively, it is possible to eliminate full reloads and reactively act on the state changes and cause views to update. There are multiple ways to achieve reactivity of the views, but as the description of the problem already hints, it brings several complexities to the application development. In order to address this complexity, a concrete design pattern was invented - Elm architecture (Czaplicki, 2012). Eventually, this pattern was popularized by Redux - state management library for JavaScript programs, mainly meant to be used with React view library. In the course, the model-view-update pattern is used to give a name to the pattern. The idea of MVU is simple - the current state of the application (model) is passed to a function (view) that renders UI based on the model. From the view function, it is possible to dispatch messages which are handled by the update function. The update function is a pure function that accepts a dispatched message and the application model returning a new model. The message tells the update function how the model should be updated (Figure 10).

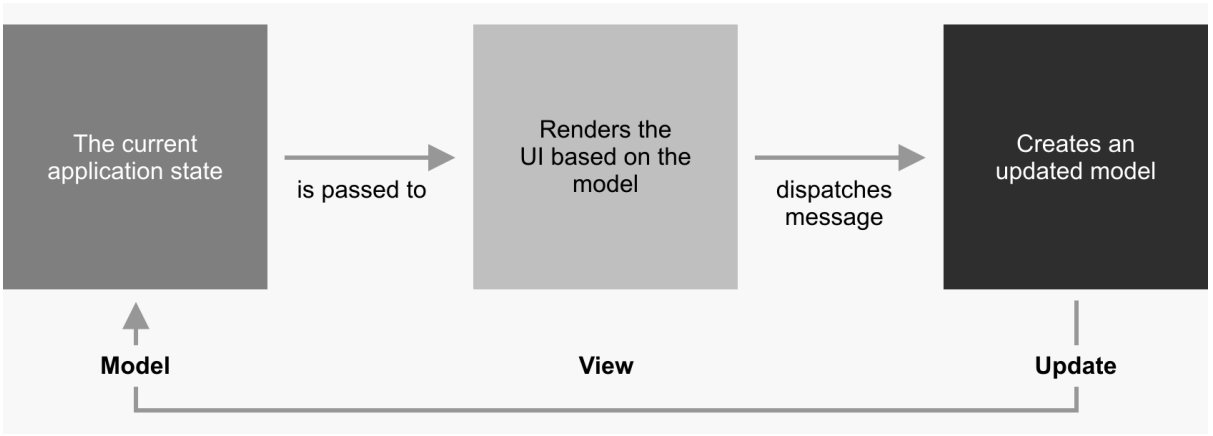

Figure 10. Model-view-update (Bandt, 2020).

After explaining MVU pattern, short code snippets are shown on the lecture slides how this pattern is implemented in practice. An example is a simple number counter. There are buttons for incrementing and decrementing counter value with an initial state of 1.

The lecture will explain that the MVU pattern is used in the popular Redux state management library. Understanding how it is built from scratch benefits understanding Redux's internals.

Lecture 5: Introduction to Algebraic Data Structures

In the fifth lecture, the journey of discovering algebraic data structures begins. Firstly, an introduction to category theory is given. After which, ambiguity with names of different functional programming ideas is resolved. Extended Hindley-Milner type-notation is explained and how and why they are used in this course and other functional programming resources. Several algebras from Fantasy Land Specification are introduced. These introductions include definitions of the algebras with methods, type signatures and mathematical laws. The intuition behind every algebra is explained. Some practical and illustrative examples are presented on the lecture slides.

Learning objectives:

- Knows the background of Fantasy Land Specification.
- Can read extended Hindley-Milner type signatures for functions.
- Understand the intuition behind following algebras:
    - Semigroup
    - Monoid
    - Functor
    - Chain
    - Apply
    - Applicative

Students have been familiarised this far into the course with functional programming basics: pure functions, higher-order functions, composition, immutability. Side-effects were explicitly handled by invoking impure code at the boundary of the program. This is fine if side-effectful code is small and handling it will not be a big problem. Once the codebase increases in size, handling side-effects becomes cumbersome. It would be good to test this code and be in charge of when and how it is called.

Another motivation for this week's topics is to provide a general and standard interface for objects. Having standard interfaces and laws those objects have to obey enables programmers to work with confidence without worrying about specificities and focusing on the programs' important bits.

The interface and laws referenced above are coming from category theory. Category theory is a general theory for formalizing relations between objects (*Category Theory*, s.a.). Category theory is only briefly mentioned in the lecture to share the background. In the JavaScript world, category theory concepts are specified in the Fantasy Land Specification.

Fantasy Land Specification uses extended Hindley-Milner type notation to annotate function signatures. JavaScript is not a strongly typed language and type signatures are written using comments above the function definitions. Type definitions specify types of return values and give type constraints function arguments. In order to be valid in the sense of Fantasy Land Specification functions must follow specific type signatures. Although the compiler does not automatically check types and it may seem redundant and cumbersome to read and write them, then actually many functional programming resources and libraries (Ramda) use them.

The structures in Fantasy Land Specification are called algebras. In different resources different names are being used. For example, in Haskell they are called type classes. This ambiguity is bad since it confuses beginners. In this course term "algebra" is used to reference specifications in Fantasy Land Specification. The term "algebraic data structure" is used to reference the concrete implementation of an algebra.

The first algebra introduced is a semigroup. A semigroup is any object with an associative `concat` method. The type signature for `concat` method is following (Figure 11).

```
concat :: Semigroup a => a ~>[1] a -> a
```

Figure 11. Semigroup concat type signature.

Semigroup has to follow associativity law over the `concat` method. This law states it does not matter how we combine semigroup instances; the result is still the same (order is still important, i.e., not commutative).

```
a.concat(b).concat(c) === a.concat(b.concat(c))
```

Figure 12. Semigroup's associativity law.

It is crucial to give intuition about different algebras. Semigroups allow us to combine two similar things with a common interface. It is a simple idea but proves to be powerful (Williams, 2015). In JavaScript strings and arrays are valid semigroups having `concat` method (Harding, 2017). It is possible to construct custom semigroups. In the lecture slides an example is given for `Sum` semigroup which captures an addition over numbers (Figure 13)

[1] Squiggly arrow represents the instance method. Its left hand operand is an instance of type a and the right hand side is type of instance method's return value (a -> a, in this example).

```
const Sum = x => ({
  val: x,
  concat: other => Sum(x + other.val),
})
```

Figure 13. Sum semigroup.

All algebraic structure definitions used on the slides are for illustrative purposes and might not be valid implementation in terms of Fantasy Land Specification and in the more advanced examples the structures from an external library are being used.

The next structure introduced is called a monoid. It builds on the semigroup definition. The only thing it does, it specifies an additional method called `empty` (Figure 14).

```
empty :: Monoid m => () -> m
```

Figure 14. Monoid empty type signature.

Again, the type signature and method definition on their own are not useful to understand the intuition behind monoids. If semigroups allowed us to combine one or more things, then monoids allow us to combine zero or more things (Harding, 2017, #5).

The nextstructure being introduced is a functor. The intuition behind functor is a function application. Functors allows applying a value to a function. Functor has a `map` method with the following signature (Figure 14).

```
map :: Functor f => f a ~> (a -> b) -> f b
```

Figure 14. map type signature.

Valid functors have to follow functor laws (Figure 15). JavaScript native arrays are functors having a `map` method. This allows us to apply values inside the array to a function.

```
// Identity
map(id, F) === id(F)
// Composition
compose(map(f), map(g)) === map(compose(f, g))
```

Figure 15. Functor laws.

On the lecture slides, an example of `Maybe` functor is given to illustrate how to use `Maybe` to retrieve a value from the object literal. This example is contrived but helps to illustrate the practical purposes of the functors.

The journey of discovering different algebraic structures continues with looking into a chain structure. A structure implementing chain algebra has a `chain` method. Other names for a `chain` method used in different resources and programming languages are `bind`, `flatMap`, `collect`. Chain is a bit similar to a functor. Looking at its type signature (Figure 16), it is a good opportunity to test students' understanding and explain the intuition behind the `chain` method.

```
chain :: Chain m => (a -> m b) -> m a -> m b
```

Figure 16. chain type signature.

The chain allows us to apply a value to a function similar to the functor, but this function has to return value wrapped within a chain. In practice, this means we can chain functions without worrying about getting stuck with deeply nested structures. Again, arrays are good candidates to illustrate the idea behind chain since its `flatMap` method is equivalent to `chain` method.

Also, concrete algebraic structures may implement different algebras. We have seen that arrays are semigroups and functors. A `Maybe` structure introduced earlier is also chain structure after adding `chain` method (Figure 17).

```
const Just = x => ({
  map: f => Just(f(x)),
  chain: f => f(x),
})

const Nothing = () => ({
  map: _f => Nothing(),
  chain: _f => Nothing(),
})
```

Figure 17. Maybe chain.

Retrieving value from deeply nested object literal is a good example of the practicality of `Maybe` chain structure.

Apply is an algebra that enables to apply a value to a function wrapped in the context. The contexts in both cases have to be the same. Apply specifies `ap` function (Figure 18).

```
ap :: f (a -> b) -> f a -> f b
```

Figure 18. ap type signature.

Students may ask what the use case is? How often do we have functions wrapped in the context? That is a valid question as usually programs are being constructed with regular functions meaning they do not operate on the values within contexts. There exists a family of functions that allow us to lift regular functions to work with wrapped values. Figure 19 defines `lift2` for lifting binary function. There exist functions for lifting functions with different arities, `lift3` for ternary functions for example. Note that binary function has to be curried.

```
const lift2 = f => a => b =>
  b.ap(a.map(f))
```

Figure 19. lift2 function.

It is important to illustrate this with an example. Figure 20 shows how lifting a regular function can be used to add together two numbers wrapped within `Maybe`.

```
lift2(x => y => x + y)(Just(2))(Just(3)) // Just(5)
lift2(x => y => x + y)(Nothing())(Just(3)) // Nothing
lift2(x => y => x + y)(Just(2))(Nothing()) // Nothing
```

Figure 20. Lifting function.

The next structure being introduced is applicative. An applicative specifies `of` function which lifts arbitrary value into the context (Figure 21). The intuition for applicative is similar to monoids. As Tom Harding (2017, #9) explains in his article series, applicative make it possible to combine zero or more contexts. In contrast, monoids enable to combine zero or more values. Lecture slides have an example to illustrate this concept.

```
of :: Applicative f => a -> f a
```

Figure 21. of type signature.

Lastly, a monad is defined. Monad is a structure that defines both chain and applicative and follows monad laws defined in Figure 22.

```
// Left identity:
M.of(x).chain(f) === f(x)
// Right identity:
M.chain(of) === M
// Associativity:
M.chain(f).chain(g) === M.chain(x => f(x).chain(g))
```

Figure 22. Monad laws.

Lecture 6: Algebraic Data Structures in Practice

This lecture goes more in depth of how to use algebraic data structures in practice for common problems encountered when building real-life applications. Practical uses of `Maybe`, `Either` and `Result` monads are explained. `Async` monad is introduced and how to use it to model asynchronous operations. It is explained why Promises do not comply with Fantasy Land Specification.

Learning objectives:

- Understands what is Monad and intuition behind it.
- Knows how to use algebraic data structures in practice:
    - Maybe monad for handling missing values.
    - Either monad for handling errors.
    - Result monad for conditional branching.
    - Async monad for asynchronous computations.

In Lecture 5 there were a couple of brief examples how to use `Maybe` to handle retrieving the value of property from object literal. In this lecture similar practical uses for `Either` and `Result` structures will be reviewed. There will be no definition of those structures explicitly brought out on the slides as it were for `Maybe` structure in Lecture 5. This lecture's main focus is on how to use algebraic data structures in practice.

In order to understand how to handle asynchronous code using `Async` monad it is useful to understand how asynchronicity is handled in JavaScript in general. JavaScript's event based execution model is described briefly following with examples how to use callbacks and how this leads to something called "Callback Hell". Promises will be introduced and explained how they are a solution to explicitly handle asynchronous computations, but unfortunately in the context of this course they are not "fully functional". This means they are eager in their execution and do not adhere to monad laws. In order to work with asynchronous code in a functional manner, `Async` monad is introduced and differences with Promise are explained. Slides contain contrived examples of how to refactor code which uses callbacks to use Promises and then `Async` monad. `Async` monad is discovered further and showed how to use it with `lift2` function defined in Figure 19 to run asynchronous functions in parallel.

Lecture will be concluded with a discussion over advantages and disadvantages of using algebraic data structures in practice. In the author's opinion algebraic data structures have a steep learning curve and require quite an effort in order to understand all the caveats they introduce. The students are also advised to exercise caution before introducing them into

already mature projects because the rest of the system may not be designed to work with monadic interfaces.

Lecture 7: Functional Reactive Programming

Lecture 7 explores functional reactive programming (FRP). The model-view-update pattern described in Lecture 4 was one example of reactive programming, but FRP lifts the idea of reactivity to another level. It describes that data is passed through the program using streams and defines a set of operations to create and manipulate those streams. Sources of streams can be button clicks, data from remote HTTP requests, or simply an array of numbers. Main idea of FRP is composition of streams. As a regular function is a first class value type in functional programming, a stream is a first class value type in FRP.

Learning objectives:

- Knows what is FRP and its use-cases in real world applications.
- Knows how to use FRP techniques in practice.
- Understands advantages and disadvantages of FRP.

The lecture starts with exploring what reactivity is. Reactivity is illustrated using different examples. One example is a spreadsheet cell. It is possible to define the cell's value as a formula and make it depend on another cell's value. If the value of another cell changes this change is propagated to the cell with formula and value in that cell is updated accordingly.

Next, stream as a concept is explored. A stream is a sequence of ongoing events ordered in time. There are three event types: a value, an error, or a "completed" signal. Actions are taken on these events asynchronously. This act of listening on a stream and reacting to emitted events are called subscribing, similar to the model-view-update pattern.

An Observer and an Observable are defined. An Observer is something that listens to an Observable instance. An Observable is a primitive value for modeling streams. Observable contains API to create, subscribe to and to transform streams. Observables and streams are not the same thing. A stream is a high-level concept of time-based events. Observable is a notion of streams in code.

RxJS library is used in the course for practicing FRP (*RxJS*, s.a.). It is a popular library in the JavaScript ecosystem to do event-based asynchronous programming. For example, the Angular web application framework includes RxJS to manage application state and do side-effects.

The lecture starts with exploring FRP and RxJS using a simple number counter example already familiar from Lecture 4. On Figure 23 the full code is given, but on the slides this will be presented step-by-step. This example demonstrates creating streams of button clicks using `fromEvent` function, transforming events using the `map` function, merging two streams, calculating current state based on events using the `scan` function and finally, subscribing to the stream to perform a side-effect - changing text value of an HTML element.

```
import { fromEvent, merge } from 'rxjs'
import { map, scan } from 'rxjs/operators'

const $id = (selector) =>
```

```
  document.getElementById(selector)

const upStream = fromEvent($id('up'), 'click')
  .pipe(map(_ => 1))
const downStream = fromEvent($id('down'), 'click')
  .pipe(map(_ => -1))

merge(upStream, downStream)
  .pipe(scan((acc, cur) => acc + cur, 0))
  .subscribe(text => $id('counter').textContent = text)
```

Figure 23. Counter example using FRP.

RxJS has operators similar to plain JavaScript `Array`. For example, it is possible to map and filter events or concat them. RxJS documentation has visual representation in the form of marble diagrams to illustrate different operators' behavior. Documentation also includes code examples.

Next, it is shown how to use FRP together with a model-view-update pattern. Same counter example will be used, but instead of assigning event listeners directly to buttons in the `view` function, event streams for both buttons will be created and callbacks in `subscribe` method will dispatch the necessary actions. This requires that both buttons are already rendered on the page. It will not be a better solution compared to direct event listeners, but exists for showing that FRP can be used together with other patterns and is framework agnostic.

Another example is used to show how to use FRP to create a typeahead system. A typeahead system allows the user to type something into the text input and user interface will suggest possible options as the user types. This example is a step further from the previous one and shows how to switch from one stream to another depending on the value of the first stream. In FRP it is essential to get the feeling for how streams work and how they can be combined.

Next example will build upon the previous one. Instead of querying data from static data structures defined in code, it will be fetched using a remote HTTP request to the external API endpoint. Data will be parsed and rendered as the user types. This demonstrates how to create streams of external data sources and how to make sure API won't be queried too much. With that, the students should be equipped with skills to solve home assignments.

The lecture concludes with a discussion of advantages and disadvantages of FRP. FRP is a powerful idea abstracting programming with time-based event series, but as always there are drawbacks to be considered. For example, RxJS adds complexity and additional dependency to the project. Also, in the author's opinion FRP has a moderate learning curve which means new developers have to learn the paradigm in order to be productive and successfully contribute to the project.

Lecture 8: Preparation for Project & Wrap-up

The last lecture is more light on topics compared to previous lectures. This lecture has two goals: conclude the course and prepare students for solving the course project.

The selection of final project ideas is presented to students and requirements are explained in terms of what is important to score maximum points for the project. It is explained that the project is used instead of a written exam as this aligns better with the course goals. The project is used to assess both theoretical knowledge and practical skills students have learned during the course. All project ideas are meant to be implemented using techniques used in the course and this is why there are both functional and non-functional requirements specified. It is possible to score 50% of points for the project not using functional techniques implementing only functional requirements, but it is explained that this is strongly discouraged. The final project is meant to be completed within two weeks. It is explained how points for the project count towards the final grade.

The lecture continues with revising and giving a high-level overview of the topics covered in the course. There might be a simple and interactive quiz to test students' understanding of the topics. This quiz does not affect the final grade.

Other functional programming languages will be introduced. The introduction of the programming language includes its main features, differences to other languages, main use cases and short code examples. The purpose of this is to provide students with references to discover functional languages on their own and increase the interest in learning different languages. The languages being introduced are Haskell, Scala, Clojure, and Elixir.

As this course is about functional programming in JavaScript, which is mostly used to develop interactive front-end applications, it makes sense to explore related technologies separately. All languages mentioned in this paragraph compile down to JavaScript and can be used to develop browser applications. Firstly, Elm language is introduced and explained how its programming model has affected more popular libraries like Redux. Elm has a strong type system similar to Haskell but focuses mainly on developing front-end applications. Secondly, PureScript will be introduced and explained how it fits into the world of front-end application development and functional programming. PureScript has more features than Elm and it is a general-purpose language.

4.3 Assignments materials

Assignments are ordered and correspond to topics covered in the lecture with the same order number. For example, Home Assignment 1 covers topics from Lecture 1. There are multiple exercises in each home assignment. Some exercises are independent, some depend on each other. There are also unit tests for most of the exercises, excluding only those which require showing UI elements on the web page. Test suites are not meant for grading purposes. Rather, the goal is to guide the students through the process of solving exercises. This section gives an overview of each assignment without going too deep into technicalities. It is worth mentioning that exercises referenced in this thesis text are just examples and are subject to change, but the main ideas and overall structure of each assignment will be preserved. Exercises can be accessed from the public Github repository (Appendix 2).

Home Assignment 1

Home Assignment 1 is about validating that the students have successfully set up the development environment and can start solving exercises.

Home Assignment 2

In Home Assignment 2 students have to implement a couple of simple functions which will be later used in other exercises. Those may include writing already known functions like `map` from scratch. Later exercises are about transforming lists with functions defined in earlier exercises and writing functions to render HTML elements on the web page. At the end of exercises the students should have practiced writing pure functions and separating pure function calls from impure examples. Also, they should have experience using functions that return HTML elements and rendering data on the web page. An example of the function to build HTML elements will be given in the assignment description.

Home Assignment 3

In Home Assignment 3 students have to implement functions using `compose` (from Ramda or crocks libraries). Those functions will be used to filter lists of data and group by some arbitrary property value. Again, as a more practical exercise, students have to build an HTML table using pure functions and render that on the web page. There will be an exercise to filter and sort data in the table by providing filtering and sorting criterias via query parameters. A code example of how to read and parse query parameters will be given in the assignment description.

Home Assignment 4

In Home Assignment 4 students have to build a small application using already learned functional programming techniques. The project has an initial state and the render UI elements for that. It is possible to interact with UI elements to change state of the application and update UI based on the changes to state. Application has to use a model-view-update pattern.

Home Assignment 5

In Home Assignment 5 students have to use different algebraic data structures to solve exercises. They have to implement custom semigroups for already known operations like maximum number, number multiplication, etc. They will use `Maybe` to handle possibly missing values and `Either` to handle errors. They will have to write their own implementation of `apply` function for arrays.

Home Assignment 6

In Home Assignment 6 students have to use algebraic data structures to handle side-effects - fetching data from remote API endpoints, reading data from a file and writing data into a file.

Home Assignment 7

In Home Assignment 7 students have to use FRP and extend an existing application. They will have to use techniques learned in the lecture to react to user input and transform data streams.

4.4 The Final Project

This section lists proposed final project ideas with requirements - both functional and non-functional - that the application must fill to get maximum points for the solution.

The non-functional requirements are the same for every project idea. This list is not complete and there might be additional proposals. In practice the project ideas will be more detailed in order to communicate requirements and there will be stronger criterias to assess the solutions.

Non-functional requirements:

- All core logic should be handled using pure functions (2 points).
- Pure functions must have unit tests (1 point).
- There is at least one use of `compose` function (1 point).
- Updating data structures are handled immutably (1 point).
- Application state is handled with model-view-update pattern (2 points).
- There is at least one use of algebraic data structure (addition to `Async`) (1 point).
- Side-effects are handled by using `Async` monad (from crocks library) or FRP (1 point).
- Starting application has to be documented. All missing features have to be documented (1 point).

Project ideas

Project idea	**Functional Requirements**
Pomodoro app	● Possible to start a timer which starts countdown. By default the timer is set to 25 minutes. ● It is possible to change timer value. ● There exists buttons for resetting the timer value for 5 and 15 minutes. ● It is possible to pause and reset the ticking timer. ● For every timing there is a log created and shown on the page. The log should contain date and time range timer has been running (with exact clock values). ● It is possible to download a CSV export file for current timings. Each row in the file should contain the

	same data that logs have. ● BONUS: When timer finishes notify user with a notification and/or alarm sound.
Todo app	● It is possible to create todo items with a title, description and date. Todo item has also status, which is set to “not-done” by default. ● All attributes are required. App should show validation errors on invalid todo item. ● It is possible to change the status of todo items. Possible values are “not-done”, “in progress” and “done”. ● It is possible to delete a todo item. ● It is possible to filter todo items by status and date range. ● It is possible to download a CSV export file. Every row in the file should contain info about one todo item. All attributes are present. ● BONUS: Add possibility to group todo items by status or date. Every group’s todo items are shown in different columns.
Country and city weather fetching app	● There are two inputs, one for searching countries and another one for cities. ● Two inputs are dependent. Country value will determine values of cities. ● It is possible to select a city. For that city data including current weather will be shown in UI. ● City data will include population. ● Weather data will include temperature, description, humidity, wind speed. ● Countries data is queried from .json file. ● Cities data is queried from external API[2]. ● Weather data is queried from external API[3].

[2] https://countriesnow.space/
[3] https://openweathermap.org/

	• BONUS: Query and show weather data for the last 7 days.

Table 3. Project ideas with functional requirements.

4.5 Topics that were considered

This section will discuss topics that were left out from the initial course plan. They are worth mentioning here if the course is iterated on in the future, as they might be worth adding if the course is ever changed.

Closures are a way to define a function inside another function with access to the outer function's data. This data will be contained in the closure and can be used in other parts of the program. Closures are related to higher-order functions (as they use them internally). There are many practical use-cases for closures. Closure can be used to simulate classical objects meaning it is possible to associate behaviour with data. Closures could be covered in Lecture 2 together with higher-order functions or in Lecture 4 describing them as vessels for state and behaviour.

Transducing is an advanced topic in functional programming. Transducing can be considered a performance tuning technique. It allows composing reducers in order to run data transforming functions only once for every element in the data structure. Transducing is useful to use in case of large data sets or lazy data streams (observables). Kyle Simpson describes very well how transducing internally by deriving transducer from the first principles (Simpson, 2016, Appendix A). Transducing could be introduced at the end of Lecture 3.

Introduction to lambda calculus using JavaScript could be an additional topic to Lecture 8. As the lambda calculus will be mentioned during Lecture 1 describing background and history of functional programming, then there will be no time in Lecture 1 to dive deeper. Using JavaScript syntax to explain lambda calculus and derive Church encoding could be an exciting example of having only variables, functions and function application is enough to derive primitives for computation (*Church Encoding*, 2005).

Topics described above are not fundamental to functional programming and this the main reason why they didn't end up in the course topics. Also, transducing and lambda calculus are both more advanced topics and it didn't make sense to explain them to students who are at the beginning discovering functional programming.

5. Conclusion

The main goals for the thesis were to explain the reasoning behind the creation of the course and describe the materials that were developed for it. Several resources like the lecture slides, descriptions of home assignments and illustrative code snippets were developed. During the course development, the author reviewed multiple resources on the topic and explained the reasoning of the existence of such a course. The use of JavaScript as the chosen language for the course was justified. The author reviewed the background of functional programming in general and described the goals of the course and target audience. The goal of the course was to introduce functional programming in a practical setting. Administrative considerations for the course, like grading criteria and delivery process, were described as part of the thesis. In Chapter 4 the author described all the lecture topics in detail. Much effort was put into trying to find ways to explain functional programming concepts using practical examples. Code examples in the slides were used to pass on the level of practicality of the content and delivery process. Home assignments and a final project were designed to be practical and interesting as well.

After successfully completing the course the students should understand how functional programming fundamentals can be used in practice. They have practiced writing small working applications following functional programming guidelines and have had a chance to compare how writing programs in the functional paradigm affects code quality and the actual solution itself. Students are encouraged to continue discovering the world of functional programming. After finishing this course they should have all the necessary basic knowledge to do so.

References

Bandt, T. (2019, September 23). *Who Cares About Functional Programming?* Retrieved May 6, 2021, from https://thomasbandt.com/who-cares-about-functional-programming

Bandt, T. (2020, 02 09). *Model-View-Update (MVU) – How Does It Work?* https://thomasbandt.com/model-view-update

Braithwaite, R. (2013, April 8). *When FP? And when OOP?* https://raganwald.com/2013/04/08/functional-vs-OOP.html

Category Theory. (s.a.). In Wikipedia. https://en.wikipedia.org/wiki/Category_theory

Chiusano, P. (2016, September 15). *The advantages of static typing, simply stated*. Retrieved May 12, 2021, from http://pchiusano.github.io/2016-09-15/static-vs-dynamic.html

Church encoding. (2005). In Wikipedia. https://en.wikipedia.org/w/index.php?title=Church_encoding

Copes, F. (2018, November 19). *Should you use or learn jQuery in 2020?* Retrieved May 12, 2021, from https://flaviocopes.com/jquery/

Czaplicki, E. (2012). *The Elm Architecture*. An introduction to Elm language. Retrieved 04 26, 2021, from https://guide.elm-lang.org/architecture

Eich, B., & Wirfs-Brock, A. (2020). JavaScript: the first 20 years. *Proceedings of the ACM on Programming Languages*, *4*(Article No.: 77). https://dl.acm.org/doi/10.1145/3386327

Elliot, E. (2017, February 19). *The Rise and Fall and Rise of Functional Programming*. Retrieved May 6, 2021, from https://medium.com/javascript-scene/the-rise-and-fall-and-rise-of-functional-programming-composable-software-c2d91b424c8c

Elliot, E. (2018, December 24). *Lenses. Composable Getters and Setters for Functional Programming*. Retrieved April 26, 2021, from https://medium.com/javascript-scene/lenses-b85976cb0534

Elliot, E. (2019). *Composing Software: The Book*. https://medium.com/javascript-scene/composing-software-the-book-f31c77fc3ddc

Fantasy Land Specification. (s.a.). https://github.com/fantasyland/fantasy-land

Functional Programming Principles in Scala. (s.a.). Retrieved May 6, 2021, from https://www.coursera.org/learn/progfun1

Harding, T. (2017, 03 13). *Fantas, Eel, and Specification 4: Semigroup*. http://www.tomharding.me/2017/03/13/fantas-eel-and-specification-4/

Harding, T. (2017, 03 21). *Fantas, Eel, and Specification 5: Monoid*. http://www.tomharding.me/2017/03/21/fantas-eel-and-specification-5/

Harding, T. (2017, 04 17). *Fantas, Eel, and Specification 9: Applicative*. http://www.tomharding.me/2017/04/17/fantas-eel-and-specification-9/

Idris. (s.a.). Idris: A Language for Type-Driven Development. https://www.idris-lang.org/

Lonsdorf, B. (2015). *Professor Frisby's Mostly Adequate Guide to Functional Programming*. Gitbook. https://github.com/MostlyAdequate/mostly-adequate-guide

Major Programming Paradigms. (s.a.). https://www.cs.ucf.edu/~leavens/ComS541Fall97/hw-pages/paradigms/major.html

Miyata, S. (2017, August 25). *The Object-Oriented Programming vs Functional Programming debate, in a beginner-friendly nutshell*. https://medium.com/@sho.miyata.1/the-object-oriented-programming-vs-functional-programming-debate-in-a-beginner-friendly-nutshell-24fb6f8625cc

ML (Programming language). (s.a.). In Wikipedia. Retrieved May 12, 2021, from https://en.wikipedia.org/wiki/ML_(programming_language)

Most Popular Technologies: Programming, Scripting, and Markup Languages. (s.a.). Stack Overflow Developer Survey 2020. Retrieved May 6, 2021, from https://insights.stackoverflow.com/survey/2020#most-popular-technologies

Moutafis, R. (2020, August 5). *Why developers are falling in love with functional programming*. Retrieved May 6, 2021, from https://towardsdatascience.com/why-developers-are-falling-in-love-with-functional-programming-13514df4048e

Mundy, I. (2017, February 20). *Declarative vs Imperative Programming*. Retrieved May 6, 2021, from https://codeburst.io/declarative-vs-imperative-programming-a8a7c93d9ad2

Petricek, T. (2012). *Teaching Functional Programming to Professional .NET Developers*. University of Cambridge.

Programming Languages, Part A. (s.a.). Retrieved May 6, 2021, from https://www.coursera.org/learn/programming-languages

Ramda. (s.a.). A practical functional library for JavaScript programmers. Retrieved May 6, 2021, from https://ramdajs.com/

React. (s.a.). Retrieved May 6, 2021, from https://reactjs.org/

Redux. (s.a.). Retrieved May 6, 2021, from https://redux.js.org/

RxJS. (s.a.). Retrieved May 12, 2021, from https://rxjs.dev/

Scalfani, C. (2020, November 30). Why is Learning Functional Programming So Damned Hard? Retrieved May 12, 2021, from

https://medium.com/@cscalfani/why-is-learning-functional-programming-so-damned-hard-bfd00202a7d1

Simpson, K. (2016). *Functional-Light JavaScript*. https://github.com/getify/Functional-Light-JS/blob/master/manuscript/apA.md/#appendix-a-transducing

Sinclair, J. (2016, August 7). *The Marvellously Mysterious Javascript Maybe Monad*. Retrieved May 12, 2021, from https://jrsinclair.com/articles/2016/marvellously-mysterious-javascript-maybe-monad/

Sinclair, J. (2019, October 28). *Things I Wish Someone Had Explained About Functional Programming*. Retrieved May 6, 2021, from https://jrsinclair.com/articles/2019/what-i-wish-someone-had-explained-about-functional-programming/

TailwindCSS. (s.a.). https://tailwindcss.com/

Tan, G. (2004). *A Brief History of Functional Programming*. A Brief History of Functional Programming. Retrieved 2 4, 2021, from http://www.cse.psu.edu/~gxt29//historyOfFP/historyOfFP.html

Wikibooks:Textbook considerations. (s.a.). https://en.wikibooks.org/wiki/Wikibooks:Textbook_considerations

Williams, B. (2015, February 15). *Algebraic Structure and Protocols*. Retrieved May 12, 2021, from https://www.fewbutripe.com/swift/math/algebra/2015/02/17/algebraic-structure-and-protocols.html

Appendices

Appendix 1

This appendix lists links to lecture slides. Lecture slides are dynamic in nature and including them as links is appropriate. Links are public and anyone with a link access can view the slides.

Lecture 1

https://docs.google.com/presentation/d/1mONQ3JvjHRjSoLUxwDV3R46BVfUhS42N5I-cQcD-0OE/edit?usp=sharing

Lecture 2

https://docs.google.com/presentation/d/1OFLzPJzpA5Qd91ihdoBZWqxZW5LwGy35eYub94Jh8eo/edit?usp=sharing

Lecture 3

https://docs.google.com/presentation/d/1OFLzPJzpA5Qd91ihdoBZWqxZW5LwGy35eYub94Jh8eo/edit?usp=sharing

Lecture 4

https://docs.google.com/presentation/d/1crxwDgLdaDXT2N_xUGUs6dAFGTyaLnurYfek13BV3vc/edit?usp=sharing

Lecture 5

https://docs.google.com/presentation/d/1jtZ7EibzEx9pWvkTK3Gl4LsmM1YS1Bl_aGYluyJ0l_E/edit?usp=sharing

Lecture 6

https://docs.google.com/presentation/d/12XD5vurLKnBWnXqyrgkH4F1uzp0UJxDJsObiIbmCYDQ/edit?usp=sharing

Lecture 7

https://docs.google.com/presentation/d/12wwFW1EWS9s8CjhlkyxLOMZQPflsxC9RXGgtUxjxCOc/edit?usp=sharing

Lecture 8

https://docs.google.com/presentation/d/1n-YmDpJ8Nl6P1DDIuwAKvZxgq3JjvoRaE4CXInYfJFE/edit?usp=sharing

Appendix 2

This appendix lists links to home assignment descriptions. Home assignments source code with description live in a single Github repository and for each week there is a branch following naming convention: `week-<week_number>-assignments`. The `main` branch has README.md file documenting setting up the development environment.

Table 4 describes paths to files and their meaning.

Path	**Description**
src/assignments/week<week_number>/home-assignment-<week_number>.md	This file contains a description of assignment(s).
src/assignments/week<week_number>/index.js	Students will write their solution into this file.
src/assignments/week<week_number>/index.test.js (optional)	This file contains a test suite for assignments.
src/index.js	This file will be used to call functions defined in the solution file.

Table 4. Description of main home assignment files.

Main link to repository: https://github.com/kaareltinn/fp-js-starter

Appendix 3

Course has a Wiki page which lists all the resources. Lecture slides and home assignment descriptions will be announced biweekly. Wiki has links to Google Form to submit the home assignment solutions and to grades spreadsheet.

https://github.com/kaareltinn/fp-js-starter/wiki

License

Non-exclusive licence to reproduce thesis and make thesis public

I, Kaarel Tinn,

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,
   Developing a Course on Teaching Functional Programming in JavaScript, supervised by Margus Niitsoo and Vesal Vojdani.

2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.

3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.

4. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Kaarel Tinn

14/05/2021