Bachelor Thesis

Functional Standard Library for the Kolibri Web UI Toolkit

Andri Wild
Tobias Wyss

Bachelor Computer Science

Windisch, August 2023

University of Applied Sciences and Arts Northwestern Switzerland FHNW

Supervisors

Prof. Dierk König
Dirk Lemmermann

Abstract

JavaScript supports just a few array processing operations. Additionally, these operations are inconsistent: some modify the existing array, but others create a new one. The programming language Haskell provides a much more extensive and concise collection of such operations for lists. Additionally, its powerful type system can express constructs (called type classes) that enable the definition of abstract operations usable with many different types.
This project, therefore, investigated how to port various functionalities of the Haskell standard library to JavaScript. Furthermore, the focus was to figure out how to model type classes analogous to Haskell using "JSDoc", the optional type system of JavaScript. The solutions and findings of this project will become part of the existing zero-dependency Web UI Toolkit "Kolibri".
Since Haskell is a functional programming language, it uses many concepts that are unfamiliar with JavaScript. However, JavaScript supports some of them, which have been used where possible. Using them allows the writing of robust, well-testable, and reusable code. Unit and user tests ensured the correctness and usability of the artefacts.
JavaScript defines iteration protocols to process data structures element by element. This project exploited these protocols to define a new data structure called "sequence" and operations to transform and process it. These artefacts form the "Sequence library". Operations of the Sequence library are compatible with any data structure that complies with the iteration protocols. All functions evaluate their input lazily and do not mutate it.
Using JSDoc, the type class monad could be defined. However, limitations lie in the missing name overloading in JavaScript, the reduced type safety, and the lack of higher-kinded types in JSDoc.
Based on these findings, JINQ emerged, capable of uniformly querying any data structures implementing the type class monad.
In his paper "Why Functional Programming Matters", John Hughes shows how functional programming enables developers to write reusable and modular code. He explains this with several example programs. The results of this project allow the implementation of the programs showcased in Hughes' paper. Additionally, JINQ demonstrates the power of monads in JavaScript. Thus, this project reveals that JavaScript can effectively implement and utilize many concepts known from functional programming.

Introduction

Problem Statement

JavaScript already includes some concepts known from functional programming. For example, it supports higher-order functions, arrow functions or auto-currying of function parameters.
However, it also lacks many concepts included in languages such as Haskell. Based on the previously mentioned language characteristics, it is possible to recreate some of these missing features. The Haskell standard library, for example, has a wide range of operations to transform or evaluate lists [1]. JavaScript, on the other hand, only implements a few of these operations for its arrays. In addition, Haskell only executes these operations if the list or parts are consumed (lazy evaluation). Thus, it invests no unnecessary computing time.
The JavaScript iteration protocols [2] allow implementing this laziness. This project work [3], therefore, explores building a functional standard library based on these iteration protocols, providing operations analogous to the Haskell standard library.

However, another goal is to examine how it is possible to implement Haskell type classes in JavaScript. For example, the type class "monad" makes it possible to write more abstract functions that can handle a wide variety of data structures.

The resulting artefacts integrate seamlessly into the Web UI toolkit "Kolibri" [4]. On the one hand, this integration means that the artefacts belong to the toolkit. On the other hand, they maintain the high standards of the Kolibri. These standards include high code quality, expressed by automated tests, fully typed JavaScript using JSDoc, and keeping the Kolibri's zero dependency approach. For this, things like specific data structures or the integrated testing framework, which Kolibri already brings, can be used.

Results

Through the knowledge gained, a functional standard library has emerged, which mainly consists of two parts:

1. **Sequence library:** The Sequence library introduces a new data structure called "sequence". A sequence implements the mentioned JavaScript iteration protocols and can therefore be lazily evaluated. In addition, the Sequence library provides various operations on a sequence to transform and evaluate it. These operations work on sequences and any data structure that implements the iteration protocols. Thus, among other things, JavaScript arrays can also be lazily evaluated and transformed using the Sequence library.
   Operations performed on sequences are static functions that do not belong to an object or a

class. This means they are not accessible using the dot on a sequence but are imported using ES6 modules [5], which brings several advantages:

   a) The Sequence library is easily extensible. Without changing existing code, anyone can add new operations.
   b) All operations work with any iterable.
   c) The implementations can be divided into different modules.

   Passing an iterable to such an operation does not change the underlying iterable but only decorates it. The operation returns a new sequence with the same API, which calls when consumed, the underlying iterable.

2. **Monadic API in JavaScript:** The knowledge gained from the Sequence library enabled adapting the type class monad from Haskell into JavaScript. This type class is defined as JSDoc, which allows writing generic functions that can handle data structures of various kinds. A data type must implement each operation defined by the JSDoc type `MonadType` to be compatible with such a function.
   The sequence already implements this type completely. Since the Sequence library allows converting any iterable into a sequence, these operations are available for all iterable objects. To show the versatility of the `MonadType`, an implementation for the already existing `MaybeType` was also added. The definition of the monad in JavaScript allows the implementation of language-integrated queries (LINQ) [6]. LINQ allows querying different data structures implementing the monadic API using the exact same notation.
   The power of this concept becomes even more visible in practical examples: A new type `JsonMonad`, which also implements the monadic API, allows the processing of inconsistent JSON files using language-integrated queries and thus prevents frequently occurring mistakes. This makes it easier to process data that originates from an API.

Automated unit tests ensure the correctness of the two previously explained artefacts. Since many operations of the Sequence library must provide the same properties, testing in a structured way is necessary. A so-called testing table represents this structure. This testing table allows writing general tests which assert specific properties applying to every operation. Newly discovered bugs that can potentially occur in all of these operations can thus be caught in a single place by one single test.
In addition, some operations must fulfil unique properties or invariants. When an operation is performed on a sequence, these invariants must always hold, regardless of the underlying sequence. The testing table also supports the definition of such invariants as predicates. This allows finding bugs occurring in edge cases, such as the empty sequence.

Examples

The lazy evaluation of the Sequence library offers the possibility to implement the alpha-beta heuristic [7, Ch. 5]. This algorithm allows the implementation of a computer-controlled opponent for turn-based games. We have implemented such an opponent for the game tic tac toe to demonstrate the Sequence library. This opponent automatically selects the best moves for its turn. Figure 1 shows a screenshot of a web application that uses this algorithm:

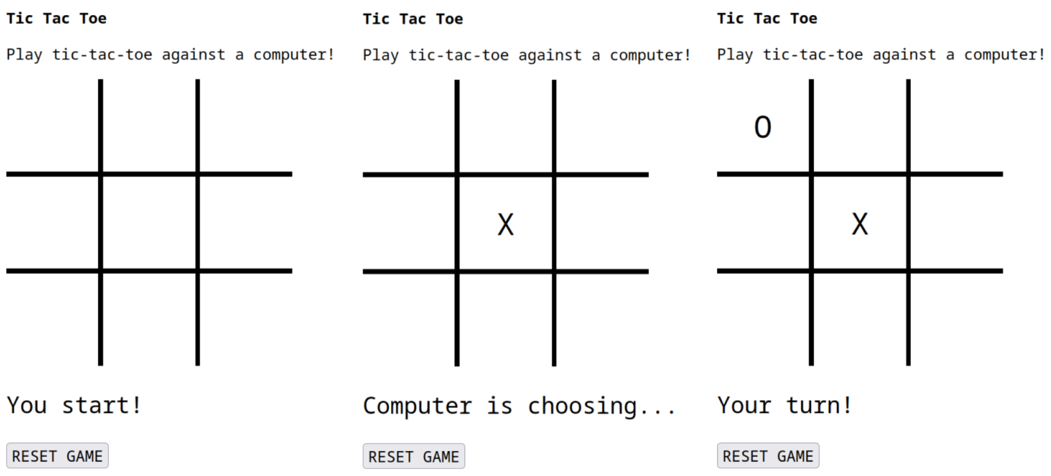


Figure 1.: A website using the algorithm

Methodical Elaboration

To achieve essential goals we followed certain concepts and methods:

1. **Iterative approach**: This bachelor thesis is not a pure implementation task. It was unclear how much could ultimately be achieved. Therefore, we chose an iterative approach to solve the problems. Regular meetings with the project supervisor ensured the progress of the project. At the end of each meeting, we discussed the next iteration, and the direction was slightly adjusted based on the prior results.
2. **Test-driven development:** Operations for the sequence are very well testable since they are independent of state. Therefore, before implementing a new operation, we wrote a test for the main branch of functionality to ensure correct behaviour. Before we implemented the next sub-branch, we again wrote a test. Thus, as the functionality grows, so does the test coverage. This not only ensures code quality but also makes refactorings safer.
3. **User tests:** User tests conducted with several developers ensure the usefulness of the functional standard library. Test users solve various tasks with the help of the functional standard library and complete a questionnaire about them. The questionnaire aims to determine whether the users find the library easy to use and to receive general suggestions for improvement — findings from this flow back into the implementation. Questions with text answers allow for capturing the general mood of the participants.

Reader Guidance

This thesis consists of two parts. The first part (chapters 2-6) deals with conceptual aspects. Each chapter focuses on a specific part of the thesis and explains all concepts and ideas behind it. These chapters, therefore, give a deep understanding of the work.
The second part (chapters 7-8) summarizes the thesis. Chapter 7: "API" gives an overview of the whole API without going into depth. Chapter 8: "Results" evaluates the work and concludes. It also contains different examples showing the power of the implemented artefacts.

Acknowledgment

We want to thank Prof. Dierk König for his invaluable guidance and unwavering support throughout our bachelor thesis, "Functional Standard Library for the Kolibri Web UI Toolkit." His expertise and encouragement have been crucial in shaping our research and contributing to its success.

We also extend our sincere appreciation to all the people who supported us during our academic journey, especially during the completion of this thesis. Our fellow students deserve a special mention for engaging in discussions, which sparked new ideas and inspired us.

Furthermore, we would like to express our deepest gratitude to our friends and family for their constant support and understanding throughout our studies and the completion of this thesis. Their encouragement and belief in us have kept us motivated and focused.

Lastly, we would like to acknowledge the contributions of all individuals who directly or indirectly assisted us in this endeavour. Your help, encouragement, and valuable input have played a significant role in successfully completing our bachelor thesis.

We are truly grateful for all the support and guidance we have received.

Contents

List of Figures

List of Tables

List of Listings

1. Application Domain

This chapter introduces the project "A Functional Standard Library for the Kolibri Web UI Toolkit". It begins by briefly describing the Kolibri toolkit and its benefits. It then discusses the motivation for developing a standard library for the toolkit and how it fits into the overall architecture. The chapter then lists the most relevant sources that were used in the research for this thesis and briefly describes each source, highlighting the most helpful parts. Finally, it discusses similar projects and how the functional standard library differs.

1.1. Description of the Application Domain

1.1.1. Functional Programming in JavaScript

JavaScript is omnipresent on the web client side. In many aspects, JavaScript differs from other programming languages, and sometimes it has the reputation of being a bit strange. Nevertheless, the language offers some excellent concepts which enable functional programming. Therefore, the language provides the following features:

- Arrow functions: Assignment of functions to variables
- Higher-order functions: Functions that can take other functions as arguments or return functions as results
- First class functions: Functions are objects that can be treated as any other object. A function can also have properties. [8]
- Lazy evaluation: calculations are done when the result is needed
- Curried functions: passing one argument to a function taking two parameters returns a new function just taking one parameter

Listing 1.1 shows these concepts with some examples. First, on line 1, the arrow function `plus` defines a function that expects two arguments in curried style. Then, the function is partially evaluated by passing one argument to it. The result is a new function `plusOne`. This process is called partial application. `plusOne` is now a function that expects one more argument to evaluate the result. Passing the next argument calculates the addition and stores the result in the variable `result` on line 3.

```
1 const plus    = a => b => a + b; // an arrow function taking two params
2 const plusOne = plus(1);         // plus(1) returns a new function
3 const result  = plusOne(5);      // applying another param calculates the result
4
5 const commentResult = func => {  // a function as argument
6
7   // defining a new function which takes one argument
```

```
 8    const returnFunction = value => {
 9      const comment = "The result is: ";
10      const result  = func(value);
11      return comment + result;
12    };
13
14    // returning a function from a function
15    return returnFunction;
16  };
17
18  const commentResultFn = commentResult(plusOne); // yields a new function
19  const result          = commentResultFn(6);     // evaluating the result
20  console.log(result);
21  // => Logs 'The result is: 6'
```

Listing 1.1: Functional concepts in JavaScript

Next, let us consider the `commentResult` function on line 5 to 16, which takes another function as input and generates a new function. To achieve this, the original function, referred to as `func` is enclosed within a wrapper function to produce a commented result. The outcome of invoking `commentResult` with the `plusOne` function is stored in `commentResultFn`, which itself becomes a function. The actual computation and output of results occur when `commentResultFn` is called on line 19. This example showcases how functions can be treated as regular objects.

1.1.2. The Kolibri Web UI Toolkit

The web page of the Kolibri describes the toolkit as, "Kolibri wants to be a sustainable, high-quality toolkit" [4], - and it is. The Kolibri Web UI Toolkit is a collection of JavaScript abstractions. Experts test all components, which must satisfy high functional and non-functional standards. An essential part of the toolkit is the dependency-less implementation. Thus, it represents a certain antithesis to the frameworks dominated by `npm` libraries. This is important to fulfil the quality requirements in the long term.
The toolkit is under continuous development, with each student project focusing on a new topic. The main goal of this thesis is to extend the toolkit with a standard library based on functional concepts.

1.1.3. The Basis of the Project

This work continues the previous project from the fifth semester of Computer Science studies. Therefore, we have already dealt with the topic before this thesis, but on a smaller scale. This predecessor project's artefacts include a Logging Framework for the Kolibri toolkit and the first version of the functional standard library. These results made it possible to develop the library to its current state. By researching the literature and trying different approaches, a versatile and robust implementation of a functional standard library emerged.

1.1.4. Who is the Toolkit for?

The toolkit is intended for people who want to develop robust web applications without the burdensome dependencies and vulnerabilities associated with multiple packages. As an open-source project, it is available for anyone to utilize. The fundamental concept entails copying the toolkit into your project, allowing you to utilize the necessary components. Naturally, users have the freedom to modify elements to suit their specific needs or contribute new implementations as desired.

1.2. Literary and Studies

This section is about formative sources for this thesis. The following briefly introduces the sources and explains their meaning for this work.

1.2.1. Why Functional Programming Matters

Why Functional Programming Matters [7] by John Hughes gives an impressive insight into functional programming, mainly higher-order functions, and laziness is discussed intensively. This paper strongly influenced the development of the standard library, as it describes many concepts and best practices. In addition, the user test is based on an example of this paper.
Furthermore, this paper has improved our understanding of function composition, partial application, and laziness. Therefore, it has enhanced the quality and robustness of the standard library.
The Sequence library, part of the functional standard library, implements ideas and functionalities described in this paper. It is a prominent part of this thesis, allowing to write good, modularized, and reusable code. Chapter 3: "Modularizing Programs" deals with these ideas in depth.

1.2.2. Frege Goodness

Frege Goodness [9] by Prof. Dierk König is a book about Frege, a functional programming language for the JVM. This book contains examples and explanations of various applications. We gained new insights and inspiration for implementations and examples based on these. We treated individual chapters more intensively, such as *The Power of the Dot*. The insights gained helped to make far-reaching decisions for the Sequence library. Chapter 5: "The Power of the Dot" describes more details about that.

1.2.3. Quickcheck

QuickCheck: A Lightweight Tool for random testing of Haskell Programs [10] by Koen Claessen and John Hughes describes how property-based testing works in Haskell. The adaption of specific approaches described by this paper improved the code quality of the Sequence library. Since this project did not focus on this topic, we only took over a few ideas. However, further implementations in this direction are possible. Chapter 6.4: "Testing based on Properties" discusses the adapted concepts in more detail.

1.2.4. MDN - Iteration Protocols

The Mozilla Firefox webpage describes the iteration protocols [2] and is the basis for developing the Sequence library. The page contains all relevant facts about iterables and iterators. Section 2.1: "Iterators in General" explains the implementation of these protocols.

1.2.5. LINQ

LINQ (Language integrated queries) [6] is a widely used concept for querying data. It enables to process different data sources in a fluent and declarative way. This declarative nature makes it straightforward to write queries and understand them later. JINQ, which emerged in this project, is based on the same ideas. LINQ heavily inspired the naming and the scope of JINQ, enabling users familiar with LINQ to use JINQ immediately. Section 4.4: "Query different Data Structures" discusses JINQ in detail.

1.3. Comparable Works

This section explains projects that influenced this thesis. All of them are similar to the implemented artefacts but differ fundamentally since the standard library was built explicitly for the Kolibri, also supporting its `Pair` and `Stack` types. Furthermore, it does not introduce new dependencies to other projects or dependency management systems.

1.3.1. Lodash

Lodash [11] is a modern JavaScript utility library that brings many solutions similar to the functional standard library. Nevertheless, there are some differences:

- Functions in Lodash do not take their parameters curried.
- Functions in Lodash expect the receiver as first parameter.

Section 3.2: "Placing the Receiver at the End" describes how these two ideas (and some more) are essential when writing reusable code. However, Lodash also provides a functional programming first module that supports the above.[1] Nevertheless, the documentation for this is much less extensive than that of the regular Lodash module. Both library versions provide much less precise typing than this standard library.
The Sequence library additionally supports monadic operations described in section 4.3: "Monads in JavaScript". Thus, the Sequence library allows using generic functionalities based on the monadic API, such as JINQ.

[1] https://github.com/lodash/lodash/wiki/FP-Guide

1.3.2. rxjs

rxjs [12] offers very similar functionality as the Sequence library. However, these functions work with a reactive data structure called `Observable`. Nevertheless, rxjs served several times as a good role model for design decisions. The idea to combine multiple operators using `pipe` comes from rxjs.

1.3.3. Lambda Calculus for Practical JavaScript

"Lambda Calculus for Practical JavaScript" [13] provides an introduction and some applications of lambda calculus in JavaScript. The immutable data structure `Stack` comes from this project work. The constructor `StackSequence` provided by the Sequence library offers the possibility to make a `Stack` iterable.

2. Iterators in JavaScript

This chapter explains the concept of iterable and iterator objects. The sequence is an iterable data structure developed during this thesis. Alongside many operations to process iterable objects, it forms the "Sequence library". Many discussions cover challenges and solutions for creating a robust and user-friendly API.

2.1. Iterators in General

In Computer Science, iterators are a popular concept. An iterator provides access to elements of a data structure. It does not matter if the iterator accesses a data structure fully kept in memory (like arrays) or if it computes the elements lazily when queried. In both ways, a function call on the iterator retrieves the next value. Iterators are an essential part of this work. Therefore, it is crucial to understand the fundamentals of this concept.

2.1.1. Iterable Data Structures in JavaScript

JavaScript knows two protocols that define how an object can be used for iteration - the iterable protocol and the iterator protocol [2]. An iterable object, also known as an iterable, represents a collection of values. A function call on its iterator, gives access to the next element in the collection - there is no way to access a previous value. Once an iterator reaches its end, it is considered "used up" - subsequent calls to such an exhausted iterator won't produce any meaningful results.

The Iterable Protocol

In JavaScript, an object is considered as iterable if it contains a property called `[Symbol.iterator]`. This property signifies that the object is of type `Iterable<T>`. Invoking the `[Symbol.iterator]` property creates an iterator that follows the iterator protocol explained in the next section. Objects in JavaScript that implement this property can be utilized with destructuring [2]and `for...of` loops. Listing 2.1 showcases an example of an object defining the `[Symbol.iterator]` property.

```
return {
  [Symbol.iterator]: () => {
    return { next: next }; // next is defined in 2.2
  }
}
```

Listing 2.1: Iterable protocol

[2]The destructuring assignment syntax is a JavaScript expression that allows to extract items of iterables into individual variables.

The Iterator Protocol

An invocation of the `[Symbol.iterator]` property of an iterable returns an object complying with the iterator protocol. It must implement a function `next`, which defines how and which values are returned when iterating. Each iteration on an iterator calls this function. `next` returns an object, which must include two properties: `value` and `done`. The property `value` contains the current value of the iteration, while `done` represents the information on whether the end of the iterator has been reached. The code of listing 2.2 shows a simple implementation of it. Each call on `next` returns an object with the `value 1`. `done` is always `false` - hence the iterator never ends.

```
const next = () => {
  return { done: false, value: 1 };
};
```

Listing 2.2: Iterator protocol

Creating Iterables

By combining these two protocols, the result could look like listing 2.3. The constructor `InfiniteOnesIterable` on line 1 wraps the two previously defined implementations. Therefore, this constructor constructs iterable objects. Since the return value is a number, this is an iterable of type `Iterable<Number>`.

```
 1 const InfiniteOnesIterable = () => {
 2   const next = () => ({ done: false, value: 1 });
 3
 4   return {
 5     [Symbol.iterator]: () => ({ next })
 6   }
 7 };
 8
 9 const [one, anotherOne, andOneAgain] = InfiniteOnesIterable();
10
11 for (const _one of InfiniteOnesIterable()) {
12   // hangs forever, since done is always false
13 }
```

Listing 2.3: Iterable and iterator protocol

Because objects created by `InfiniteOnesIterable` adhere to the JS iteration protocols, it is possible to use language features like destructuring and `for..of` to process these objects as shown on line 9 to 13. However, beware of this case. This would lead to an infinite loop because the property `done` never becomes `true`. As a result, the iteration never ends. This work shows finite iterables later. For now, the focus is on the protocols. Therefore, this example is sufficient at this point.
Such protocols make it possible to build customized iterables and collections. This opens new possibilities. Various programming tasks can have different, more straightforward solving approaches in a more declarative way to write. There are already some JavaScript iterables present. Arrays and `HTMLCollections` are probably the most prominent of these.

2.1.2. Types of Iterables

JavaScript distinguishes between iterables and iterators. These protocols also have their corresponding types. Listing 2.4 shows an excerpt of them:

```
 1 // lib.es2015.iterable.d.ts
 2
 3 interface Iterable<T> {
 4     [Symbol.iterator](): Iterator<T>;
 5 }
 6
 7 interface Iterator<T, TReturn = any, TNext = undefined> {
 8   next(...args: [] | [TNext]): IteratorResult<T, TReturn>;
 9     return?(value?: TReturn): IteratorResult<T, TReturn>;
10     throw?(e?: any): IteratorResult<T, TReturn>;
11 }
12
13 type IteratorResult<T, TReturn = any> = IteratorYieldResult<T>
14                                       | IteratorReturnResult<TReturn>;
15
16
17 interface IteratorYieldResult<TYield> {
18     done?: false;
19     value: TYield;
20 }
21
22 interface IteratorReturnResult<TReturn> {
23     done: true;
24     value: TReturn;
25 }
```

Listing 2.4: Types of iterables

- Line 3 to 11: An iterable is of type `Iterable<T>`, whereas the object returned by the property `[Symbol.Iterator]` is of type `Iterator<...>`. An iterator requires having a property `next`. This is the function that returns values when iterating. These values must be of type `IteratorResult<...>`.
- Line 13 to 25: `IteratorResult<...>` itself is defined to return an object of type `IteratorReturnResult<...>`, which either is of type `IteratorYieldResult` or `IteratorReturnResult`. This object contains the actual values we want to work with.

Section 2.3.3: "Stateful Decorating" explains the reason for this nested architecture of the JS iteration protocols.

Closer Look to IteratorReturnResult

When iterating an iterable, the returned elements are of type `IteratorYieldResult<T>`. Calling the function `next` on an exhausted iterator returns an element of type

IteratorReturnResult<TReturn>. This ensures that done is set to true, as can be seen on line 23 in listing 2.4.

2.1.3. Illustration of the JS Iteration Protocol

Listing 2.5 illustrates the behaviour of the JS iteration protocols more clearly using an array:

```
1 // array including two values, 0 and 1
2 const list = [0, 1];
3 const iterator = list[Symbol.iterator]();
4
5 iterator.next(); // returns { done: false, value: 0 }
6 iterator.next(); // returns { done: false, value: 1 }
7 iterator.next(); // returns { done: true,  value: undefined }
8 iterator.next(); // returns { done: true,  value: undefined }
```

Listing 2.5: Example: JS Iteration Protocol

On line 3, invoking [Symbol.iterator] returns the iterator of the iterable list. After that, next gets called four times directly on the iterator. Since the iterable contains only two elements, the third and fourth call on next returns an object of type IteratorReturnResult<TReturn>. Thereby, done is true and value is undefined.
Using destructuring and for...of, an iterable would stop iterating after the second call. This means a for...of loop runs until the property done is set to true.

2.2. Sequence: An iterable Series of Values

In Computer Science, naming elements accurately poses a significant challenge when aiming to develop sustainable and robust code. We decided to name series of data a "sequence". It has been influenced by several factors:

- Sequences are not conventional lists known from other programming languages.
- The name sequence is already known from mathematics.
- Giving this data structure a more familiar name, for example "list", leads to wrong assumptions.

What distinguishes the object generated by the sequence from the conventional list is that a sequence generates its values when they are requested. Therefore, it needs almost no memory. At the same time, you have the impression that you are dealing with a vast amount of data. So the constructor Sequence emerged, which generates such a series of data.

2.2.1. Components of a Sequence

Defining a sequence requires specifying three essential points:

1. A fixed starting value for the sequence
2. A function that determines whether the sequence should generate further values
3. A function to calculate the next value based on its predecessor

Listing 2.6 on line 2 shows the passing of these three elements as arguments to the constructor. To keep the focus on the core elements of the sequence, listing 2.6 omits some functionality - a following section will discuss it. The `next` function, explained by section 2.1.1: "The Iterator Protocol", is on lines 11 to 15. It contains the logic to return the next value in an iteration. First, it uses `whileFunction` to check if the sequence has finished. If this is not the case, the `incrementFunction` calculates the next element, which afterward will be returned.

```
 1 // Sequence.js
 2 const Sequence = (start, whileFunction, incrementFunction) => {
 3 
 4   const iterator = () => {
 5     let value = start;
 6     /**
 7      * @template _T_
 8      * Returns the next iteration of this iterable object.
 9      * @returns { IteratorResult<_T_, _T_> }
10      */
11     const next = () => {
12       const current = value;
13       const done = !whileFunction(current);
14       if (!done) value = incrementFunction(value);
15       return { done, value: current };
16     };
17 
18     return { next };
19   };
20 
21   return  /* omitted */;
22 };
```

Listing 2.6: Parts of Sequence

2.2.2. Using a Sequence

Listing 2.7 shows the definition of a sequence of even numbers smaller than ten and how to use it.

```
const startValue        = 0;
const whileFunction     = x => x < 10;
const incrementFunction = x => x + 2;

const seq = Sequence(startValue, whileFunction, incrementFunction);

for (const elem of seq) {
  console.log(elem);
}
```

```
10 // => Logs '0, 2, 4, 6, 8'
```

Listing 2.7: Sequence of even numbers

The `for..of` loop iterates over the sequence until `done` becomes `true`. Meanwhile, `console.log` writes the elements to the console. Line 10 shows the output produced.

2.2.3. Generating Numbers with a Range

When considering sequences, a common requirement is to generate a series of numbers, often with a specified starting point, an end value, and adjustable increments. This is a classic definition of implementing a range. Ranges are built into many programming languages such as Haskell [1], Python [14] or Kotlin [15].

Listing 2.8 shows the construction of a range in Haskell:

```
ghci> r = [1..10] -- contains the numbers from 1 up to 10.
ghci> r
[1,2,3,4,5,6,7,8,9,10]
ghci> r = [1..] -- contains an infinite range
```

Listing 2.8: Range in Haskell

Listing 2.9 demonstrates different ways to initialize the range of the Sequence library:

```
const range               = Range(3);
const [five, three, one]  = Range(1, 5, -2);
const [three, four, five] = Range(5, 3);
```

Listing 2.9: Kolibri toolkit Range

What can a Range be used for?

Ranges offer efficient processing of sequences of almost any size, consuming minimal memory resources. As a result, they serve as excellent foundational components for larger structures. They can be utilized for lazily generating numbers or executing a specific action several times.

The Boundaries

Built-in ranges are commonly available in various programming languages - the concept of ranges is therefore well-known. However, it is worth noting that not all languages handle range boundaries similarly. For instance, Kotlin and Haskell consider both boundaries inclusive. Thus, a range from 1 to 3 in these languages would include the elements 1, 2, and 3. On the other hand, in Python, only the lower limit is inclusive. Therefore, the boundaries must be set as 1 and 4 to create an equivalent range in Python. Since it seems more intuitive to have both borders inclusive, the range for Kolibri toolkit works accordingly.

Implementation

A range uses a sequence under the hood. Helper functions analyze the passed parameters and, if necessary, reorder them to return the desired sequence. Listing 2.10 shows the main part of the implementation of the range:

```
/**
 * @constructor
 * @pure
 * @param { !Number } firstBoundary  - the first boundary of the range
 * @param { Number }  secondBoundary - optional second boundary of the Range
 * @param { Number }  step - size of a step, processed during each iteration
 * @returns SequenceType<Number>
 */
const Range = (fstBoundary, sndBoundary = 0, step = 1) => {
  const stepIsNegative = 0 > step;
  const [left, right] = normalize(fstBoundary, sndBoundary, stepIsNegative);

  return Sequence(
    left,                                              // start value
    value => !hasReachedEnd(stepIsNegative, value, right), // while function
    value => value + step                              // increment function
  );
};
```

Listing 2.10: Implementation of the Range

While closely examining the features of a range, we will also cover the functions `normalize` and `hasReachedEnd` to explain them in more detail later in this chapter.

Features

- **Parametrization:** The range takes 1 to 3 parameters. The parameters specify the lower limit, the upper limit, and the step size. If not all parameters are set, the range uses suitable default values.
- **Interchangeable boundaries:** The order of the lower and the upper limit is interchangeable when creating a new range.
- **Negative boundaries and step:** The boundaries and the step size of the range can be positive or negative integers.

Normalization of the Boundaries

To ensure flexibility, the range provides the interchangeability of boundaries and the option for positive or negative step values. Initially, the range boundaries are normalized based on the step value. This normalization determines which boundary will contain the first generated number and which will represent the last number. The following figure 2.1 explains the process in detail:

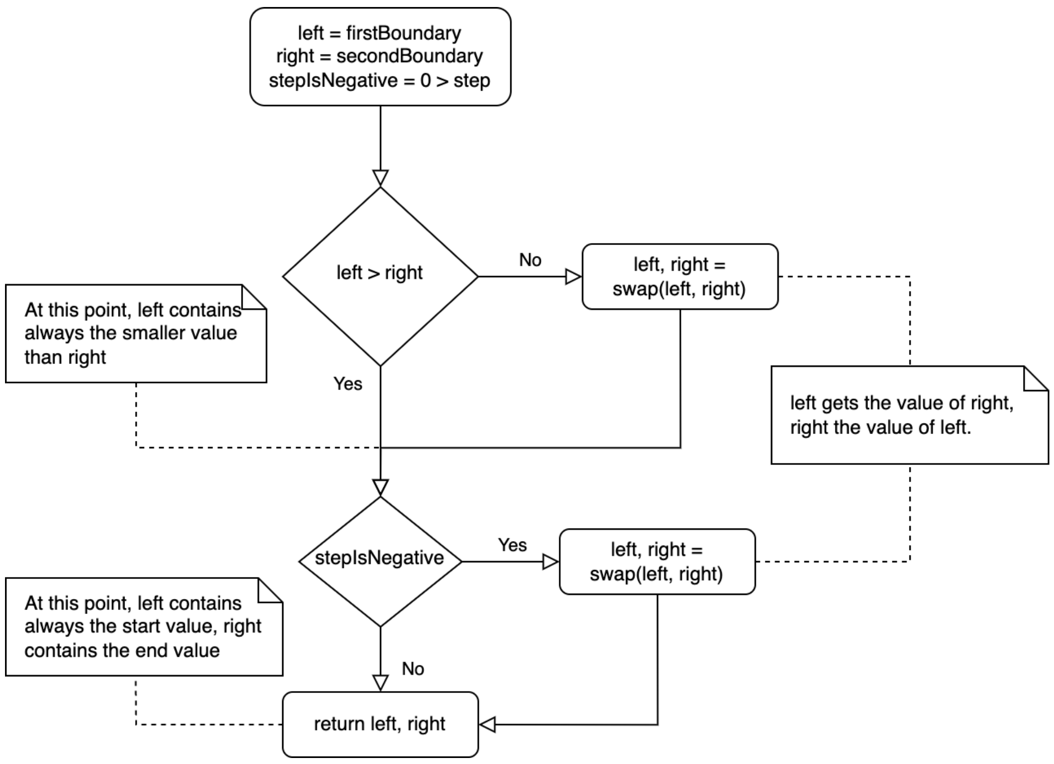

left = firstBoundary
right = secondBoundary
stepIsNegative = 0 > step
left > right
No
left, right = swap(left, right)
Yes
At this point, left contains always the smaller value than right
left gets the value of right, right the value of left.
stepIsNegative
Yes
left, right = swap(left, right)
No
At this point, left contains always the start value, right contains the end value
return left, right

Figure 2.1.: Normalization Flow-Chart diagram

The End of a Range

The `hasReachedEnd` function detects the end of the range and defines the `untilFunction` of the underlying sequence. Listing 2.11 shows the implementation of the function:

```
/**
 * Determines if the end of the range is reached.
 * @param   { Boolean } stepIsNegative
 * @param   { Number }  next - the current element of the range
 * @param   { Number }  end  - the last value of the range
 * @returns { boolean }
 */
const hasReachedEnd = (stepIsNegative, next, end) =>
    stepIsNegative ? next < end : next > end;
```

Listing 2.11: hasReachedEnd implementation

The function must be able to distinguish between two different cases:

- If the step size is a negative, the range counts from top to bottom: The range reached its end as soon as next is smaller than right.
- If the step size is positive, the range counts from bottom to top: The range reached its end as soon as next is larger than right.

Some Restrictions of Using a Range

When using the range, one has to pay attention to the contract:

- End value may not be reached exactly, but will never be exceeded.
- Zero step size leads to infinite loops, returning always the same values.
- Only values that behave correctly with respect to addition and size comparison may be passed as arguments.

Using a Range

Following listing 2.12 demonstrates various ways to construct a range.

```javascript
// typical cases
for (const value of Range(3)) { console.log(value); }
// => Logs '0, 1, 2, 3'

for (const value of Range(2,3)) { console.log(value); }
// => Logs '2, 3'

// lower and upper boundaries interchanged
for (const value of Range(3,2,1)) { console.log(value); }
// => Logs '2, 3'

// negative step size
for (const value of Range(4,6,-2)) { console.log(value); }
// => Logs '6, 4'

for (const value of Range(6,4,-2)) { console.log(value); }
// => Logs '6, 4'

// range with negative boundary
for (const value of Range(0,-2,-1)) { console.log(value); }
// => Logs '0, -1, -1'
```

Listing 2.12: Range examples

2.3. Decorating Sequences

This section explores how to modify sequences by implementing the decorator pattern and effectively managing sequence state.

2.3.1. Decorator Pattern

Let us look at the decorator pattern [16, p. 226] to understand the content of the following sections. In object-oriented programming, the decorator pattern is a widely used concept: An object decorates another, as the name implies. As a result, an outer object holds an inner object, while both implement the same interface. The outer object forwards requests to the inner one. This enables the outer to modify (decorate) the behaviour of the inner. Figure 2.2 shows how a decorator forwards the receiving calls and transfers the answer back to the client.

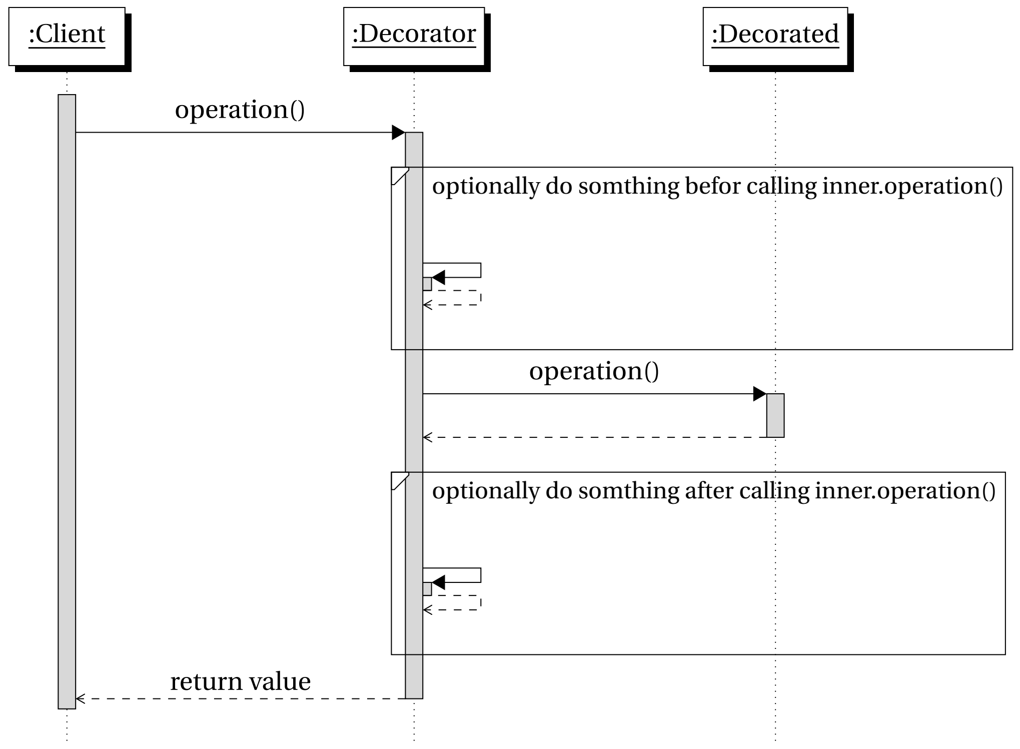

:Client
:Decorator
:Decorated
operation()
optionally do somthing befor calling inner.operation()
operation()
optionally do somthing after calling inner.operation()
return value

Figure 2.2.: Decorator sequence diagram

The decorator pattern is often used in object-oriented programming languages because of its ability to add functionality to an object at runtime. It is also possible to add extended functionality to a single class object. This pattern gives a great way to modify the behaviour of iterables. The Sequence library uses this approach to create operations to transform sequences.

2.3.2. Processing Sequences

Processing Iterables with Functions

In the following, `map` serves as a representative for any function of the Sequence library. The Sequence library calls such functions operators, according to the module they are in. Listing 2.13 shows how `map` processes a sequence. The implementation uses the decorator approach just mentioned. `map`

decorates a sequence on line 21.
The function signature of `map` on line 1 shows that a client can invoke `map` with two arguments:

1. A mapping-function to transform one single element.
2. An object of type `Iterable<T>`

An iterable must adhere to the JS iteration protocols outlined in section 2.1.1: "The Iterable Protocol". Therefore, `map` can process sequences, arrays, or any other iterable.

```
 1 const map = mapper => iterable => {
 2
 3 const mapIterator = () => {
 4    const inner = iteratorOf(iterable);
 5    let mappedValue;
 6
 7    const next = () => {
 8      const { done, value } = inner.next();
 9      if (!done) mappedValue = mapper(value);
10
11      return { done, value: mappedValue }
12    };
13
14    return { next };
15   };
16
17   // explained in 2.3.4:  "Adding Monadic Functions to Iterables"
18   return createMonadicSequence(mapIterator); };
19
20 const sequence = Sequence(0, x => x < 10, x => x + 1);
21 const mapped   = map(x => x * 2)(sequence);
```
Listing 2.13: Implementation of map

Because `map` decorates iterables, it also returns an object adhering to JS iteration protocols. We define this object on line 21 as `mapped`. Consequently, `map` also has a function called `next`. Since the JS iteration protocols specify only this single function, it is the only one that must be exposed. The newly created object `mapped` forwards function calls on `next` to the inner function `next`, on line 8. The `mapper` function then processes the result and returns it. So `map` *decorates* the function `next` of the inner iterable.

2.3.3. Benefits of the Decorator Approach

Stand-alone Functions

In an object-oriented approach, the sequence object would provide a function `map`. Therefore, such operators are available by using the dot notation, similar to Java implementing the Stream API [17]. However, with the approach of providing independent functions, there are three significant advantages:

1. Adherence to the open-close principle [18, p. 3]. Changes to an operator do not affect the implementation of constructors which create sequences. Also, extensions to the Sequence library do not affect existing code.
2. Adherence to the single responsibility approach. The constructor `Sequence` has only the task of creating a sequence. Mapping the elements of a sequence is not its responsibility.
3. Easy scalability is guaranteed. It is very straightforward to add new functionality from the outside.

Additionally, the `operators` implement their arguments in a curried style and accept the receiving iterable at the last position. This allows using eta-reduction and other advantages, as section 3.2: "Placing the Receiver at the End" outlines.

Stateful Decorating

A state is present as soon as operators decorate iterables or implement additional functionalities. In the case of `map` this state is the function `mapper` and the iterator of the inner iterable.

There are two valid possibilities for including the state:

1. Placing the state into the closure scope of the iterable.
2. Implementing the state into the closure scope of the iterator.

In both variants, ensuring the underlying object remains unchanged is crucial.

Scenario 1: Placing the State inside the Iterable

Listing 2.14 shows an example of a sequence generating numbers from zero to five. On line 3, value `i` works as a counter and represents the state. A call to `SampleIterable` creates a state which is valid for the entire iterable's lifetime.

```
1 const SampleIterable = () => {
2
3   let i = 0;
4   const next = () => {
5     return { done: i > 5, value: i++ };
6   };
7
8   return { [Symbol.iterator]: () => ({ next }) }
9 };
```

Listing 2.14: Scenario 1 - State in closure scope of iterable

Listing 2.15 shows a possible program flow. The program first creates an object of `SampleIterable`, maps it and processes an element on the original object. So both for loops work on the same underlying iterable. Line 13 shows an expected result.

```
const seq = SampleIterable(); // [0, 1, 2, 3, 4]
const mapped = map(id)(seq);

for (const elem of mapped) {
  console.log(elem);
  break; // Just consuming one element
}
// => Logs: '0'

for (const elem of seq) {
  console.log(elem);
}
// => Should log: '0, 1, 2, 3, 4'
```

Listing 2.15: Scenario 1 - Example usage

However, since both loops operate on the same iterable (`seq`), the second output will not give the expected result but "1, 2, 3, 4".
To achieve the expected result, `map` must copy the object before processing. That implies that each iterable must be copyable. An example implementation of a copyable iterable shows listing 2.16.

```
1  const SampleIterable = () => {
2
3    let i = 0;
4    const next = () => {
5      return { done: i > 5, value: i++ };
6    };
7
8    const copy = () => SampleIterable();
9
10   return {
11     [Symbol.iterator]: () => ({ next }),
12     copy: copy
13   };
14 };
```

Listing 2.16: Iterable with copy

Since `copy` must be callable from outside, returning it alongside `[Symbol.iterator]` on line 12 is required. `map` is now able to copy the underlying iterable before consuming its elements. However, `map` itself must also implement `copy`, because its interface must be the same as the decorated object (and also to ensure that iterables created using `map` can be decorated as well). The following listing 2.17 shows an implementation of `map` supporting `copy`:

```
const map = mapper => iterator => {
 const inner = iterator.copy();
 let mappedValue;

 const next = () => {
```

```
 6     const { done, value } = inner[Symbol.iterator]().next();
 7     if (!done) mappedValue = mapper(value);
 8     return { done, value: mappedValue }
 9   };
10
11   return {
12     [Symbol.iterator]: () => ({ next }),
13     copy: () => map(mapper)(inner)
14   };
15 };
```

Listing 2.17: Implementation of map with copy

On line 2, `map` references a copy of the original iterable. Further processing on line 6 works on this copy to protect the original from being mutated.

In many respects, `copy` is an elaborate thing. All operators and constructors dealing with iterables must implement it. Thus, implementation errors are almost certain. On the other hand, it is an extra effort from a performance point of view. It means more function calls and also larger call stacks to manage. The advantage of this implementation is that partially processed iterators can be further utilized while maintaining their current state without reinitializing the state with a new iteration. Operators can only process objects that implement `copy`. Other iterable objects are not processable anymore, which is a significant drawback.

Scenario 2: Placing the State inside the Iterator

Listing 2.18 represents scenario 2. Here, the state is on line 5 inside the iterator. Running the code from the previous example 2.15 using this implementation produces the expected output straight away. The distinction lies in the fact that the object does not need to be copyable. Each call to `[Symbol.iterator]` creates a new state. Apart from the iterator, this iterable is immutable. Another advantage is that all Sequence library operators can handle any object conforming to the JS iteration protocols.

```
 1 const SampleIterable = () => {
 2
 3   return {
 4     [Symbol.iterator]: () => {
 5       let i = 0;
 6       return {
 7         next: () => ({ done: i > 5, value: i++ })
 8       };
 9     };
10   };
11
12 };
```

Listing 2.18: Scenario 2 - State in closure scope of iterator

Comparing the two Scenarios

As mentioned in section 2.1.2: "Types of Iterables", there is a reason for the deep nesting of the iterator into the iterable object. Each iterator can have its own state and the iterable itself is immutable!
In programming, immutability refers to creating objects that cannot be changed once they are created. There are several advantages to using immutability in code:
Firstly, it helps in creating more reliable and bug-free programs. Since immutable objects cannot be modified, there is no risk of accidental changes or unintended side effects. This simplifies the code and makes it easier to reason about.
Secondly, immutability promotes functional programming practices. Immutable data structures are crucial in functional programming, allowing for more predictable and declarative code that is easier to test.

In scenario 2, the immutable approach has a specific impact on an iterable. When an iterable is immutable, each time we access its iterator, a new iterator instance is returned. This means it is impossible to process an iterable partially or in multiple stages. To achieve this behaviour, we need to implement scenario 1.
Because of the significant benefits of immutability and, consequently, the advantages of scenario 2, the Sequence library uses this concept. This way, developers who work with sequences can be confident that they always operate with a valid and unused instance that does what they expect.

2.3.4. Adding Monadic Functions to Iterables

This last section of this chapter explains the function `createMonadicSequence`, which the previous sections left out. Many of the operations of the Sequence library use this function for creating new sequences.

A Convenience Function

Consider listing 2.13 once again showing the implementation of `map`. This implementation uses a function `mapIterator`, which returns an object containing a function `next`. Thus, `mapIterator` forms an iterator. `createMonadicSequence` now receives such an iterator as parameter. Listing 2.19 demonstrates the implementation of `createMonadicSequence`:

```
/**
 * Builds an {@link SequenceType} from a given {@link Iterator}.
 * @template _T_
 * @param { () => Iterator<_T_> } iterator
 * @returns { SequenceType<_T_> }
 */
 const createMonadicSequence = iterator => {
    const result = {[Symbol.iterator]: iterator};
    return setPrototype(result);
};
```

Listing 2.19: Implementation of createMonadicSequence

Line 7 to 10 implement the functionality of `createMonadicSequence`:

- Assign the passed iterator to the `[Symbol.iterator]` property
- Add the sequence prototype to the resulting object

Sequence Prototype

“Prototypes are the mechanism by which JavaScript objects inherit features from one another.” [19] Multiple objects can use the same implementation of functions simultaneously if they have the same prototype.
Note: An object can have just *one* prototype. Hierarchies emerge through chaining prototypes, meaning setting the prototype of a prototype.

A prototype is just a function. A static function `setPrototypeOf` adds this prototype to an existing object. Thus, any properties set on the prototype become available to that object. These properties can access the current object directly with the keyword `this`. Listing 2.20 shows how this works for any object:

```
// The prototype is just a function
const Prototype = () => null;

// Add a new function to this prototype
Prototype.getName = function () { return this.name; };

// Creating an object with a name
const tw = { name : "Tobias Wyss" };


// Setting the prototype
Object.setPrototypeOf(tw, Prototype);

console.log(tw.getName());
// => Logs 'Tobias Wyss'
```

Listing 2.20: Setting the prototype to an object

The function `setPrototype` on line 7 of listing 2.21 adds the sequence prototype to an iterable using the function `setPrototypeOf`. This causes some functions to be accessible on each sequence using the "`.`" (dot) notation. These are, among others, monadic functions. You will read more about monads and their purpose in

```
1 /**
2  *
3  * @template _T_
4  * @param { Iterable<_T_> } iterable
5  * @returns { SequenceType<_T_> }
6  */
7  const setPrototype = iterable => {
8   Object.setPrototypeOf(iterable, SequencePrototype);
```

```
  return /**@type SequenceType*/ iterable;
};
```

Listing 2.21: Implementation of setPrototype

JSDoc for the SequenceType

JSDoc is the optional type system of JavaScript [20]. As the name says, it is pure documentation, therefore, not enforceable. Since newer IDEs support it very well, it is still very reasonable.
The `SequenceType` is the JSDoc representation of a sequence. Listing 2.22 shows the definition of the type. Alongside, the monadic functions `toString`, `pipe`, and equals `"=="` are also accessible via the prototype. Having these functions on each `SequenceType` is very convenient, as they improve readability and usability.

```
/**
 * @template _T_
 * @typedef {
 * {
 *   and:  <_U_>(bindFn: (_T_)     => SequenceType<_U_>) => SequenceType<_U_>,
 *   pure: <_U_>(_U_)              => SequenceType<_U_>,
 *   fmap: <_U_>(f: (_T_) => _U_) => SequenceType<_U_>,
 *   empty:     ()                 => SequenceType<_T_>,
 *   pipe: function(...transformers: SequenceOperation<*,*>): SequenceType<*>|*,
 *   toString:  ()                 => String,
 *   "==":      (that:SequenceType<_T_>) => Boolean
 *  } & Iterable<_T_>
 * } SequenceType
 */
```

Listing 2.22: SequenceType definition

This approach ensures that sequences become monadic but still iterable. Monadic APIs can thus also process sequences. An example of such an API is JINQ, which section 7.5: "JINQ" discusses in detail.

SequenceType vs. Iterable

The `SequenceType` describes iterable objects having their prototype set to `SequencePrototype`. An object of type iterable is just iterable. Every operator of the Sequence library receives an iterable as a parameter and returns an object of type `SequenceType`.

Turning an Iterable into a SequenceType

Sometimes the `SequenceType` must be set to an arbitrary iterable. For example, if you want to use the monadic functions on a JavaScript array. To make this possible, the Sequence library provides a `toMonadicIterable` function. Listing 2.23 shows the implementation of it. This function takes an arbitrary iterable and returns a `SequenceType` mapped by `id`[3]. Therefore, the returned object is then of the desired type `SequenceType`.

```
/**
 * Casts an arbitrary {@link Iterable} into the {@link SequenceType}.
 * @template _T_
 * @param { Iterable<_T_> } iterable
 * @return { SequenceType<_T_> }
 */
const toMonadicIterable = iterable => map(id)(iterable);
```

Listing 2.23: Casting an arbitrary iterable to SequenceType

2.4. Iterables Everywhere

The JS iteration protocols are suitable for a wide variety of data structures. With the Sequence library defining many operations for iterables, it makes sense to make more objects iterable.
The Kolibri Web UI Toolkit defines the immutable data structure "pair", for which there are noteworthy applications if it is iterable.

2.4.1. Making immutable Data Structures iterable

Listing 2.24 defines the type pair and shows its usage:

```
/**
 * @typedef PairType
 * @type {  <_T_, _U_>
 *          (x: _T_)
 *       => (y: _U_)
 *       => (s: PairSelectorType<_T_, _U_>) => ( _T_ | _U_ )
 *      }
 */
const Pair = x => y => selector => selector(x)(y);

const pair = Pair(1)(2);

const one  = pair(fst);
const two  = pair(snd);

console.log(one + " " + two);
// => Logs '1 2'
```

Listing 2.24: Immutable pair

[3]The id (identity) function always returns the value passed to it.

The only way to make a pair immutable is to build it using functions. The type signature on line 1 to 8 shows that the first two arguments are arbitrary values. Pair stores these two values in its closure scope, the only scope in JavaScript that can not be changed in any way from outside. Selector functions named `fst` on line 13 and `snd` on line 14 grant access to these values. However, pair offers no way to modify these values. Listing 2.24 shows that handling a pair can be cumbersome. It would be great to use the built-in JavaScript language features to access the elements of a pair.

Iterable Pair

Listing 2.25 demonstrates the implementation of an iterable pair. Still, pair operates only with functions. However, it additionally defines the `[Symbol.iterator]` property:

```
1  const Pair = x => y => {
2    /**
3     * @template _T_, _U_
4     * @type { PairSelectorType<_T_,_U_> }
5     */
6    const pair = selector => selector(x)(y);
7
8    pair[Symbol.iterator] = () => [x,y][Symbol.iterator]();
9
10   return pair;
11 };
```

Listing 2.25: Iterable Pair

Line 8 shows that this property defines a function, that only returns the `[Symbol.iterator]` property of an array. The array stores the values of the pair. With that, pairs are now iterable. Listing 2.26 shows the usage of such an iterable pair. Line 3 shows the deconstruction of a pair in the same way as an array. This access option is more convenient than in listing 2.24. Line 5 demonstrates the usage of operations from the Sequence library alongside a pair. `show` converts an iterable into a string, analogous to how `toString` works.

```
1 const pair = Pair(1)(2);
2
3 const [one, two] = pair;
4
5 console.log(show(pair));
6 // => Logs '[1,2]'
```

Listing 2.26: Working with iterable pairs

This has significant advantages because it is now possible to process different collections with the same abstractions. Therefore, the motivation is great to make all collections iterable.

2.5. Conclusion

Based on the JS iteration protocols, sequences are the foundation for many valuable functionalities that benefit from functional programming in JavaScript. Basic constructs such as a range introduce fresh opportunities for data processing. Combining these data structures with the decorator approach and its corresponding extensibility options forms a robust library called "Sequence library". The Sequence library enables seeing programming problems from a new angle, leading to fresh approaches and solutions.

3. Modularizing Programs

This chapter explains how to break down programs and functions into small, reusable pieces using lazy evaluation and higher-order functions. The start shows how Haskell implements these concepts. The subsequent discussion covers the application and limitations of these concepts in JavaScript before showing how the Sequence library overcomes these limitations, providing valuable tools for writing reliable and reusable code.

3.1. Gluing Programs together

John Hughes argues that functional programming languages provide two extra ways to better partition programs and easily combine them later. He refers to them as "glues" [7, p.3]. These are, on the one hand, higher-order functions and, on the other hand, lazy evaluation.

3.1.1. Using these Glues in Haskell

Higher-order Functions in Haskell

Higher-order functions receive another function as arguments or produce a function as a result. This concept is fundamental in Haskell, where the whole program is a single function. The easiest way to understand higher-order functions is to look at an example:

```haskell
double    = \x -> 2 * x
quadruple = \x -> 4 * x
doubles   = map double [1,2,3,4]
quads     = map quadruple [1,2,3,4]

print $ doubles ++ quads
-- Prints '[2,4,6,8,4,8,12,16]'
```

Listing 3.1: Higher-order functions in Haskell

Listing 3.1 shows how higher-order functions work in Haskell. This code is very reusable since `map` takes a function as an argument to transform the values of a list.

Lazy Evaluation

Lazy evaluation describes a concept where a program only evaluates as many statements as required. This concept makes it possible to work with lists that have an infinite amount of values. Again, an example explains the concept best:

```haskell
1 as = [1..]
2 doubles = map double as
3
4 print $ take 4 doubles
5 -- Prints '[2,4,6,8]'
```

Listing 3.2: lazy evaluation in Haskell

Line 1 of listing 3.2 defines a list with infinite values. Then the function `double`, defined in the previous listing 3.1, is applied to this list. As the list has infinite elements, eagerly evaluating it would take forever. But since Haskell evaluates its statements lazily, nothing happens on line 2. Haskell applies `double` not before line 4, where `print` evaluates a part of the list (the first four elements). Even then, Haskell exclusively uses `double` on those four elements.
More theoretically, when applying the function f to $g(x)$, g produces only as much output as f actually needs. Therefore, working with large data sets becomes much more convenient and faster. Suppose the elements in a list are generated with a rule (as with list comprehensions or generators). Then lazy evaluation brings excellent advantages: The whole list will never materialize in the memory, only the current value. So you don't have to worry about what happens when large amounts of data are processed.
Using these two concepts, building large programs consisting of small parts is much easier.

3.1.2. Using these Glues in JavaScript

Higher-Order Functions in JavaScript

Higher-order functions in JavaScript work similarly as they do in Haskell. Transferring listing 3.1 to JavaScript results in the following:

```javascript
const double    = x => 2 * x;
const quadruple = x => 4 * x;
const doubles   = [1,2,3,4].map(double);
const quads     = [1,2,3,4].map(quadruple);

console.log(doubles.concat(quads));
// => Logs '2, 4, 6, 8, 4, 8, 12, 16'
```

Listing 3.3: Higher-order functions in JavaScript

Eager Evaluation in JavaScript

It becomes less intuitive when it comes to lazy evaluation in JavaScript: JavaScript does not know this concept when working with its in-built arrays. Mapping over an array producing a side effect can shed some light on this:

```
const as     = [1,2,3,4,5];
const bs     = [];
const mapped = as.map(x => {
  bs.push(x);
  return 2*x;
});

console.log(mapped.slice(0,4));
// => Logs '2, 4, 6, 8'
console.log(bs);
// => Logs '2, 4, 6, 8, 10'
```

Listing 3.4: JavaScript evaluates eagerly

Listing 3.4 clarifies that even though the program only uses the first four elements of the array, it traverses each element using the passed function. This gets clear because the array `bs` contains all array elements of `as` and not just its first four.

Lazy Evaluating Iterables

The eager evaluation of JavaScript is a significant limitation. Fortunately, JavaScript can emulate this laziness using the JS iteration protocols, which section 2.1: "Iterators in General" describes.

With the use of the Sequence library, it is possible to rewrite the JavaScript code of listing 3.4 lazily:

```
const as     = [1,2,3,4,5];
const bs     = [];
const mapped = map(x => {
  bs.push(x);
  return 2*x;
})(as);

console.log(...take(4)(mapped));
// => Logs '2, 4, 6, 8'
console.log(bs);
// => Logs '1, 2, 3, 4'
```

Listing 3.5: Lazy evaluation in JavaScript

The first output is the same as in listing 3.4 - the first four elements of the array `as` have been multiplied by 2. However, the second output differs: the array `bs` now contains only four elements. So

`map` applied the passed function only to four initial array elements. This fact proves that transforming arrays using the Sequence library happens lazily.
This works because the JS iteration protocols process element by element. The iterator delivers the next element only if it is requested. If no further elements are needed, it just stops iterating.
Iterators, therefore, decouple the termination condition (asking for another element) from the loop body (generating/accessing the next element).

Furthermore, it is even possible to port the initial Haskell laziness example defined in listing 3.2 to JavaScript:

```
1 const seq     = Sequence(1, _ => true, x => x + 1);
2 const doubles = map(double)(seq);
3
4 console.log(...take(4)(doubles));
5 // => Logs '2, 4, 6, 8'
```

Listing 3.6: Process infinite sequences in JavaScript

If the code displayed in Listing 3.6 were evaluated eagerly, this would go on forever since line 1 defines a sequence with infinite elements.

3.1.3. Lazy Evaluation and Side Effects

Decoupling the loop body and termination condition helps to consider these two concepts separately. This split reasoning makes sense since they often do not relate to each other. Why change the whole list of elements if the program only uses the first five? In JavaScript, however, this thinking carries a danger that cannot occur in pure functions like those used in Haskell.
The following listing 3.7 states the problem:

```
let i = 0;
const mapped = map(x => {
  i++;
  return 2 * x;
})([1,2,3,4]);

console.log(i);
// => Logs '0' i has not been incremented
console.log(...take(2)(mapped));
// => Logs '2, 4'
console.log(i);
// => Logs '2' i has only been incremented two times
```

Listing 3.7: Side effects and lazy evaluation

It is impossible to conclude from the length of the list how often a function runs. Performing side effects in the loop body can lead to unexpected behaviour.

It is tough to make assumptions about `i`, because:

- `i` is only calculated when the program is evaluated
- It is impossible to predict what value `i` will have when the program finishes.

This example shows that lazy evaluation and side effects almost exclude each other. Therefore, omitting side effects when working with lazy evaluation makes sense. In Haskell, where the type system can exclude side effects per function definition, this is not a problem. Since side effects often make a program harder to understand, getting rid of it feels even more natural.

3.2. Placing the Receiver at the End

Compared to object-oriented programming languages, where the receiver of an operation is always placed before the point (i.e., at the beginning), functions in Haskell usually expect it as the last argument. If one sticks to this, it is possible to give a new name to a partially applied (pre-configured) function. Line 2 of the following Haskell listing 3.8 does precisely this and results in a reusable pre-configured function (`doubleAll`) with a very descriptive name:

```
1 double    = \x -> 2 * x
2 doubleAll = map double -- preconfigure map with the function double
3 as        = [1,2,3,4]
4 bs        = [5,6,7,8]
5
6 print $ doubleAll as ++ doubleAll bs
7 -- Prints '[2,4,6,8,10,12,14,16]'
```

Listing 3.8: Pre-configured functions in Haskell

This paradigm allows it also to link existing functions together using the `.` (dot) operator:

```
-- "sum" sums up all elements in the list
doubleAndSum = sum . doubleAll

print $ doubleAndSum [1,2,3,4]
-- Prints '20'
```

Listing 3.9: Function composition in Haskell

Listing 3.9 links the functions `doubleAll` and `sum` to create a new function applicable to any list with numerical values. This concept makes the code very reusable.
Placing the receiver at the end is another powerful modularization tool. Therefore, the Sequence library supports this kind of modularization. Refactoring the initial example from listing 3.3, which shows how higher-order functions work in JavaScript, results in the following:

```
// initial code
const double  = x => 2 * x;
const doubles = [1,2,3,4].map(double);

// refactored using plain JS
const doubleAll = receiver => receiver.map(double);
```

```
console.log(doubleAll([1, 2, 3, 4]));
// => Logs '2, 4, 6, 8'
```

Listing 3.10: Receiver before the point

Listing 3.10 fixes this issue using a function that gets the receiver as an argument. However, this is not very convenient. The Sequence library provides a more elegant way to solve this:

```
const doubleAll = map(double);
```

Listing 3.11: Place the receiver at the end in JavaScript

The code of listing 3.11 is not only less to type but also more accurate when reading because the term `doubleAll = map(double)` says that the pre-configured function `map(double)` gets a new name (`doubleAll`). No worrying about parameter naming is needed here.

Conclusion

Higher-order functions and lazy evaluation are powerful glues to compose smaller parts into a more extensive program. Haskell provides a good reference for this. In JavaScript, higher-order functions are familiar. However, it offers less support for lazy evaluation - it eagerly evaluates arrays and their operations. The Sequence library closes this gap because sequences work similarly to Haskell lists. Thus, the Sequence library allows using both types of glue in JavaScript.

4. Monads in JavaScript

The first part of this section describes what contexts in Haskell are and how Haskell wraps values in such contexts. "Maybe" serves as an example of this. Implementing this example using JavaScript shows that such contexts are also useful in other programming languages.
The second part then explains monads and describes how to implement them with JavaScript. It also highlights the missing features of JavaScript to adopt this concept exactly from Haskell and describes acceptable workarounds. Thus, it is also possible to transform values in a context in JavaScript. The last part shows what the monadic operations of the sequence look like.

4.1. Wrapping Values in a Context

In Haskell, you often work with values wrapped in a particular context. This context can be, for example, a list. Nevertheless, this context can also be another data structure, for instance, the polymorphic type `Maybe`.
For the context `Maybe` two implementations in Haskell co-exist:

```haskell
-- defining the datatype Maybe
data Maybe a = Nothing | Just a
```

Listing 4.1: The data type Maybe in Haskell

Listing 4.1 defines the datatype `Maybe`. Why is this useful? Imagine you want to create a function `head` which returns the first value of a given list. `head` is pretty simple to implement. But wait, what to do when the list is empty? In object-oriented languages, one might return `null`. The problem with this solution is that the user of `head` must remember that the list can be empty, and thus the result of `head` can be `null`. This is very error-prone.

That is where the new datatype `Maybe` comes in. `Maybe` allows describing either the absence of a value or the value itself. The following listing 4.2 defines a new function `safeHead` (line 2), which does precisely this - it returns `Just a` with a containing the first value of the list when it is not empty (line 3) and `Nothing` otherwise (line 4):

```haskell
1  -- a function which produces a Maybe:
2  safeHead :: [a] -> Maybe a
3  safeHead (a:_) = Just a
4  safeHead []    = Nothing
5
6  printHead list = print $ case (safeHead list) of
7    Just val -> show val
8    Nothing  -> "List was empty!"
9
10 printHead [1,2,3,4]
```

```
-- Prints '"1"'
printHead []
-- Prints '"List was empty!"'
```

Listing 4.2: Safely get the first element of a list

The function `printHead` is forced to deal explicitly with the case when the passed list is empty. When returning `null`, the user must remember to deal with the case where there is no first element. Therefore, `Maybe` leads to improved safety.

4.1.1. Doing the Same in JavaScript

Listing 4.3 defines the same type in JavaScript:

```
1 const Just    = x => _f => g => g(x);
2 const Nothing = f => _g => f(undefined);
```

Listing 4.3: The data type Maybe in JavaScript

So `Just` and `Nothing` are only functions. How could it be different in functional programming?
Line 1 defines the `Just` case: `Just` takes a value `x` and two functions while not using the first function at all. `Just` applies the second function to the initial value `x`.
`Nothing` looks very similar to `Just`, with the difference that it does not receive a value. Furthermore, `Nothing` calls the second function at the end.
Now how to use this? Noticeably, `Just(value)` and `Nothing` have the same structure: both receive two functions as arguments. Meaning they are structurally the same. Listing 4.4 takes advantage of this property to port the previous function `safeHead` to JavaScript:

```
1  const safeHead = list => {
2    const head = list[0];
3    return (undefined === head) ? Just(head) : Nothing;
4  };
5
6  const printHead = list => {
7    const maybeHead = safeHead(list);
8    maybeHead
9      (_ => console.log("List was empty!")) // Nothing case
10     (head => console.log(head));          // Just case
11 };
12
13 printHead([1,2,3,4]);
14 // => Logs '1'
15 printHead([]);
16 // => Logs 'List was empty!'
```

Listing 4.4: safeHead implemented in JavaScript

Using the structural similarity of `Just(value)` and `Nothing`, lines 8 to 10 evaluate the result of `safeHead`. The function `printHead`, therefore, works for both cases.

4.2. Working with Values in a Context

4.2.1. Introducing fmap

The example with maybe shows how values work in a context. Over time, however, it can become tedious to keep making this distinction whether it has a value. Therefore, Haskell offers a concept of working with values in a context. The simplest way to work with the value using this concept is using the function `fmap`. `fmap` knows how to execute a function for the value(s) in a context. Listing 4.5 shows the usage of `fmap` for maybe:

```haskell
double = \x -> 2 * x
print $ fmap double (Just 5)
-- Prints 'Just 10'
print $ fmap double Nothing
-- Prints 'Nothing'
```

Listing 4.5: fmap applied to Maybe

Similar to maybe, also lists describe such a context. Listing 4.6 shows how `fmap` works on lists:

```haskell
print $ fmap double [1,2,3,4,5]
-- Prints '[2,4,6,8,10]'
```

Listing 4.6: fmap applied to a list

Applying `fmap` to a list has the same result as applying `map` to a list. Therefore, `fmap` is just a way to "map values".

4.2.2. Making a Context Monadic

Haskell defines many other functions analogous to `fmap` to interact with values in a context. So-called type-classes define these functions. In his book "Programming in Haskell" Hutton describes type-classes like "[...], a class is a collection of types that support certain overloaded operations called methods. " [21, p. 31]
Some of these overloaded methods are more powerful than `fmap`. The following section briefly overviews which operations a context must provide to be considered monadic:

- **Functor**: A context `f` is a `Functor` when it provides a function `fmap`. `fmap` receives another function as an argument and applies it to the value(s) in the context.
  `fmap` signature: `fmap :: Functor f => (a -> b) -> f a -> f b`
- **Applicative**: A context `f` is an `Applicative` when it is a `Functor` and additionally provides two functions:
  - `<*>` (pronounced "app") receives another function as an argument wrapped in the same context and applies the unwrapped function to the value(s) in the context.
    `<*>` signature: `(<*>) :: Applicative f => f (a -> b) -> f a -> f b`

  - `pure` receives a single argument and wraps it in the context. When describing `pure`, people often use the term lifting instead of wrapping. `pure` "lifts" a value into a context.
    `pure` signature: `pure :: Applicative f => a -> f a`

- **Monad**: A context `m` is a `Monad` when it is an `Applicative` and additionally provides a function >>= (pronounced "bind"). >>= takes another function as an argument which, when applied to the value(s), again creates a value in the same context. So that the values are not nested twice in the context, >>= must resolve the inner context again. Therefore, >>= is like `fmap` but after mapping, the value(s) get flattened. Therefore, this operation is often also referred to as `flatMap`.
  >>= signature: `(>>=) :: Monad m => m a -> (a -> m b) -> m b`

This list should briefly overview this hierarchy of type classes. For a little deeper introduction to this topic, see, for example, [22]. To really dive into it and see the rules each function follows, please see [21, Ch. 12].

Why using these Abstractions?

These abstractions allow building general functions that can handle various types. Listing 4.7 shows the function `doubleAll` defined in listing 3.9 in a more general way, which is applicable to lists and also maybes:

```haskell
doubleAll :: (Functor f) => f Int -> f Int
doubleAll = fmap double

print $ doubleAll [1,2,3,4]
-- Prints '[2,4,6,8]'
print $ doubleAll $ Just 1
-- Prints 'Just 2'
```

Listing 4.7: Double the values in a context

Even though a list and a maybe have nothing in common, the same function can be applied to both types.
Note: Another example that makes use of these abstractions is JINQ, which is described in the section 4.4: "Query different Data Structures".

4.3. Monads in JavaScript

Since monads are so good at handling values in a context, this concept is also interesting for JavaScript. However, JavaScript has a weaker type system than Haskell. Therefore, two important concepts can not be transferred to JavaScript:

1. In Haskell, it is possible to force the type hierarchy hinted in section 4.2.2: "Making a Context Monadic".

2. Haskell can use its type system to determine a specific implementation for a function. The listing 4.8 shows the binding of the function body to the name using the example of `fmap`:

```haskell
-- The implementation for List is used
print $ fmap double [1,2,3,4]
-- The implementation for Maybe is used
print $ fmap double (Just 1)

```

Listing 4.8: Haskell determines the correct function body

Since it is not possible to implement these two concepts in JavaScript, a different approach to these two problems is needed:

1. Although it is possible to model a type hierarchy in JavaScript, enforcing it is impossible. Therefore, we created only one JSDoc type, the "`MonadType`". `MonadType` can be used as an interface, which defines all operations a monadic type must support.
2. Instead of inferring global functions automatically, the prototype of a monadic object gives access to them. [19]

Section 4.3.2: "Using JSDoc to type Monads" describes these workarounds in detail.

4.3.1. Which Operations fit JavaScript?

Since JavaScript works quite differently than Haskell, not all operations are equally suitable to adopt. The `MonadType` specifies the following operations:

- `fmap`: Changing values inside a context is a typical pattern, also in JavaScript. Porting `fmap` to JavaScript brings many benefits.
- `pure`: At first sight, `pure` is a function that is not needed. However, it quickly became apparent that lifting an element into a context is often practical via an abstracted function.
  This is often the case when using >>=.
- `and (>>=)`: The `bind` operator allows access to the result of the last computation. `bind` is the only way to determine a new result depending on the previous result.
  Since in JavaScript function names must not contain the special characters > and =, >>= cannot be used. `bind` is also already an existing function on every object. Using another term is required, therefore. We used the name `and` because it nicely expresses that the following result depends on the previous one.
- `empty`: The `empty` function creates a context without a value. For example, with `Maybe` it is `Nothing` with `List` it is `[]`. A monad does not need to provide a function `empty`, so section 4.2.2: "Making a Context Monadic" does not include it. However, `empty` is a function available on many monads and is very handy during the programming of generic functions. Therefore, the `MonadType` specifies `empty` as well.

What about the app Operator (<*>)?

The operator <*> is great when applying a function wrapped in a context to multiple values wrapped in the same context. This use-case is relatively rare in JavaScript because you do not operate as strictly in contexts as in Haskell. In addition, the app operator also only works on functions that receive their arguments curried. Most functions in JavaScript do not. Using the JavaScript maybe as an example, listing 4.9 shows how <*> would work:

```
1  const xs        = [1, 2, 3];    // array with x values
2  const ys        = [4, 5, 6];    // array with y values
3  const maybeX    = safeHead(xs); // getting the head of xs (4.4)
4  const maybeY    = safeHead(ys); // getting the head of ys
5
6  const plus      = x => y => x + y;
7  const maybePlus = Just(plus);
8
9  const sum       = maybePlus.app(maybeX).app(maybeY);
10 sum
11   (_ => console.log("Something went wrong") // Nothing case
12   (s => console.log(s));                    // Just case
13 // => Logs '5'
```

Listing 4.9: The app operator in JavaScript used on Maybe

Suppose you have two lists from which you want to add the first two elements. Of course, these lists can be empty. safeHead fetches the first elements safely. The function plus on line 6 allows you to sum up two numbers. However, these two addends are now in a context. So plus cannot be applied directly to the values. That is when app comes into play. app expects a parameter in the same context as the original wrapped function. Line 7, therefore, wraps plus in Just. So it is now also in the maybe context. Line 9 repeatedly calls app on the wrapped function to pass parameter after parameter. In the end, the variable sum contains the result of the addition. If one of the addends was Nothing because one list was empty, app deals with that, and the whole result evaluates to Nothing.
As said above, this only works because plus receives its arguments in a curried style. As listing 1.1 explains, applying plus to x returns a function taking another argument. Calling this function with another number returns the sum.

Even if this pattern appears less, it is still useful sometimes. Luckily "and" is capable of everything app can do. So consider listing 4.10, which does the same thing using and, even though it is a bit more to type:

```
// maybeX, maybeY and maybePlus defined in Listing 4.9
const maybeSumX = maybePlus.and(p => maybeX.fmap(p));   // plus(x)
const maybeSum  = maybeSumX.and(pX => maybeY.fmap(pX)); // plus(x)(y)

maybeSum
  (_ => console.log("Something went wrong") // Nothing case
  (s => console.log(s));                    // Just case
// => Logs '5'
```

Listing 4.10: The app operator replaced by and

Line 2 shows how the function `and` gives access to the last element, which is `p`, the function `plus`. `fmap` applies this function to `maybeX`. This creates a new function which is again in the maybe context. Line 3 does exactly the same - but for the second argument.
This shows that `and` can emulate the behaviour of `app`.

4.3.2. Using JSDoc to type Monads

A type system must represent monads to unveil their true power. With some limitations, JSDoc can model monads in a newly created type called `MonadType`.

The MonadType

The `MonadType` combines all the operations described in section 4.3.1: "Which Operations fit JavaScript?" into a single JSDoc type. It serves as a structural interface that functions can use to describe their parameters or return values.

```
/**
 * Defines a Monad.
 * @template  _T_
 * @typedef  MonadType
 * @property {<_U_>(bindFn:(_T_) => MonadType<_U_>) => MonadType<_U_>} and
 * @property {<_U_>(f:     (_T_) => _U_)            => MonadType<_U_>} fmap
 * @property {     (_T_)                            => MonadType<_T_>} pure
 * @property {     ()                               => MonadType<_T_>} empty
 */
```

Listing 4.11: The MonadType

Listing 4.11 shows this type with the four available operations `and`, `fmap`, `pure` and `empty`. `and` as well as `fmap` *do* something with the current value(s) wrapped in this context. `pure` and `empty` work more like static operations. They are accessible using the object but have no direct connection to its properties.

Listing 4.12 defines a function `keepEven` which discards odd values in the context. As a parameter, this function accepts every context adhering to `MonadType`. Therefore, `keepEven` can operate on every monadic object and access all its monadic operations.
The `pure` and `empty` functions are independent of the values inside of the context, but they are still called on the passed monad so that JavaScript can find the correct implementation:

```

/**
 * @param { MonadType<Number> } monad
 * @returns MonadType<Number>
 */
const keepEven = monad =>
  monad
   .and(x => {
    if (x % 2 === 0) return monad.pure(x);
    else return monad.empty();
  });
```

Listing 4.12: Usage of the MonadType

Listing 4.13 shows how `keepEven` can be called with a maybe. This works because maybe is a monad. If it contains an odd number, `keepEven` discards it. The best aspect is that `Nothing` does not need any special treatment - the and implementation of maybe knows what to do when a `Nothing` appears. So it is not the job of `keepEven` to handle such special cases.

```
/** Prints the value of a Maybe if it exists */
const evalMaybe = maybe =>
  maybe
  (_ => console.log("There was no value!"))
  (x => console.log(x));

const maybe1 = Just(1);
const maybe2 = Just(2);
const maybe3 = Nothing;

evalMaybe(keepEven(maybe1));
// => Logs 'There was no value!'
evalMaybe(keepEven(maybe2));
// => Logs '2'
evalMaybe(keepEven(maybe3));
// => Logs 'There was no value!'
```

Listing 4.13: keepEven called with a Maybe

The same works for a sequence - only because it is a monad. `keepEven` discards every odd number of this sequence:

```
const seq = Sequence(0, i => i < 5, i => i + 1);

console.log(...keepEven(seq));
// => Logs '0, 2, 4'
```

Listing 4.14: KeepEven called with a Sequence

The example shows well how one can program very declaratively using monads. You do not have to worry about the actual structure of the context but only about the logic being correct. `keepEven` can also be tested easily: you create a new type that conforms to `MonadType`, which has little logic.

4.3.3. Implementing Monadic Operations in JavaScript

Making a context conform to `MonadType` requires the following steps:

1. Creating a prototype for the context (see section 2.3.4: "Sequence Prototype" which explains prototypes).
2. Defining the four operations specified by `MonadType` on this prototype.
3. Specify the JSDoc of this context.

Note: Defining the monadic operations on a prototype is mostly a good idea, since there are often multiple implementations of a type, like it is in maybe with `Just` and `Nothing`. This way, it becomes possible to share the implementations among them.

Defining the Monadic Operations

Listing 4.15 uses the example of maybe to show how to implement the monadic operations `fmap`, `pure`, `empty`, and `and`.

```
 1 const MaybePrototype = () => undefined;
 2
 3 MaybePrototype.pure = val => Just(val);
 4
 5 MaybePrototype.empty = () => Nothing;
 6
 7 MaybePrototype.and = function (bindFn) {
 8   let returnVal;
 9   this
10     (_ => returnVal = Nothing)
11     (x => returnVal = bindFn(x));
12   return returnVal;
13 };
14
15 MaybePrototype.fmap = function (mapper) {
16   return this.and(x => Just(mapper(x)));
17 };
```

Listing 4.15: Making Maybe monadic

This code does the following:

- Line 3 defines `pure` - the given value is wrapped by `Just`.
- Line 5 defines `empty` - no value is in this context.
- Line 7 and following define `and` - if a value is present, `and` applies the function `bindFn` to this value and returns the result. If there is no value, `and` returns `Nothing` directly.
- Line 15 and following define `fmap` - `and` is used to transform the current value. Since `bindFn` needs to return a maybe again, the result of `mapper` gets wrapped with `Just`.

Both `Just(value)` and `Nothing` must provide the respective operations as properties. Listing 4.16 now sets the prototype to the functions `Just(value)` and `Nothing`:

```
const Nothing = f => _g => f(undefined);
Object.setPrototypeOf(Nothing, MaybePrototype);
```

```
3
4 const Just = x => {
5   const just = _f => g => g(x);
6   Object.setPrototypeOf(just, MaybePrototype);
7   return just;
8 };
```

Listing 4.16: Setting the prototype of Maybe

The definition of `Nothing` on line 1 stays exactly the same as before. But consider line 4-8. These changed because not `Just` should become a `MonadType` but `Just(value)`.

To make another object monadic, apply the same pattern to it.

Specifying the JSDoc

JSDoc types structurally. We exploit this to extend existing types with the monadic type. Such complemented types are then compatible with functions that expect `MonadTypes` as parameters.
Listing 4.17 defines the `MonadType` JSDoc for Maybe:

```
1  /**
2   * @template _T_
3   * @typedef {
4   *              ((f:<omitted>)  => (g:<omitted>) => _T_)
5                 & MaybeMonadType<_T_>
6              } MaybeType
7   */
8
9  /**
10  * @typedef MaybeMonadType
11  * @template _T_
12  * @property { <_V_> ((_T_) => MaybeType<_V_>) => MaybeType<_V_> } and
13  * @property { <_V_> ((_T_) => _V_)            => MaybeType<_V_> } fmap
14  * @property { <_V_> (_V_)                     => MaybeType<_V_> } pure
15  * @property {       ()                        => MaybeType<_T_> } empty
16  */
```

Listing 4.17: Adding JSDoc to Maybe

What happens here?
Line 1 and the following specify the updated `MaybeType`. `MaybeType` is a function which receives two other functions as parameters (line 4). And it is also everything that is specified by `MaybeMonadType` (line 5).
This & sign allows intersecting two JSDoc type definitions into a new one.

Using such an intersection type brings the advantage of splitting up type declarations, which makes it easier to extend types and also to reuse specific parts of a type definition.

Limitations of JSDoc

You may have recognized that MaybeMonadType and MonadType look almost identical. So why not drop the specification for MaybeMonadType and directly use MonadType? The reason is simple: MonadType is less specific than MaybeMonadType - every operation on MonadType returns a MonadType again.
Consider line 5 in listing 4.18, where the MaybeType is defined using the more generic MonadType (defined in listing 4.11) instead of the MaybeMonadType (defined in listing 4.17):

```
 1 /**
 2  * @template _T_
 3  * @typedef {
 4  *              ((f:<omitted>)  => (g:<omitted>) => _T_)
 5              & MonadType<_T_>
 6             } MaybeType
 7  */
 8 const maybe        = Just(5);              // MaybeType
 9 const mappedMaybe  = maybe.fmap(x => 2*x); // MonadType
10 
11 // produces a type error
12 evalMaybe(mappedMaybe); //  Defined in Listing 4.13
13 
14 // workaround using an explicit type cast
15 const mappedMaybe2 = /** @type { MaybeType } */ maybe.fmap(x => 2*x);
16 evalMaybe(mappedMaybe);
```

Listing 4.18: Why not use MonadType with Maybe?

Since fmap is defined by MonadType, line 9 loses the information that maybe was of MaybeType. Therefore, line 12 no longer accepts mappedMaybe since evalMaybe requires a MaybeType and not a MonadType. The workaround would be to add a typecast every time any monadic function has been called, like on line 15.

For this to work without type casting, JSDoc would have to support higher-kinded types [23], which would allow type definitions as defined by listing 4.19:

```
/**
 * Defines a Monad.
 * @typedef   MonadType
 * @template  { MonadType } _M_
 * @template  _T_
 * ...
 * <other functions omitted>
 * ...
 * @property  { <_U_>
 *                 ((_T_) => <_U_>)
 *              => _M_<_U_>
 *            } fmap
```

```
    */
```

Listing 4.19: Not working JSDoc types

Line 4 defines a generic type variable that has to conform to `MonadType`. So far, everything works with JSDoc.
But line 11 does not work anymore because it is not possible to abstract over another type using JSDoc.

4.3.4. Making the Sequence Monadic

Section 4.3.2: "Using JSDoc to type Monads" defines the function `keepEven`, which accepts any `MonadType` as an argument. Since the sequence is monadic, listing 4.14 applies `keepEven` to a sequence without defining how to implement the monadic operations on it. This section shows the implementation of the monadic operations for the sequence.
The signatures of the operations are analogous to those of maybe. However, they do not base on `MaybeMonadType` but on `SequenceMonadType`. Again, the carriers define these functions. `Object.setPrototypeOf` then adds it to the sequence.

Listing 4.20 shows the definition of these operations - it is straightforward:

```
/**
 * @template _T_
 * @returns SequenceType<_T_>
 */
SequencePrototype.empty = () =>
  Sequence(undefined, _ => false, _ => undefined);

/**
 * @template _T_
 * @param { _T_ } val - lifts a given value into the context
 * @returns SequenceType<_T_>
 */
SequencePrototype.pure = val => PureSequence(val);

/**
 * @template _T_, _U_
 * @param { (_T_) => _U_ } mapper - maps the value in the context
 * @returns SequenceType<_U_>
 */
SequencePrototype.fmap = function (mapper) {
  return map(mapper)(this);
};

/**
 * @template _T_, _U_
 * @param { (_T_) => SequenceType<_U_> } bindFn
```

```
27  * @returns { SequenceType<_U_> }
28  */
29 SequencePrototype.and = function (bindFn) {
30   return bind(bindFn)(this);
31 };
```

Listing 4.20: The monadic operations on the Sequence

- `empty` (line 6): An empty sequence contains no values and returns, therefore immediately `done: true` when calling `next` on its iterator.
- `pure` (line 13): The iterator of a sequence with one single value iterates only once, returning this value. `PureSequence` does precisely this.
- `fmap` (line 20): Mapping a sequence applies the given function to each element. Which is what `map` does.
- `and` (line 29): Bind on a sequence applies the given function to each element and then flattens it. This is what `bind` does.

4.4. Query different Data Structures

Based on the knowledge from the previous sections of this chapter, other possibilities analogous to `keepEven` become realizable - i.e., functions that handle arbitrary monadic structures.
This section introduces such a concept called Language Integrated Queries (LINQ), which queries any monadic data structure.

4.4.1. Introduction to LINQ

Some programming languages offer uniform ways to query different data structures in an SQL-like syntax.

C# calls this concept LINQ (Language Integrated Query). LINQ allows querying compatible data sources declaratively. Listing 4.21 shows how to use LINQ to query a simple array of numeric values.

```
// Specify the data source.
int[] scores = { 97, 92, 81, 60 };

// Define the query expression.
IEnumerable<int> scoreQuery =
    from score in scores
    where score > 80
    select score;

// Execute the query.
foreach (int i in scoreQuery)
{
    Console.Write(i + " ");
}

// Output: 97 92 81
```

Listing 4.21: LINQ in C# [6]

You can replace the data structure defined on line 2 with any other one compatible with this API. This abstraction makes it very easy to define reusable queries.

4.4.2. Why does this not exist in JavaScript?

However, JavaScript does not define a uniform API for data structures except for the JS iteration protocols and thus cannot offer language-integrated queries without further effort.
Section 4.3.4: "Making the Sequence Monadic" introduced the monadic sequence, which provides additional operations to work with a sequence. All of these operations are available through the sequence prototype. Since these operations work solely on the committed properties of the JS iteration protocols, the function `toMonadicIterable`, explained in section 2.4: "Iterables Everywhere", can quickly turn them into a monadic sequence.
With that, a more extensive API is now available to every data structure conforming to the JS iteration protocols. This monadic API makes it possible to implement abstractions similar to LINQ in JavaScript.

4.4.3. Introducing JINQ

JINQ (JavaScript integrated query) is the implementation of LINQ for JavaScript. It can handle any data that conforms to the `MonadType` explained in section 4.3.2: "The MonadType". So JINQ can handle monadic iterables and every monadic type, such as the type maybe introduced in section 4.1.1: "Doing the Same in JavaScript".
Note: All operations supported by JINQ are explained in detail in section 7.5: "JINQ". This section describes how JINQ works internally.

Listing 4.22 shows again the function `keepEven` already introduced in section 4.3.2: "The MonadType", which works on a maybe as well as on a sequence:

```
const keepEven = monad => monad
  .and(x => {
    if (x % 2 === 0) {
      return monad.pure(x);
    } else {
      return monad.empty();
    }
});
```

Listing 4.22: Recapitulate keepEven

JINQ makes it possible to simplify the implementation of this function. Listing 4.23 therefore introduces a new function `keepEvenJINQ`, which does precisely the same as `keepEven`:

```
const keepEvenJINQ = monad =>
  from(monad)
    .where(x => x % 2 === 0)
    .result();
```

Listing 4.23: keepEvenJINQ does the same as keepEven

`keepEvenJINQ` is not only shorter but also more readable. It becomes clear within moments what this function does, as it reads almost like prose.
This is the power of these abstractions - types must only follow a minimal API to be compatible with JINQ. Additionally, they are very readable.

Note: As explained before, JINQ works only on the monadic API. You can use the monadic functions to achieve the same. However, the example `keepEven` shows that it is easier to work with JINQ and save lines of code.

How does JINQ work?

JINQ uses a pattern analogue to the Builder pattern [16, Ch. 3.2] to create a new structure which can be used to transform the initially passed monad.
Have a look at the function on line 3 of listing 4.24: If `from` is called, a new builder is created. `from` expects a monad, which serves as the starting point of the builder. The next operation executed on the builder operates on this monad. `result` then returns the monad created based on the builder.

Note: None of the functions change the parameter `monad` - therefore, JINQ is immutable. This allows reusing an intermediate state of the JINQ builder.

```
 1 // jinq.js
 2 export { from }
 3 const jinq = monad => ({
 4   pairWith: pairWith(monad),
 5   where:    where   (monad),
 6   select:   select  (monad),
 7   map:      select  (monad),
 8   inside:   inside  (monad),
 9   result:   () =>    monad
10 });
11
12 const from = jinq;
13
14 // ...
```

Listing 4.24: JINQ works on monads

Listing 4.25 uses the already familiar function `where` to showcase how the builder operations work:

```
1 // jinq.js
2 // ...
3
4 const where = monad => predicate => {
5   const processed =
6     monad.and(a => predicate(a) ? monad.pure(a) : monad.empty());
7   return jinq(processed);
8 };
9
10 // ...
```

Listing 4.25: Implementation of where

You may have already guessed it - the implementation of `where` is almost precisely the implementation of `keepEven`. It is more general because the predicate `x % 2 === 0` is outsourced to a function called `predicate`. See line 6 in listing 4.25: `and` keeps a value matching the predicate using `monad.pure` or discards it otherwise using `monad.empty`.
Line 7 then wraps the resulting monad in a new builder instance and returns it.

Another notable function of JINQ is `pairWith` - its implementation is shown in listing 4.26. Use it to combine a data source with another one (or even with itself):

```
// jinq.js
// ...

const pairWith = monad1 => monad2 => {
  const processed = monad1.and(x =>
    monad2.fmap(y => Pair(x)(y))
  );

  return jinq(processed)
};
// ...
```

Listing 4.26: Implementation of pairWith

`pairWith` takes a second monad. It now forms a new monad with a pair in it. But what happens here? Difficult to say because we can not know it. It just calls `and` on `monad1` and combines it with `monad2`. So when combining two sequences, every element of the first sequence gets paired up with every element of the second sequence. One could argue that this will take too much memory and could be more efficient. But let us step back and think about section 3.1.2: "Lazy Evaluating Iterables", which discusses the laziness of sequences. `monad1` (a sequence in the current example) is evaluated lazily. Therefore, never all combinations will be materialized in memory at once.

Creating Sequences using JINQ

A list comprehension in Haskell is an expression form that allows generating lists in a declarative way. See [21, Ch. 5] for an in-depth introduction to list comprehensions.

Listing 4.27 creates a list using list comprehension in Haskell:

```haskell
pairs = [(i,j) |         -- create a list of pairs where
          i <- [1..10], -- i can have the values 1 to 10
          j <- [1..4],  -- j can have the values 1 to 4
          i - j == 1]   -- only keep pairs with i - j == 1

print pairs
-- Prints '[(2,1),(3,2),(4,3),(5,4)]'
```

Listing 4.27: List comprehension in Haskell

Using JINQ, it is possible to create sequences similarly:

```javascript
const pairs =
  from(Range(1,10))                // create a seq with values 1 to 10
    .pairWith(Range(1, 4))         // union with a seq containing 1 to 4
    .where(([i, j]) => i - j === 1) // only keep pairs with i - j == 1
    .result();

console.log(pairs.fmap(show).toString());
// => Logs '[[2,1],[3,2],[4,3],[5,4]]'
```

Listing 4.28: Creating Sequences using JINQ

Of course, list comprehensions in Haskell provide more syntactical sugar than JINQ. Nevertheless, JINQ provides an easy way to create sequences based on rules and, thus, comes close to a list comprehension.

Using the JSONMonad to process lists of JavaScript objects

The `JSONMonad` gives arrays of JavaScript objects a monadic API. The `JSONMonad` makes them, therefore, compatible with JINQ. This is especially useful when JavaScript objects are created from JSON because they often have missing or nullable fields. The monadic operations of `JSONMonad` deal with that and ensure that querying nullable attributes do not cause problems.

Listing 4.29 dives right into an example defining two records that are related to each other. The first one contains data about an ancient battle, while the second one contains data about the heroes of the battle. The `heroId` connects these two records. The battle had a `winner` and a `loser`:

```javascript
const battleData = JSON.parse(`
  {
    "battleName": "The battle of Curly",
    "numberOfDeaths": 420000,
    "winner": { "teamName": "JSON", "outStandingHeroes": [1] },
    "loser": { "teamName": "XML", "outStandingHeroes": [] }
  }
`);

```

```
const heroes = JSON.parse(`
  [
    { "heroId": 1, "kills": 47076, "name": "Atonadias" },
    { "heroId": 2, "kills": 5691,  "name": "Tanobiri" },
    { "heroId": 3, "kills": 3707,  "name": "Tonadri" }
  ]
`);
```

Listing 4.29: Two data sources

Now imagine you want to find all names of the heroes of the winner team.
Listing 4.30 combines these two data sources using JINQ. Line 2 wraps the `battleData` with the `JSONMonad` to become queryable. `select` then accesses the property `winner` and from there, the property `outstandingHeroes`. Line 5 pairs the second data source before line 6, then destructures the pair and only keeps the heroes (from the second data source) that belong to the winning team (from the first data source).

```
 1 const outstandingHeroNames =
 2   from(JsonMonad(battleData))
 3     .select   (x => x["winner"])
 4     .select   (x => x["outStandingHeroes"])
 5     .pairWith (JsonMonad(heroes))
 6     .where    (([heroId, hero]) => heroId === hero["heroId"])
 7     .select   (([_, hero]) => hero["name"])
 8     .result   ();
 9
10 console.log(...outstandingHeroNames);
11 // => Los 'Atonadias'
```

Listing 4.30: Combining data sources using JINQ and JSONMonad

Conclusion

Monadic APIs allow the building of very general abstractions that can handle a wide variety of data types, even if they seem to have nothing in common at first glance - such as a sequence or a maybe. JINQ showcases an excellent instance of such a general abstraction - its operations only compose generic monadic functions. Nevertheless, JINQs operations have very expressive names describing their purpose, making it easy to use JINQ with different data types. That is truly beautiful and quite rare: general concepts with specific names.
JINQ shows its versatility through the possibility of creating new lists based on declarative rules and using the `JSONMonad`.

5. The Power of the Dot

In object-oriented languages, the receiver of an operation usually precedes the dot. As described in section 3.2: "Placing the Receiver at the End", this has some downsides. However, it also brings advantages: The IDE can help you find the correct method because the syntax "`receiver.`" allows the IDE to list all operations of the receiver's type. [9, Ch. "The Power of the Dot"]
This chapter shows how one can exploit the module system of JavaScript to get similar suggestions from the IDE, even when not working with objects.

5.1. The JavaScript Module System

Nowadays, when JavaScript programs are increasingly complex, it makes sense to structure the code better. Modules are a great way to do this - they define clear interfaces in which things are made available to other modules. For a JavaScript file to become a module, it must either

- export part of its functionalities using `export`
- or import functionality from other modules via `import`.

Listing 5.1 shows an example of a simple JavaScript module:

```
import { Sequence } from "../src/sequence/sequence.js";

export { endlessSeq }

const incrFn     = i => i + 1;
const whileFn    = _ => true;
const endlessSeq = Sequence(0, whileFn, incrFn);
```

Listing 5.1: A simple JavaScript module

Listing 5.1 imports the constructor `Sequence` and becomes a module. It exports `endlessSeq`, which then becomes available in other modules. `incrFn` and `whileFn` are not exported and, therefore, only available in this module.

5.1.1. IDE Support through modules

The module system offers several possibilities to bring autocompletion to the IDE using the dot.

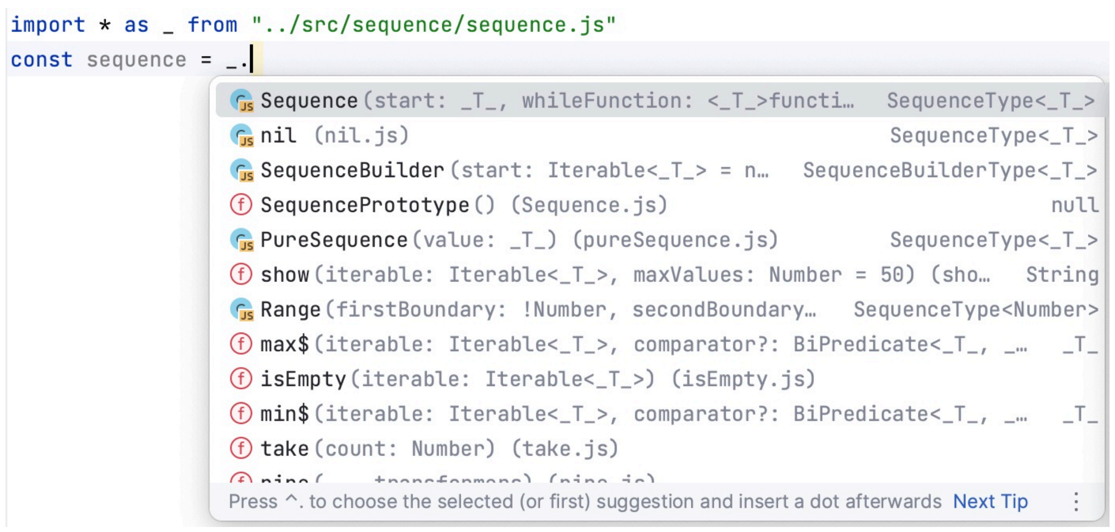
Named Imports

The Sequence library offers many loose functions. To use them, you need to know their names. This is not easy, especially for beginners, and reduces the development speed. Fortunately, this can be solved using named imports. Line 1 in listing 5.2 imports the Sequence library with the name "_":

```
1 import * as _ from "../src/sequence/sequence.js"
2
3 const seq = _.Sequence(0, i => i < 5, i => i + 1);
4
5 console.log(_.show(seq));
6 // => Logs '[0,1,2,3,4]'
```

Listing 5.2: Named imports in JavaScript

This allows the developer to access exported symbols of "_" using "_.". Thanks to the "." (dot) after the module name (_), IDEs like IntelliJ IDEA will suggest exported members of this module.
Figure 5.1 shows exactly this behaviour:



```
import * as _ from "../src/sequence/sequence.js"
const sequence = _.
```

```
Sequence(start: _T_, whileFunction: <_T_>functi…       SequenceType<_T_>
nil (nil.js)                                           SequenceType<_T_>
SequenceBuilder(start: Iterable<_T_> = n…         SequenceBuilderType<_T_>
SequencePrototype() (Sequence.js)                                   null
PureSequence(value: _T_) (pureSequence.js)             SequenceType<_T_>
show(iterable: Iterable<_T_>, maxValues: Number = 50) (sho…       String
Range(firstBoundary: !Number, secondBoundary…     SequenceType<Number>
max$(iterable: Iterable<_T_>, comparator?: BiPredicate<_T_, _…      _T_
isEmpty(iterable: Iterable<_T_>) (isEmpty.js)
min$(iterable: Iterable<_T_>, comparator?: BiPredicate<_T_, _…      _T_
take(count: Number) (take.js)
```

Press ^. to choose the selected (or first) suggestion and insert a dot afterwards Next Tip

Figure 5.1.: IntelliJ suggests exported members of a named import

This is a way to combine the power of the dot with the advantages of placing the receiver at the end. For users who know the library well, typing the additional name represents an unnecessary extra effort. The good thing about named imports is that every export supports them - the library developer does not need to adapt the exports.
This leaves it up to the user to import the library named. Named imports, therefore, allow the flexible customization of module names.

Additionally, object destructuring allows using often used symbols without module names as listing 5.3 shows:

```
1 import * as _ from "../src/sequence/sequence.js"
```

```
2 const { Sequence, show } = _;
3
4 const seq = Sequence(0, i => i < 5, i => i + 1);
5
6 console.log(show(seq));
7 // => Logs '[0,1,2,3,4]'
```

Listing 5.3: Destructuring of a named import

The statement on line 2 destructures the named module. Thus, `Sequence` is available as if a developer had imported it typically.

5.1.2. The Namespace Object Pattern

The previously described approach with named imports is great for functionalities that are only loosely related. What do you do if you want to define modules that share a common interface with other modules? Good examples of such a use case are services. Often you use dummy services in unit tests that should have the same interface as the production services that access an API. The Namespace Object pattern solves this problem.

Listing 5.4 defines a service with two methods. The object `service` gathers these two methods into a single namespace. This object is the only thing that the module exports.

```
export { service }

const getAll  = () => { /* <omitted> */ };
const getById = id => { /* <omitted> */ };

const service = { getAll, getById };
```

Listing 5.4: The Namespace Object pattern

A user of this service can now import it very easily, as listing 5.5 shows:

```
import { service } from "./power-of-the-dot.js";

const allElements = service.getAll();
```

Listing 5.5: Import a Namespace Object

Since this object can follow a defined interface (for example, using JSDoc), it can easily be replaced by just changing the import to another implementation.

Conclusion

The module system of JavaScript offers several possibilities to couple loose functions. While namespace objects can fulfil an interface, the flexibility of named imports makes it easier for developers to

get started with a potentially comprehensive library.
Both approaches allow the IDE to provide the developer with additional support by bringing "the power of the dot" to JavaScript modules.

6. Effective Testing

This chapter explains the testing strategy of the Sequence library. The start shows the instruments already available from the Kolibri Web UI Toolkit and how we used them. After that, we focus on the testing table - the core of the implemented testing framework during this project. The end of the chapter discusses some advanced concepts regarding property-based testing.

6.1. Motivation

When implementing a library, testing is an essential task. With the constant increase of functionalities, the number of tests also grows. Therefore, it brings many benefits to implementing tests in a standardized and generalized way. This leads to a robust and leakless testing framework and, thus, a solid code base. We called our implementation of this generalization "testing table". An explanation in detail follows in section 6.3: "Testing Table".
Such a testing strategy has significant advantages. The following non-concluding list shows an excerpt of the most important:

- Ensuring high code quality
- Avoid incompleteness - prevent missing test cases
- Prevent code duplication
- Less effort in writing tests
- Better overview of the code base
- Easier to understand test cases

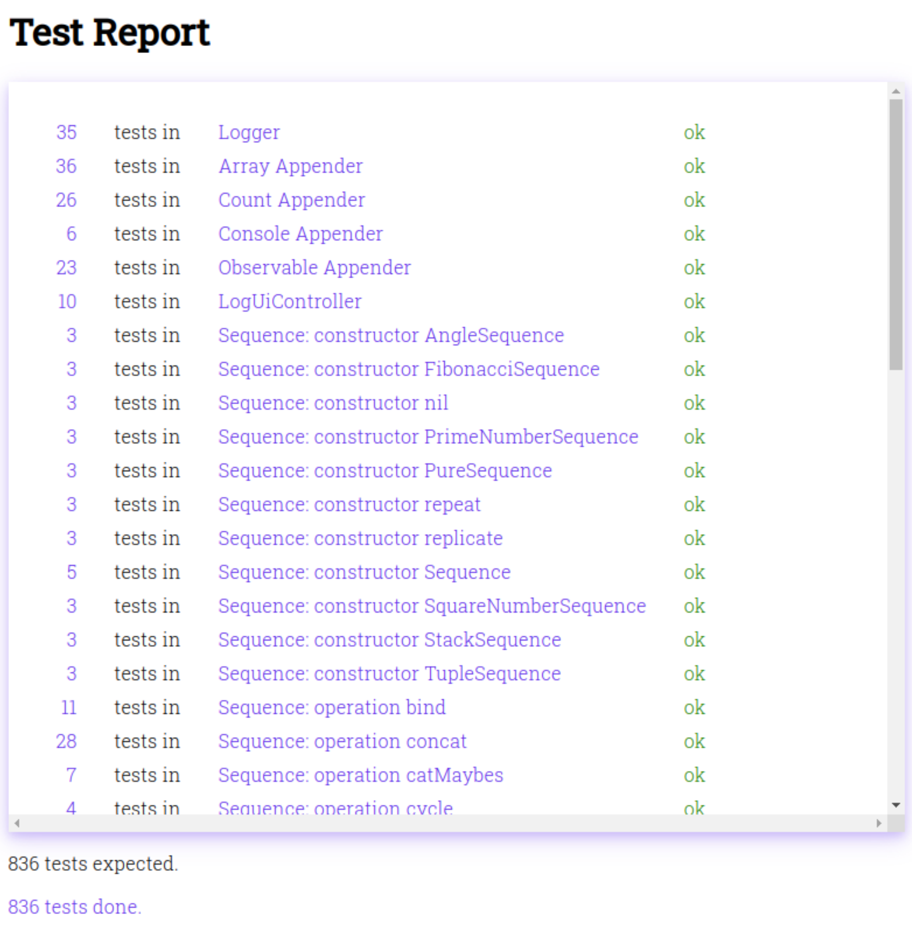
The generalization of testing is possible for functions that meet similar constraints. This enables to write configurations for such functions under test, which the testing framework then checks against certain predefined rules. Suppose a new particular case, probably a bug, is discovered during the development that also affects existing implementations. Then a new test covering this issue is added to the rules, which runs automatically against all testing configurations.

6.2. Kolibri Testing Framework

The Kolibri already provides a testing framework. We extended the framework with some iterable specific functionalities for easier testing of our implementations.

6.2.1. Parts of the Framework

The test suite is the core element of the testing framework. Typically, a test suite includes several tests, whereas a testing framework can consist of several suites. Executing the test suites generates a test report in HTML. An example of such a summary report shows figure 6.1:



Test Report

```
35  tests in  Logger                                   ok
36  tests in  Array Appender                           ok
26  tests in  Count Appender                           ok
 6  tests in  Console Appender                         ok
23  tests in  Observable Appender                      ok
10  tests in  LogUiController                          ok
 3  tests in  Sequence: constructor AngleSequence      ok
 3  tests in  Sequence: constructor FibonacciSequence  ok
 3  tests in  Sequence: constructor nil                ok
 3  tests in  Sequence: constructor PrimeNumberSequence ok
 3  tests in  Sequence: constructor PureSequence       ok
 3  tests in  Sequence: constructor repeat             ok
 3  tests in  Sequence: constructor replicate          ok
 5  tests in  Sequence: constructor Sequence           ok
 3  tests in  Sequence: constructor SquareNumberSequence ok
 3  tests in  Sequence: constructor StackSequence      ok
 3  tests in  Sequence: constructor TupleSequence      ok
11  tests in  Sequence: operation bind                 ok
28  tests in  Sequence: operation concat               ok
 7  tests in  Sequence: operation catMaybes            ok
 4  tests in  Sequence: operation cycle                ok
```

836 tests expected.

836 tests done.

Figure 6.1.: Example test report

To demonstrate how to work with the test suite, examine the following listing 6.1:

```
const suite = TestSuite("The truth");
suite.add("true is really true", assert => {
  assert.is(true, true);
});
suite.run();
```

Listing 6.1: TestSuite demonstration

Description of the functionalities:

- `TestSuite`: A `TestSuite` contains the test cases and is identified by a name. This name represents the suite on the summary page in figure 6.1 when executing.
- `run`: executes all tests containing the `TestSuite`.
- `add`: A function for including a test case into `TestSuite`. Its parameters are a string `name` and a callback which provides the test.

6.2.2. Additional Assertions

Assert functions in the Kolibri testing framework execute tests by comparing two values and store the result in a summary collection of the test suite. Listing 6.1 shows the comparison of two booleans using the `assert.is(...)` function. This function compares an `actual` and an `expected` value. The references of these two values must match. Otherwise, the test fails.

Assertion

Comparing iterables is slightly more complicated. For this reason, we extended the testing framework with an additional assert function that compares two arbitrary iterables. Listing 6.2 shows the usage of this function called `iterableEq`:

```js
testSuite.add("demo iterableEq", assert => {
  const sequence = Sequence(0, x => x < 2, x => x +1);

  assert.iterableEq(sequence, [0,1]);
  assert.iterableEq(sequence, Pair(0)(1));
  assert.iterableEq(sequence, Range(1));
});
```

Listing 6.2: Usage of iterableEq

The implementation of `iterableEq` is suitable for any two iterables. Listing 6.3 shows the implementation of `iterableEq`:

```js
// test.js
...
iterableEq: (actual, expected, maxElementsToConsume = 1_000) => {

    if (actual[Symbol.iterator]   === undefined) error("...");
    if (expected[Symbol.iterator] === undefined) error("...");

    const actualIt     = actual[Symbol.iterator]();
    const expectedIt   = expected[Symbol.iterator]();

    let iterationCount = 0;
    let testPassed     = true;
    let message        = "";

    while (true) {
```

```
16          const { value: actualValue,   done: actualDone   } = actualIt.next();
17          const { value: expectedValue, done: expectedDone } = expectedIt.next();
18
19          const oneIteratorDone      = actualDone || expectedDone;
20          const bothIteratorDone     = actualDone && expectedDone;
21          const tooManyIterations    = iterationCount > maxElementsToConsume;
22
23          if (bothIteratorDone) break;
24          if (oneIteratorDone) {
25              testPassed = false;
26              message    = '...'
27              break;
28          }
29          if (tooManyIterations) {
30              message    = '...'
31              testPassed = false;
32              break;
33          }
34          if (actualValue !== expectedValue) {
35              testPassed = false;
36              message    = '...'
37              break;
38          }
39
40          iterationCount++;
41      }
42      if (!testPassed) error(message);
43      results.push(testPassed);
44      messages.push(message);
45  }
```

Listing 6.3: Implementation of iterableEq

Since an iterable can be endless, `iterableEq` defines a default maximum iteration amount on line 3 (`maxElementsToConsume`). This is handy because it is often the case that an iterable runs infinitely when something goes wrong. If an iterable exceeds this limit, the test fails.
The implementation checks on line 5 if both values are iterable. If this is the case, it compares the iterables using a `while` loop: Lines 23 to 38 check the equality for each iteration - if one iterator has fewer elements, it took too many iterations or the values at the same position are not equal, the test fails.
On line 43, the test stores the result in the summary collection of the corresponding test suite.

Assertion for Exceptions

Another extension for the testing framework is the `assert.throws` function. It is often the case that functions must throw an exception in a specific situation. For example, when looking for the maximum in an empty iterable. `max` throws an exception in this case. Therefore, testing the function's behaviour in error cases is also required. For this purpose, listing 6.4 shows the implementation of `assert.throws`:

```
1  // test.js
2  ...
3  throws: (functionUnderTest, expectedErrorMsg = "") => {
4       let testResult    = false;
5       let message       = "";
6       const hasErrorMsg = expectedErrorMsg !== "";
7
8       try {
9           functionUnderTest();
10          message = "Did not throw an error!";
11          if (hasErrorMsg) {
12              message += ` Expected: '${expectedErrorMsg}'`;
13          }
14          error(message);
15      } catch (e) {
16          testResult = true;
17
18          if (hasErrorMsg) {
19              testResult = expectedErrorMsg === e.message;
20          }
21      }
22      results .push(testResult);
23      messages.push(message);
24  }
25  ...
```

Listing 6.4: Implementation of assert.throws

Line 9 calls the function under test. As expected, this function should throw an exception that is caught in the catch block. If this is the case, the test is successful. Otherwise, the test fails and a corresponding error message is stored.

6.3. Testing Table

The testing table enables generalizing test cases for operations of the Sequence library. Each function under test must provide a configuration for the testing framework.

The architecture consists of three main parts:

1. a table containing all testing functions
2. configuration objects to define test properties
3. a function `addToTestingTable` including all tests from the testing table into a given test suite

Each constructor and operator of the Sequence library has its configuration that specifies the test behaviour. Likewise, each has its `TestSuite`.

6.3.1. Configuring the Testing Table

In order to test operators and constructors using the testing table, it is necessary to implement a test configuration. Listing 6.5 shows the testing file of the function `reduce`. Lines 3 to 12 define the test configuration. The testing table does not handle special cases. Those follow later in the same file from line 14 on:

```
 1 const testSuite = TestSuite("Sequence: terminal operation reduce$");
 2 
 3 addToTestingTable(testSuite)(
 4   createTestConfig({
 5     name:      "reduce$",
 6     iterable:  () => newSequence(UPPER_SEQUENCE_BOUNDARY),
 7     operation: () => reduce$((acc, cur) => acc + cur, 0),
 8     expected:  10,
 9     evalFn:    expected => actual => expected === actual,
10     excludedTests: [TESTS.TEST_CB_NOT_CALLED_AFTER_DONE]
11   })
12 );
13 
14 testSuite.add("test: special case", assert => {
15   // Given
16   // When
17   // Then
18 });
19 
20 testSuite.run();
```

Listing 6.5: Test configuration reduce

On line 4, the function `createTestConfig` sets default values of configuration properties to simplifying the code.

The table 6.1 shows the available configuration properties and their purpose:

Property	**Description**	**Required**
`name`	The name of the test for meaningful reporting messages.	y
`iterable`	A function that constructs a new iterable to apply the operation to.	y
`expected`	The expected result of the operation applied to the iterable defined in property `iterable`.	y
`excludedTests`	An optional array of testing functions to exclude tests in the testing table.	n
`operation`	The operation to test. `param` is passed as an argument to it (leave this empty for constructor tests since they do not take an inner iterator).	n
`param`	A parameter passed to this `operation`. If it is a function, some extra tests will be performed.	n
`invariants`	An optional array of `InvariantCallback`. The invariant must hold tests against different lists.	n
`evalFn`	An optional function that compares the `expected` and the `actual` values. The default is `iterableEq`.	n

Table 6.1.: Properties of the configuration object

6.3.2. The Table

The testing table is just an array (table) containing test objects. Listing 6.6 shows the implementation:

```
const testingTable = [
  { name: TESTS.TEST_SIMPLE,                   test: testSimple},
  { name: TESTS.TEST_PURITY,                   test: testPurity},
  { name: TESTS.TEST_CB_NOT_CALLED_AFTER_DONE, test: testCBNotCalledAfterDone},
  { name: TESTS.TEST_PROTOTYPE,                test: testPrototype},
  { name: TESTS.TEST_INVARIANTS,               test: testInvariants},
  { name: TESTS.TEST_ITERATE_MULTIPLE_TIMES,   test: testIterateMultipleTimes},
];
```

Listing 6.6: Testing table

An object in the table includes two properties:

- A string representing its name. The name is important for displaying an accurate error message if the test fails
- A function to test a specific behaviour

Each function of the testing table expects as argument a test configuration object of type `SequenceTestConfigType`. Listing 6.7 presents exemplary the implementation of `testSimple`:

```
 1 /**
 2  * @type {
 3  *             (config: SequenceTestConfigType)
 4  *          => (assert: AssertType)
 5  *          => void
 6  *       }
 7  */
 8 const testSimple = config => assert => {
 9   const { iterable, operation, evalFn, expected, param } = config;
10   const baseIterable = iterable();
11   const operated     = operation(param)(baseIterable);
12   evaluate(expected, operated, assert, evalFn);
13 };
```

Listing 6.7: Implementation test simple

The function `simpleTest` examines whether an operation correctly processes a typical use case. For this purpose, it executes the following tasks:

- Line 10 shows the invocation of the function `iterable` to create an iterable for further use.
- Line 11 executes the operation to test on the previously obtained iterable by passing the required parameters. `param` is optional and can, therefore, also be left empty. This covers cases where operations need to be parametrized (like `map`, which takes a mapping function).
- Line 12 calls a function `evaluate`, which then calls the passed function `evalFn`, comparing the `actual` and `expected` values. This gives the possibility to evaluate iterables or sequences containing more complex values or if the output of an operation is not an iterable anymore (as, for example, with `reduce`). If `evalFn` is undefined, the fallback function `iterableEq` is used, comparing two iterables which section 6.2.2: "Additional Assertions" explains.

The testing table includes test-functions to examine the following behaviours of implementations of the Sequence library:

- **testSimple** checks if a typical case works properly.
- **testPurity** checks if an operator makes some side effects.
- **testCBNotCalledAfterDone** checks if a given callback of an operator will be called when the iterable is exhausted.
- **testPrototype** checks if the `SequencePrototype` is set.
- **testInvariants** checks if an invariant holds by applying different parameters.
- **testIterateMultipleTimes** checks if an iterable produces the same output twice.

6.3.3. Running the Testing Table

`addToTestingTable` must be called for each configuration to be tested by the testing table. It runs each test with the given configuration (line 16). Some tests do not make sense for all functions of the Sequence library - since `addToTestingTable` runs all tests by default, it must provide an option to exclude such tests. For example, if an operation has no callback, the check of its side effect is useless.

Line 14 filters out tests excluded by the configuration:

```
1  export { addToTestingTable }
2  ...
3  /**
4   * @type {
5   *       (testSuite: TestSuiteType)
6   *    => (config: SequenceTestConfigType)
7   *    => void
8   * }
9   */
10 const addToTestingTable = testSuite => config => {
11   const { excludedTests } = config;
12 
13   testingFunctions
14     .filter (({ name })        => !excludedTests.includes(name))
15     .forEach(({ name, test })  =>
16          testSuite.add(`${name}: ${config.name}`, test(config)));
17 };
```

Listing 6.8: Implementation of addToTestingTable

6.4. Testing based on Properties

John Huges and Carl Claessen wrote the following sentence in the paper "Quickcheck, A Lighweight Tool for Random Testing of Haskell Programms": *Despite anecdotal evidence that functional programs require somewhat less testing ('Once it type-checks, it usually works'), in practice it is still a major part of functional program development* [10]. If these programming icons claim it takes many tests even in a strongly typed language, how many does it take in JavaScript? Of course, enough.

6.4.1. Invariant Testing

Invariant tests add another layer of verification to ensure the correctness of the Sequence library. Some aspects of this testing are leaned on the Quickcheck approach of the paper mentioned before. We concentrated on only some essential parts since implementing the whole Quickcheck framework would compared to the effort not bring enough benefits for this work.
The test configuration object contains a property `invariant`. It allows defining of some invariants, which are tested against different iterables. Quickcheck does this with randomly generated data. Our data is fixed and includes a mix of special cases.
Line 10 in listing 6.9 defines the invariants of `reverse`. This statement defines that an iterable must be equal to the original when reversed two times:

```
1 const testSuite = TestSuite("Sequence: operation reverse$");
2 
3 addToTestingTable(testSuite)(
4   createTestConfig({
    name:      "reverse$",
    iterable:  () => newSequence(UPPER_SEQUENCE_BOUNDARY),
    operation: () => reverse$,
    expected:  [4, 3, 2, 1, 0],
    invariants: [
      it => reverse(reverse(it)) ["=="] (it),
    ]
  })
);

testSuite.run();
```

Listing 6.9: Test configuration reverse

The testing table contains a test function `testInvariants`. Listing 6.10 defines that function:

```
/**
 * Applies a series of lists to a given invariant.
 * @template _T_
 * @type {
 *             (invariants: InvariantCallback)
 *          => (assert: AssertType)
 *          => void
 *        }
 */
const invariantPenetration = invariant => assert => {
  const testingLists = [
    // edge case
    nil,
    // edge case, done calculated
    newSequence(1),
    // typical number
    newSequence(3),

    // no big iterable, needs extra test

    // edge case, done set explicitly
    PureSequence("testString"),
    // mixing types
    ['a', 'b', 'c', 1, 2, 3, Nothing, Just("testString")],
    // iterable of iterables
    [PureSequence(1), newSequence(4), '#', "abc", 1]
  ];

  for (const list of testingLists) {
    const result = invariant(list);
    assert.isTrue(result);
  }
};
```

Listing 6.10: Implementation invariant penetration

Line 11 defines a list with several parameters of type `Iterable`. Each parameter tests if the invariant of `reverse` holds. This enables testing many cases written in a few lines of code.

An advanced Example

Listing 6.11 shows the definition of the identity law for the function <> in Haskell which concatenates two lists:

```haskell
-- left identity
x <> mempty = x
-- right identity
mempty <> x = x
```

Listing 6.11: Left and right identity of <> in Haskell

Listing 6.11 demonstrates that a list *x* concatenated with `nil`, the empty list, equals the original one. Besides <> there is also `mconcat`. `mconcat` allows flattening a list consisting of further lists to a single list. This operation is often used and, therefore, also exists in the Sequence library. As <>, `mconcat` is also associative and a concatenation with the neutral element (the empty list) has no influence on the result.
With the previously shown implementation of the invariant based testing, it becomes possible to test these laws for `mconcat`. Listing 6.12 shows the implementation of these invariants in JavaScript:

```javascript
1  addToTestingTable(testSuite)(
2    createTestConfig({
3      name:       "mconcat",
4      iterable:   () =>
5        toMonadicIterable([ newSequence(2), newSequence(2), newSequence(2) ]),
6      operation:  () => mconcat,
7      expected:   [0, 1, 2, 0, 1, 2, 0, 1, 2],
8      invariants: [
9        it => mconcat([nil, it]) ["=="] (it),
10       it => mconcat([it, nil]) ["=="] (it),
11       // ...
12     ],
13   })
14 )
```

Listing 6.12: Invariants of mconcat

Let us focus on lines 8-12. The property `invariants` expect an array of functions. Each of these functions represents a law. Therefore, the function injects an arbitrary iterable called `it`, and the law must hold for it.

6.4.2. Revealed Code Bugs: Discoveries through Testing during Development

As illustrated in the testing table section, a test case named `TEST_CB_NOT_CALLED_AFTER_DONE` guarantees that a callback is not invoked once an iterator is exhausted. This test arose from a bug during development that it is expected for a callback not to be called after the iterator is used up. Such occurrences could lead to program errors or unexpected behaviour. Consequently, we implemented a test case addressing this scenario and added it to the testing table. By adding just one more test, the entire Sequence library could be examined for this behaviour, allowing for further improvements and finding the same bug in other functions.
Another example concerns the behaviour of operators and operations when dealing with empty iterables. When a test configuration provides an invariant, it is automatically tested against the empty sequence. This is a powerful way to ensure that invariants apply to all types of sequences.
Furthermore, another special test case is the `TEST_PURITY`, which ensures that the state of an iterable is in the proper location. This safeguards against unexpected side effects, promoting a more reliable program execution.

6.4.3. Conclusion

When implementing large code bases, having a solid test framework is crucial. Structured extension of the test suite during development leads to extensive core functionality testing. Thus, the core of the library becomes even more robust when the code base grows.
The standardization and generalization of tests using a testing table imply that writing tests for new functionalities takes less time and ensures better quality.

7. API

Reader Guidance

This chapter gives an overview of all features provided by the functional standard library. The provided code examples illustrate simple cases. Therefore, the reader will only partially understand the concepts and background of the work when reading this chapter. The previous chapters provide a much deeper conceptual understanding. The chapter starts with an overview of this library and explains how to import it in your project. After that, it shows the whole API of the Sequence library and JINQ and explains the additions to the Kolibri standard library.

7.1. Introduction

7.1.1. Library Overview

This library consists of several parts. Table 7.1 gives a brief overview of each of those. The following sections discuss these different aspects.

Part	**Description**
Sequence library	The Sequence library provides operations for processing iterables. Additionally, there are constructors to create sequences.
JINQ	JINQ is a uniform query language to query different data structures.
Stdlib	Stdlib extends the Kolibri standard library with additional features.
FocusRing	The FocusRing is an immutable data structure, whose elements are arranged in a ring.

Table 7.1.: The available operators in the Sequence library

Note: The FocusRing results from the predecessor project and is only listed here for completeness. The same applies to the range, which is part of the Sequence library. Please find more information in the documentation of the previous work. [24].

7.1.2. Including the Library in a Project

This library is part of the Kolibri project [4] and uses many of its functionalities. The Kolibri ships this library, is available on GitHub and can be included directly in existing projects via the ES6 module system.

7.2. Sequence Library

This section overviews which operations exist to create and process sequences and how to import this library into your own project.

7.2.1. Getting Started

Importing the Sequence Library

The module `sequence/sequence.js` exports the whole Sequence library. The easiest way to use it is through named imports. As listing 7.1 shows, this approach allows using the dot notation, which enables the IDE to support the developer with auto-completion. Developers who have been using the library longer can import individual functions directly.

```
import * as _ from "./sequence/sequence.js"

const seq = _.Sequence(0, i => i < 5, i => i + 1);

console.log(_.show(seq));
// => Logs '[0, 1, 2, 3, 4]'
```

Listing 7.1: Importing the Sequence library using named imports

Note: Of course, importing the Sequence library with any name is possible. The advantage of _ is that it is short and takes up very little white space. Other special characters, such as the $, negatively affect the code's readability.

Evaluating endless Sequences

Section 3.1.2: "Lazy Evaluating Iterables" describes the benefits of lazy evaluation, which allows working with sequences consisting of infinite elements. When working with such sequences, pay attention to only evaluating parts of the sequence because evaluating endless values would take forever. Functions of the Sequence library, which will process the whole sequence, are therefore marked with a $ symbol in the name. These functions are only applicable to sequences with a finite number of elements.
Listing 7.2 shows an example of that:

```
import * as _ from "./sequence/sequence.js"

const seq      = _.Sequence(0, _ => true, x => x + 1);
const part     = _.take(5)(seq);

const reversed = _.reverse$(part);

console.log(...reversed);
// => Logs '4, 3, 2, 1, 0'
```

Listing 7.2: Evaluating a part of an endless Sequence

Line 3 creates a sequence `seq` which contains infinite elements. Evaluating this would go on forever. Therefore, line 4 uses the function `_.take(5)` which only evaluates 5 elements.

7.2.2. Constructors

A constructor is anything that creates a sequence without depending on another one. So they serve as an entry point to the Sequence library.
Table 7.2 gives an overview of all available constructors. Code examples and more information about the constructors delivers the appendix B.1: "Constructors".

Name	**Description**
`nil`	This constant represents a sequence containing no values.
`PureSequence`	Creates a sequence which contains just the given value.
`Range`	Generates a sequence of numbers. [24]
`repeat`	`repeat(arg)` creates an infinite sequence that will repeatedly yield the value of `arg` when iterated over.
`replicate`	`replicate(n)(x)` creates a sequence of length `n` with `x` the value of every element.
`Sequence`	Creates a new sequence based on a start `value`, `whileFunction` and `incrementFunction`.
`StackSequence`	Creates a `SequenceType` on top of the given `stack`.
`TupleSequence`	Constructs a new `SequenceType` based on the given tuple.

Table 7.2.: The available constructors in the Sequence library

7.2.3. Operators

Operators are functions that operate on any existing iterable and create a new sequence from it.
Table 7.3 gives an overview of all available operators.
Code examples, types and more information about the operators delivers the appendix B.2: "Operators".

Name	**Description**
`bind`	Applies the given function to each element of the iterable and flattens it afterward. Note: this operation adds a monadic API to the `SequenceType`.
`catMaybes`	The catMaybes function takes an iterable of maybes and returns a sequence of all the Just's values.
`concat`	Adds the second iterable to the first iterables end.
`cons`	Adds the given element to the front of an iterable.
`cycle`	Ties a finite iterable into a circular one, or equivalently, the infinite repetition of the original iterable.
`drop`	Jumps over so many elements.
`dropWhile`	Discards all elements until the first element does not satisfy the predicate any more.
`map`	Transforms each element using the given function.
`mconcat`	Flatten an iterable of iterables.
`pipe`	Transforms the given iterable using the passed operators.
`rejectAll`	Only keeps elements which do not fulfil the given predicate.
`retainAll`	Only keeps elements which fulfil the given predicate.
`reverse$`	Processes the iterable backwards.
`snoc`	Adds the given element to the end of the iterable. It is the opposite of `cons`.
`take`	Stop after so many elements.
`takeWhile`	Proceeds with the iteration until the predicate becomes true.
`zip`	Zip takes two iterables and returns an iterable of corresponding pairs.
`zipWith`	Generalises `zip` by zipping with the function given as the first argument.

Table 7.3.: The available operators in the Sequence library

Using the pipe Operator

The operator `pipe`, is unique because it provides no new functionality but is pure syntactic sugar. It offers the possibility of combining several operators. Lines 5 to 11 of listing 7.3 show nesting of multiple operators. Readability suffers very much from this. In comparison, it is easier to read the code lines 16 to 21, combining multiple operators using `pipe`.

```
1 import * as _ from "./src/sequence/sequence.js"
2
3 const numbers = _.Sequence(0, _ => true, x => x + 1);
4
5 const mySequence = _.take(5)(
6   _.retainAll(x => x > 50)(
    _.map(x => 2 * x)(
      _.rejectAll(x => x % 2 === 1)(numbers)
    )
  )
);

console.log(...mySequence);
// => Logs '52, 56, 60, 64, 68'

const mySequence2 = _.pipe(
  _.rejectAll(x => x % 2 === 1),
  _.map(x => 2 * x),
  _.retainAll(x => x > 50),
  _.take(5)
)(numbers);

console.log(...mySequence2);
// => Logs '52, 56, 60, 64, 68'
```

Listing 7.3: pipe combines multiple operators

As listing 7.4 shows, using the version of `pipe` defined on the prototype of the sequence, the readability improves even more:

```
const mySequence3 = seq.pipe(
  _.rejectAll(x => x % 2 === 1),
  _.map(x => 2 * x),
  _.retainAll(x => x > 50),
  _.take(5)
);
console.log(...mySequence3);
// => Logs '52, 56, 60, 64, 68'
```

Listing 7.4: Using pipe defined on the prototype

7.2.4. Terminal Operations

Terminal operations are all functions that operate on an existing sequence and do not necessarily create a new sequence. In other words, terminal operations evaluate a sequence.
Table 7.4 gives an overview of all available terminal operations.
Code examples and more information about the terminal operations delivers the appendix B.3: "Terminal operations".

Name	Description
`eq$`	Checks the equality of two finite iterables.
`foldr`	Performs a reduction on the elements from right to left, using the provided start value and an accumulation function, and returns the reduced value.
`forEach$`	Executes the callback for each element.
`head`	Return the next value without consuming it. `undefined` when there is no value.
`isEmpty`	Returns `true`, if the iterables head is undefined.
`max$`	Returns the largest element of a **non-empty** iterable.
`safeMax$`	Returns the largest element of an iterable.
`min$`	Returns the smallest element of a **non-empty** iterable.
`safeMin$`	Returns the smallest element of an iterable.
`foldl$`	Performs a reduction on the elements, using the provided start value and an accumulation function, and returns the reduced value.
`show`	Transforms the passed iterable to a string.
`uncons`	Removes the first element of this iterable.

Table 7.4.: The available constructors of the Sequence library

7.2.5. Sequence Prototype

The prototype of the sequence provides some operations as well. This improves code readability in some situations. The monadic operations are also available on the prototype of the sequence. Thus, it is compatible with all functions that expect a monad as a parameter.

Verify the Prototype

Listing 7.5 shows how to query the prototype of any object using the function `Object.getPrototypeOf`. Applying this to a sequence will return `SequencePrototype`:

```
import { Sequence, SequencePrototype } from "sequence/sequence.js";

const seq          = Sequence(0, _ => true, i => i + 1);
const seqPrototype = Object.getPrototypeOf(seq);

console.log(seqPrototype === SequencePrototype);
// => Logs 'true'
```

Listing 7.5: The prototype of a sequence

The Operations on the Prototype

Name	Description
`fmap`	the same as `map` described in section 7.2.3: "Operators"
`and`	the same as `bind` described in section 7.2.3: "Operators"
`pure`	the same as `PureSequence` described in section 7.2.2: "Constructors"
`empty`	the same as `nil` described in section 7.2.2: "Constructors"
`["=="]`	the same as `eq$` described in section 7.2.4: "Terminal Operations"
`toString`	the same as `show` described in section 7.2.4: "Terminal Operations"
`pipe`	the same as `pipe` described in section 7.2.3: "Operators"

Table 7.5.: The operations served on the prototype of the sequence

When to use the Operations on the Prototype?

Each operation on the prototype is also available as an operator or terminal operation. The prototype offers these functionalities additionally because sometimes it has additional advantages - the function `toString`, for example, converts by convention an object into its string representation. Other operations, such as `["=="]`, often make the code more readable. So when to use the prototype operations? It depends on the context and code readability.

7.3. Extension of the Kolibri Standard Library

From this project's findings, some existing functionalities of the Kolibri could also be improved or extended. This section describes those extensions.

7.3.1. Extensions to Maybe

The data structure maybe introduced in section 4.1: "Wrapping Values in a Context", was already part of the Kolibri before this project work. Now its prototype of maybe also supports monadic operations. Listing 7.6 shows these monadic operations of maybe in action:

```
import { Nothing, Just } from "./src/stdlib/maybe.js";

/** Prints the value of a Maybe if it exists */
const evalMaybe = maybe =>
  maybe
    (_ => console.log("There was no value!"))
    (x => console.log(x));

```

```
const just    = Just(5);
const nothing = Nothing;

// fmap
const justMapped    = just   .fmap(x => 2*x); // results in 10
const nothingMapped = nothing.fmap(x => 2*x); // nothing happens!
evalMaybe(justMapped);                        // => Logs '10'
evalMaybe(nothingMapped);                     // => Logs 'There was no value!'

// and
const justAnd    = just   .and(x => nothing); // Turns this value into Nothing
const nothingAnd = nothing.and(x => just);    // Can't change Nothing
evalMaybe(justAnd);                           // => Logs 'There was no value!'
evalMaybe(nothingAnd);                        // => Logs 'There was no value!'

// pure
evalMaybe(just.pure(2));                      // => Logs '2', Same as Just(2)
evalMaybe(nothing.pure(2));                   // => Logs '2', Same as Just(2)

// empty
evalMaybe(just.empty());    // => Logs 'There was no value!', Same as Nothing
evalMaybe(nothing.empty()); // => Logs 'There was no value!', Same as Nothing
```

Listing 7.6: The monadic operations of Maybe in action

7.3.2. Extensions to Pair

Section 2.4: "Iterables Everywhere" describes the pair as an immutable data structure. To access its values simpler, it is now iterable. Listing 7.7 shows these advantages:

```
import { Pair } from "./src/stdlib/pair.js"

const pair = Pair(1)(2);
console.log(...pair);
// => Logs '1, 2'
```

Listing 7.7: Iterable pair

Since the pair is iterable, it is also compatible with all operations defined for sequences:

```
import { Pair } from "./src/stdlib/pair.js"
import { map }  from "./src/sequence/sequence.js"

const pair = Pair(1)(2);
console.log(...map(x => 2*x)(pair));
// => Logs '2, 4'
```

Listing 7.8: Transforming a pair using map

7.4. Sequence Builder

A mutable builder to create sequences.
`SequenceBuilder` allows the creation of a sequence by generating elements individually and adding them to the `SequenceBuilder` (without the call stack overhead when doing so with `cons`).

Note: It is strongly recommended to use `SequenceBuilder` to create Sequences with more than 1000 elements when adding them individually.

7.4.1. Functions

`SequenceBuilder` provides the following functions:

Name	**Description**
`prepend`	adds one or more elements or a sequence to the beginning of the sequence
`append`	adds one or more elements or a sequence to the end of the sequence
`build`	builds a sequence containing the previously passed elements

Table 7.6.: SequenceBuilder functions

7.4.2. Examples

Listing 7.9 creates a sequence from different values and iterables:

```
import * as _               from "./src/sequence/sequence.js"
import { SequenceBuilder } from "./src/sequence/SequenceBuilder.js";

const range = _.Range(1, 3);

const result = SequenceBuilder()
.append(range)
.append(4)
.prepend(0)
.append(5,6,7)
.build();

console.log(_.show(result));
// => Logs '[0,1,2,3,4,5,6,7]'
```

Listing 7.9: SequenceBuilder example

7.5. JINQ

JINQ (JavaScript language integrated queries) enables you to query any monadic type using the same syntax.
For further details and explanations, consider section 4.4.3: "Introducing JINQ".
To create a monadic type, visit chapter 4: "Monads in JavaScript".

7.5.1. Functions

JINQ provides the following functions:

Name	**Description**
`from`	serves as starting point to enter JINQ and specifies a data source
`map`	maps the current value to a new value
`select`	alias for `map` (same functionality)
`where`	only keeps the items that fulfil the predicate
`inside`	maps the current value to a new `MonadType`
`pairWith`	combines the underlying data structure with the given data structure as `PairType`
`result`	returns the result of this query

Table 7.7.: JINQ functions

7.5.2. Examples

Even Numbers

Generating a sequence of even numbers using JINQ and a range as data source:

Note: Since `result` returns a `MonadType`, casting is needed to pass it to `show`.

```
import * as _    from "./src/sequence/sequence.js"
import { from }  from "./src/jinq/jinq.js";
import { Range } from "./src/range/range.js";

const range = Range(10)
const result =
    from(range)
      .where(x => x % 2 === 0)
      .result();

console.log(_.show(/** @type SequenceType */ result));
```

```
// => Logs '[0,2,4,6,8,10]
```

Listing 7.10: Even numbers generated by JINQ

7.6. How to Extend the Library

Since applications are versatile, you can add further functionality to the library. This section shows how to extend the Sequence library with own functionality.

7.6.1. Adding a new Operator

In this scenario, we include `concat` to the Sequence library. `concat` appends one iterable to another.

Kind of the Operation

The Sequence library distinguishes between operators and terminal operations. The difference depends on the return type. If a function takes a sequence and returns a sequence, it is an operator.
It is a terminal operation if it takes a sequence and produces a different type than a sequence, such as `isEmpty`.
Note: It doesn't matter if you want to add a constructor, an operator, or a terminal operation to the sequence library - the procedure is always the same. The following example can, therefore, also be applied to constructors and terminal operations.

Directory Structure

Each operation of the Sequence library consists of two files: a test file and an implementation file. Therefore, we create a `concat.js` and a `concatTest.js` file in a new folder `concat` in the existing folder `operators`. This helps keep overview and finding desired files faster. Figure 7.1 shows the relevant directories:

```
src
├── sequence
│   ├── operators
│   │   ├── concat
│   │   │   ├── concat.js
│   │   │   └── concatTest.js
│   │   ├── ...
│   │   ├── operators.js
│   │   └── operatorsTest.js
│   └── ...
├── AllTests.html
└── ...
```

Figure 7.1.: concat directory structure

Exports and Imports

All artefacts of the Sequence library are available via the `sequence.js` file. To make `concat` support this, we export it via the `operators.js` file. Listings 7.11 and 7.12 show the corresponding statements:

```
// concat.js

export { concat }
```

Listing 7.11: Export of concat

```
// operators.js
...
export * from "./concat/concat.js"
```

Listing 7.12: Export of concat in operators.js

The same principle applies to the test files. Importing the `concatTest.js` in `operatorsTest.js` enables to run the test cases.

```
// operatorsTest.js
...
import "./concat/concat.js"
```

Listing 7.13: Export of concat in operatorTest.js

7.6.2. Implementing a new Operator

Writing the Tests

The first user of a new operator of the Sequence library is usually a test. Therefore, we start by writing the tests.

Note: Chapter 6: "Effective Testing" explains testing of the Sequence library in detail. This section therefore does not cover how this exactly works.
Let us start with the imports. A test needs the following dependencies:

- TestSuite
- addToTestingTable
- createTestConfig

Now you are ready to create a test configuration and, optionally, some additional tests to assure the correctness of special cases. Creating a test configuration requires the following steps:

- Create a `TestSuite` with a meaningful name corresponding to the function you are implementing.
- `createTestConfig` expects an object of type `SequenceTestConfigType`. Have a look to this type or to section 6.3.1: "Configuring the Testing Table" to get the information about the available properties.
- If necessary, add some further test cases using the same test suite.
- Run the `TestSuite` by calling the `run()` function at the end of the file.

Listing 7.14 shows a scaffold of a test file.

```
import { addToTestingTable }  from "./src/sequence/util/testingTable.js";
import { TestSuite }          from "./src/sequence/test/test.js";
import { createTestConfig }   from "./src/sequence/util/testUtil.js";
import { concat }             from "./src/sequence/sequence.js";
...

const testSuite = TestSuite("Name of the TestSuite");

addToTestingTable(testSuite)(
  createTestConfig({
      ...
    })
);

testSuite.add("test special case", assert => {
  // Given
  ...
  // When
  ...
  // Then
  ...
});

testSuite.run();
```

Listing 7.14: Imports of concatTest.js

Run the `AllTests.html` file, and you should see that your tests are failing. If so, then everything is fine.

Note: While running the tests, always observe the console.

Writing the Functionality

Now you are in a comfortable position to implement your code against tests. For this, create a function `concat` in the file `concat.js`. Probably, you can use existing functionalities of the Sequence library for your implementation. Sections 7.2.2: "Constructors" - 7.2.4: "Terminal Operations" provide a list of all provided functionality of the Sequence library.
Again, run the tests and be proud if everything is working.

8. Results

This chapter contains the results of the thesis. First, it discusses the outcomes of the user test. Then, applications demonstrate the usage and usefulness of the Sequence library. The following section lists features that have not yet been implemented, before the final section reviews the results obtained in this thesis.

8.1. User Test Evaluation

8.1.1. Introduction

We conducted user tests with various developers to get feedback on the library. In total, 13 people participated in this test. All those received a link to a GitHub repository containing the description of two tasks:

1. **Numerical differentiation**: The idea is to use sequences to get an approximation as close as possible to the slope of a function f at point x. Laziness and operations on sequences make this possible.
2. **Browse JSON files**: In JavaScript, you frequently work with JSON files provided by some API. This data is often full of holes, which makes it difficult to process. JINQ simplifies this task significantly because it fills such gaps automatically.

Both tasks are pleasant use cases of the artefacts of this work. Chapter 8.2: "Examples" describes them in detail.
Finally, each participant filled out a questionnaire. This procedure enables the detection of difficulties and shows where the API still has room for improvement.

The findings are based on the questionnaire evaluation and the code of three participants who handed in their solutions. This section lists all the improvements the user test pointed out. In addition, it concludes the general impression collected by the user test. Appendix C: "User Test Results" contains all the user test questions and their corresponding answers.

8.1.2. Findings for the Sequence library

Findings and Improvements

We decided to include the following findings from the user tests in the Sequence library:

- **Renaming untilFunction**: Creating sequences using the sequence constructor worked well for everyone. The meaning of the parameters was clear to all (C.1: "Question 3"). However, a suggestion for a meaningful optimization from two participants was to rename the `untilFunction` to `whileFunction`. This name states clearer that the function should return `false` when the sequence ends. (This suggestion was handed in by a solution)
- **Better type documentation in JSDoc**: The IDEs have problems correctly displaying type information for curried functions. Therefore, the already often used JSDoc annotation `@haskell`, which describes the type signature of a function as in Haskell, is added to every artefact of the Sequence library.

The following are great ideas but are beyond the scope of this project to implement:

- **unfoldr**: This suggestion was handed in by a solution
- **uncons with empty sequences**: This suggestion was handed in by a solution

Section 8.3.2: "Operators and Operations" lists them as future features and describes them.

Conclusion

Almost everyone frequently uses `map`, `filter`, `take` or `reduce` operations when working with lists. So the Sequence library brings much-needed functions. (C.1: "Question 2")
Figure 8.1 concludes the overall opinion about the Sequence library very well. It shows that most users think that it can be an advantage in their next project.

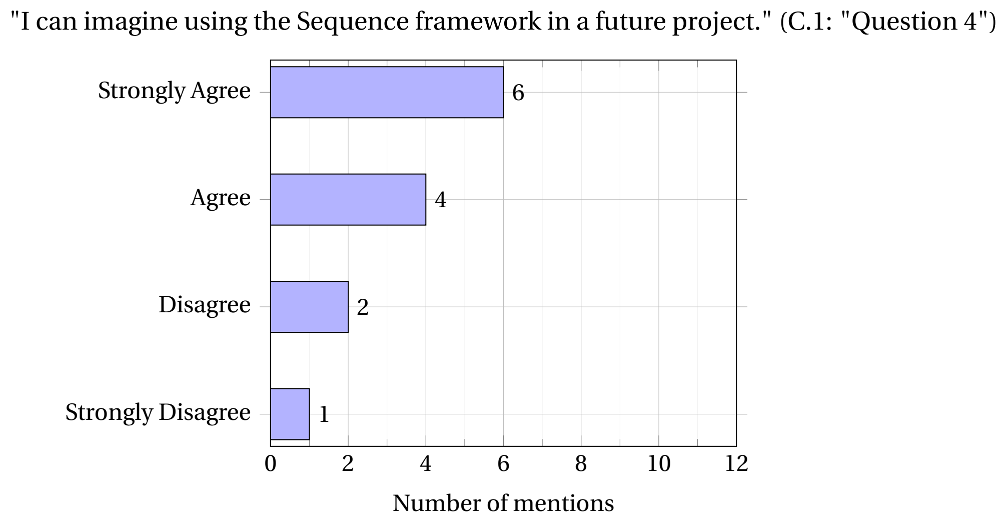


Figure 8.1.: Responses to "I can imagine using the Sequence framework in a future project."

8.1.3. Findings for JINQ

Findings and Improvements

We decided to include the following findings in JINQ:

- **Better Introduction to JINQ**: Two people suggest using `JSONMonad()` directly inside of `from` (text answer C.2: "Question 13"). However, this makes no sense since JINQ can handle arbitrary monads. Thus, the documentation provided did not give a sufficient overview. Therefore, the JSDoc of JINQ must give a better introduction to the topic. Additional examples should show the usage of JINQ using different monads.
- **Revise JSDoc (functions and examples)**: Nearly a quarter of the participants found JINQ challenging (C.2: "Question 8"). This shows that these concepts needed to be sufficiently well introduced. From the solutions given to us, it is evident that more JSDoc was required. In addition, more examples would coin to a better understanding (text answer C.2: "Question 13").
- **Not exactly the same as SQL**: Two people needed help with JINQ not writing exactly like SQL (handed in code). An SQL query `Select NAME from PERSON where AGE > 18` is translated into JINQ via `from(PERSON).where(p => p.age > 18).select(p => p.name);` So the order of `select` and `where` is reversed. Therefore, the JSDoc of JINQ needs to mention more clearly that working with JINQ is different than working with SQL.

The following is a great idea but is beyond the scope of this project to implement. Find detailed information about it in section 8.3.3: "JINQ Functions", explaining future features of JINQ.

- **Error handling**: Several people had trouble with null values, mainly when calling a function on a non-existent object property in the `select` and the `where` functions (text answer C.2: "Question 13", and handed in solution). In the discussion, the idea came up to provide functions like `notNull`, `safeWhere` or `safeMap` to catch this.

Conclusion

Although JINQ still has room for improvement, the general feedback is very good. Almost all find it helpful to search JSON structures using JINQ (C.2: "Question 10"). The majority already found the current state of JINQ easy to use (C.2: "Question 8").

As figure 8.2 shows, most people think that JINQ can be an asset in a future project:

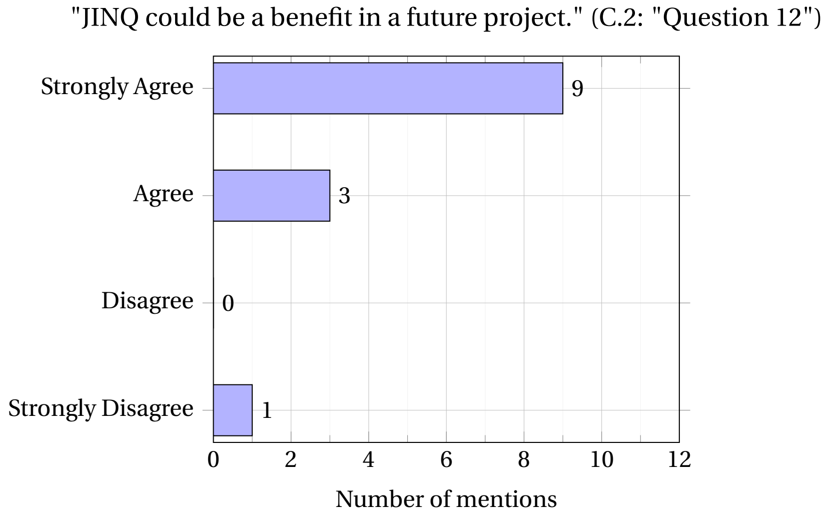


Figure 8.2.: Responses to "JINQ could be a benefit in a future project."

8.1.4. General Findings

The questionnaire concludes with general questions. The closing asks how effectively IDEs support JSDoc and whether the testers find this useful. Most participants used IntelliJ or WebStorm for this user test (C.3: "Question 15"). Figure 8.3 shows that the IDE was an excellent help for this test. Thus, It makes sense to write a detailed JSDoc. With the right IDEs, code navigation, code completion, etc., work very well.

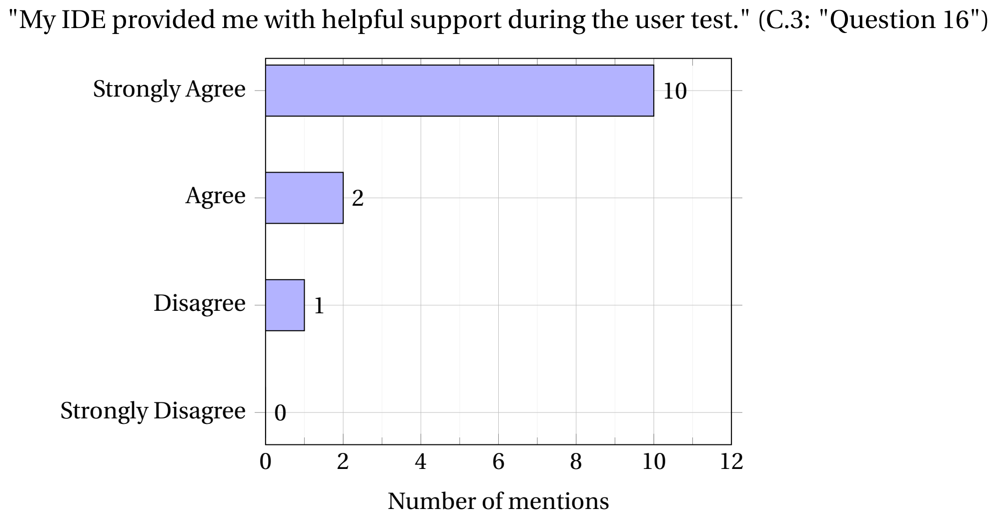


Figure 8.3.: Responses to "My IDE provided me with helpful support during the user test."

8.1.5. Conclusion

This user test shows that the features this library brings are beneficial. In addition, it offers some improvement points of the API. Implementing the documented hints improves the quality and usability of this library even more. We are delighted with the result and especially pleased with the text responses - many further underline the positive attitude towards this library.

8.2. Examples

This section shows some examples implemented with the Sequence library to demonstrate its variable usage. The following covers easy-to-understand and more sophisticated examples. Consult the listed sources to understand the applications thoroughly if something is unclear.

8.2.1. Fibonacci Sequence

This is an example of a specialized sequence constructor. Listing 8.1 shows the implementation of the Fibonacci sequence [25, p. 36]:

```
const start   = Pair(0)(1);
const whileFn =  _ => true;
const incrFn  = ([fst, snd]) => Pair(snd)(fst + snd);
/**
 * Generates the Fibonacci sequence.
 *
 * @constructor
 * @pure
 * @returns { SequenceType<Number> }
 *
 * @example
 * const result = take(8)(FibonacciSequence);
 *
 * console.log(...result);
 * // => Logs '1, 1, 2, 3, 5, 8, 13, 21'
 */
const FibonacciSequence =
                    map(pair => pair(snd))(Sequence(start, whileFn, incrFn));
```

Listing 8.1: FibonacciSequence implementation

The `FibonacciSequence` constructor generates the Fibonacci sequence using the `Sequence` and `Pair` constructors. The incrementation function produces a pair containing the last returned value and the value to be returned. The `map` function then extracts the current Fibonacci number from the pair returned by the sequence.

8.2.2. Fizz Buzz

Fizz Buzz is a simple game played with numbers, typically in a group setting. Players take turns counting upward, but instead of saying numbers divisible by 3, they say "Fizz", and instead of numbers divisible by 5, they say "Buzz". If a number is divisible by both 3 and 5, they say "FizzBuzz". This game also serves as a programming task.
Frege Goodness [9] includes a detailed explanation of the game.

First, let us look at the game's simplest form in listing 8.2. The fizzes and buzzes are combined in a sequence. The sequence returns a number if neither a fizz nor a buzz occurs:

```
const limit    = 15;
const range    = Range(1, limit);
const fizzez   = cycle(["", "", "fizz"]);
const buzzez   = cycle(["", "", "", "", "buzz"]);

const pattern  = zipWith((fizz, buzz) => fizz + buzz)(fizzez)(buzzez);
const fizzbuzz = zipWith((num, str) => str === "" ? num : str)(range)(pattern);

console.log(...take(limit)(fizzbuzz));
// => Logs '1 2 "fizz" 4 "buzz" ... 11 "fizz" 13 14 "fizzbuzz"'
```

Listing 8.2: Fizz Buzz example

As you observe, the entire setup relies on sequences. We define three distinct endless sequences. A range generates a line of numbers, while `cycle` produces an unending repetition for "fizzes" and "buzzes". Memory usage remains minimal. Subsequently, `zipWith` merges the individual sequences and yields a new sequence. This uncomplicated setup underscores that working with sequences is declarative, straightforward and highly understandable.

Now we are extending the application to add new rules during the runtime:

```
import * as _ from "./src/sequence/sequence.js";

const infiniteNumbers = _.Sequence(1, _ => true, i => i + 1);

const createSequenceForRule = rule =>
  _.pipe(
    // add rule's text to number
    _.map(a => a === rule.getNr() ? rule.getText() : ""),
    _.take(rule.getNr()), // abort on this rules number
    _.cycle
  )(infiniteNumbers);

const buildFizzBuzz = () => {
  const currentRules = model.rulesSnapshot().map(createSequenceForRule);
  const baseLine     = _.Sequence("", _ => true, _ => "");

  const fizzBuzz = _.pipe(
      // reduce to single sequence by combining all iterable values
    _.reduce$((acc, cur) => _.zipWith((a, b) => a + b)(acc)(cur), baseLine),
    _.zipWith((numbers, pattern) => pattern === "" ? String(numbers) : pattern)
                 (infiniteNumbers),
    // limit output
    _.take(model.getUpperBoundary()),
    _.drop(model.getLowerBoundary() -1),
  )(currentRules);

  model.setResult(fizzBuzz);
};
```

Listing 8.3: Fizz Buzz example extended

Listing 8.3 shows two functions, `createSequenceForRule`, and `buildFizzBuzz`.
`createSequenceForRule` is called with an argument `rule` representing a number and a string. The returned infinite sequence then contains the corresponding text at each multiple of the rules number. In the remaining places, it contains an empty string.
Calling `createSequenceForRule(Rule(3, "fizz"))` thus creates a sequence generating `"", "", "fizz", "", "", "fizz", ...`.
The function `buildFizzBuzz` reduces all sequences representing a rule to a single sequence. Figures 8.4 and 8.5 demonstrate a screenshot of a simple application showing two rules and the resulting sequence. The implementation enables adding new rules at runtime.

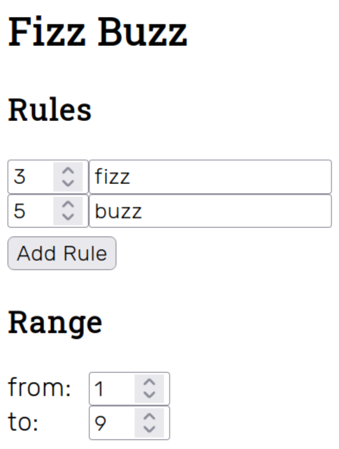


Figure 8.4.: Fizz Buzz controls

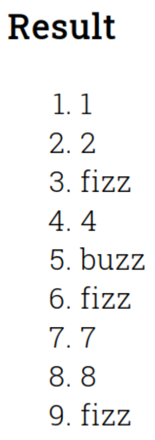


Figure 8.5.: Fizz Buzz result

8.2.3. JINQ

The following examples use JINQ to browse and process JSON files and to create new sequences based on rules. At first glance, these two topics seem to have nothing in common. However, as the following examples show, JINQ allows both and therefore brings a general way to create and manipulate various data structures.

Find all Students

We want to filter out all participants that have a student ID. As you can see in the excerpt of the JSON file in listing 8.4, the property `switch-edu-id` is not defined for all people. Because the `JsonMonad` works in the background with a maybe, such situations can be processed without null handling.

```json
[
  {
    "id": 1,
    "name": "",
    "age": 28,
    "salary": 50000,
    "favoriteLanguages": [1, 3, 5]
  },
  {
    "id": 2,
    "switch-edu-id": "12-432-23",
    "name": "Emma Johnson",
    "age": null,
    "salary": 60000,
    "favoriteLanguages": [2, 4]
  },
  {
    "id": 3,
    "name": "Sophia Davis",
    "age": 40,
    "salary": null,
    "favoriteLanguages": [null, 4, 5]
  },
  ...
 ]
```

Listing 8.4: Excerpt of a JSON File including developers

Using JINQ it is straightforward to access the desired students by just using the function `where`:

```javascript
const findAllStudentIds = developers => {
  const allIds =
    from(JsonMonad(developers))
      .select(x => x['switch-edu-id'])
      .result();
};
```

Listing 8.5: JINQ Example - find all students

Find Sophia's Programming Languages

This example combines two JSON files. We use the JSON file from the previous example and the one from listing 8.6 below. Listing 8.7 shows the code that finds Sophia's favourite programming languages by combining two JSON files using `pairWith`.

```json
[
  {
    "id": 1,
    "name": "Java"
  },
  ...
  {
    "id": 4,
    "name": "C++"
  },
  {
    "id": 5,
    "name": "Haskell"
  }
]
```

Listing 8.6: Excerpt of a JSON File including programming languages

This is also comparatively easy to do with JINQ. `pairWith` allows adding a second data source to the current one, forming all possible combinations. The function `where` then keeps only the needed pairs. A simple mapping at the end outputs the names of the programming languages.

```javascript
const sophiasProgrammingLanguages = (devs, languages) =>
    from(JsonMonad(devs))
      .where   ( dev    => dev.name === "Sophia Davis")
      .select  ( sophia => sophia.favoriteLanguages)
      .pairWith( JsonMonad(languages) )
      .where   ( ([langId, language]) => langId === language.id )
      .select  ( ([     _, language]) => language.name )
      .result  ();
```

Listing 8.7: JINQ example - find Sophia's programming languages

Pythagorean Triple

Listing 8.8 creates a new sequence containing the pythagorean triples [26] between 1 and 10:

```javascript
import { Range, show } from "./src/sequence/sequence.js"
import { from }        from "./src/jinq/jinq.js";

const range = Range(1, 10);

const result =
  from(range)
    .pairWith(range)
    .pairWith(range)
    .where ( ([ [a,b], c ]) => a * a + b * b === c * c)
    .select( ([ [a,b], c ]) => `[${a}, ${b}, ${c}]`)
    .result();

```

```
console.log(show(/** @type SequenceType */ result));
// => Logs '[[3, 4, 5],[4, 3, 5],[6, 8, 10],[8, 6, 10]]
```

Listing 8.8: The Pythagorean Triple between 1 and 10

8.2.4. Numerical Differentiation

This example shows a mathematical use case, the numerical differentiation using sequences:

```
const halve   = x => x / 2;
const repeatF = (f, x) => _.Sequence(x, _ => true, f);
const halves = h0 => repeatF( halve, h0);

const slope = f => x => h => (f(x + h) - f(x)) / h;
const differentiate = h0 => f => x => _.map ( slope(f)(x) ) (halves(h0));
```

Listing 8.9: Differentiation using sequences

Listing 8.9 implements the following steps:

- `repeatF` repeatedly applies the function f to a value of the previous calculated result, starting with x.
- `halves` is using `repeatF` and `halve` to halve a value $h0$.
- `slope` calculates the slope of a function f at the position x with delta h.

This code already provides everything needed for differentiation. The function `differentiate` calculates the slope of a function at a given point x more and more exactly. Listing 8.10 uses these functions to determine the slope at $x = 1$ of the function `parabola` defined on line 1:

```
1 const parabola = x => x * x;
2
3 const diffs = differentiate(0.5)(parabola)(1);
```

Listing 8.10: Differentiation using sequences

Since `diffs` contains an infinite sequence, one further function is still needed. Listing 8.11 shows the function `within`, which allows differentiating until two successive values are smaller than a given epsilon:

```
const within = eps => sequence => {
  const [a, rest] = _.uncons(sequence);
  const [b]       = _.uncons(rest);
  const diff      = Math.abs(a - b);

  if (diff <= eps) return b;
  return within(eps)(rest);
};
```

```
const slopeOfFAtX = within(0.000_1)(diffs);
```

Listing 8.11: Implementation of within

Listing 8.11 defines the following implementations:

- `within` calculates recursively the slope of the `parabola` using the sequence `diffs` until a satisfying accuracy. Because of the laziness of the sequence, `within` only calculates as many slopes as needed.
- At the end, `slopeOfFAtX` calculates the slope with the given parameters.

8.2.5. Tic Tac Toe with a Kind of AI

This section describes the implementation of the alpha-beta heuristic in JavaScript, an algorithm for estimating how good a position a game player is in.
This algorithm allows for building a computer-controlled opponent for a turn-based game. Hughes explains how the algorithm uses laziness and higher-order functions to be modular and extensible in his paper. [7, p. 16]. As chapter 3: "Modularizing Programs" describes, the Sequence library supports these two concepts, thus allowing to port the algorithm to JavaScript.

How does the Algorithm work?

The following explanation is meant to give a brief overview for the algorithm, allowing to understand the preceding code. For a deeper understanding of the algorithm, consider [7, Ch. 5] or the section "Incremental Development" in Frege Goodness [9].

Building a Game Tree The idea of the algorithm is to build a tree containing *all* possible playing fields. The start node in this tree is the current state of the playing field. Its children are all possible playing fields, which can arise when the players make their next moves. This results in a potentially endless tree. All direct successors of a node are the next possible moves for this node's playing field. Since this tree can be huge, depending on the game, even infinitely large, only a lazy data structure can handle it.
The algorithm must estimate whether a playfield is good or bad for a player to find the best possible next move. The simplest way to figure this out is to look at the playing field and decide if a player has already won. In the case of tic tac toe, a player has won when three of his symbols are next to each other.
A function (called `evalFunction`) does this evaluation - if the algorithm has won the game on a field, it evaluates to 1 if it has lost to -1, in all other cases, to 0.
The `evalFunction` is mapped over the tree (where higher-order functions come into play) - each node is assigned its static value, saying how promising the playfield of this node is. This results in a new tree with the value of `evalFunction` assigned to the nodes instead of all possible fields.

What is the best Move? The tree's root is the current playing field - the algorithm cannot know from the root which moves are good and which are not. The previously mentioned `evalFunction` will evaluate to 0 for most nodes directly after the root. Therefore, the computer also calculates subsequent moves. The values of the lowest calculated nodes are then propagated upwards to the root.

The function `maximize` goes through to the lowest precalculated level. It evaluates the board there: when it is the computer's turn, it selects the board with the highest value assigned (safest chance of winning). If it is the opponent's turn, it selects the smallest value (the algorithm thus assumes that the opponent also selects the best possible move). This found value is assigned to the parent node as a result - and chooses the best move on this level according to the same scheme. Ultimately, the root has the value with the best possible chances of winning.

How to evaluate the huge Tree? The tree of possible moves and possible result values can be huge. So the algorithm must have a way to limit the tree to a fixed depth. The algorithm uses pruning - from a freely selectable depth, it cuts the children off.

Implementation

Basic Types and Helper functions Listing 8.12 shows the required types for the game implementation.
A pair models the `Tree` (line 7) - the first value is the current node, and the second value is a sequence consisting of further trees. The types `Player` and `Stone` exist only for writing more expressive JSDoc.

```
1  /**
2   *
3   * @template _T_
4   * @typedef {SequenceType<Tree<_T_>>} TreeSequence
5   */
6
7  /**
8   * @template _T_
9   * @typedef { PairSelectorType<_T_, TreeSequence<_T_>> } Tree
10  */
11
12
13 /**
14  * @typedef { "Computer" | "Human" | "NoPlayer" } Player
15  */
16
17 /**
18  * @typedef { 1 | -1 | 0 } Stone
19  */
20
21 /**
22  * @typedef Board
23  * @property { Player } whosTurn
```

```
 * @property { Iterable<Stone>} fields - A board has fields from 0 to 8.
 */
```

Listing 8.12: Tic tac toe types

Listing 8.13 includes the definition of the players and some functions used in the game process. Since JavaScript does not support pattern matching, the functions `opponent` and `stone` use a workaround - they store the mappings in an object.

```
/** @type { Player } */
const Computer = "Computer";

/** @type { Player } */
const Human = "Human";

/** @type { Player } */
const NoPlayer = "NoPlayer";

/**
 * Returns the opponent of a given player.
 * @param { Player } player
 * @return Player
 */
const opponent = player => {
  const pairings = {
    "Computer" : Human,
    "Human"    : Computer,
    "NoPlayer" : NoPlayer,
  };
  return pairings[player];
};

/**
 * Returns the stone number of a given player.
 * @param { Player } player
 * @returns Stone
 */
const stone = player => {
  const pairings = {
    "Computer" :  1,
    "Human"    : -1,
    "NoPlayer" :  0,
  };
 return pairings[player];
};

/**
 * Transforms each board element using the given function f.
 * @template _T_, _U_
 * @type {
```

```
 *         (f: (Board) => _T_)
 *      => (tree: Tree<_U_>)
 *      => Tree<_T_>
 * }
 */
const treeMap = f => ([a, sub]) => Pair(f(a))(map(treeMap(f))(sub));
```

Listing 8.13: The players and some helper functions

Building trees and processing the Game Board Listing 8.14 shows the functions for building a game tree. `unfold` is a function that takes a value (for example, a board) and creates new values of the same types of it (for example, all boards possibly arising in one move).

```
/**
 * Generates recursively a game tree, which is potentially endless.
 * @template _T_
 * @type {
 *       (unfold: ((_T_) => Iterable<_T_>))
 *    => (a: _T_)
 *    => Tree<_T_>
 * }
 */
const buildTree = unfold => a => {
  const as = unfold(a);
  const children = map(buildTree(unfold))(as);
  return Pair(a)(children);
};

/**
 * Creates a game tree.
 * @param { Board } board
 * @returns Tree<Board>
 */
const gameTree = board => buildTree(moves)(board);
```

Listing 8.14: Tic tac toe game tree

Listing 8.15 includes the `moves` function. This function calculates all possible moves for a player based on the current playing field. If a player has already won, it returns an empty sequence.

```
/**
 * Indexes each field using a {@link PairType}.
 * @param { Iterable<Stone> } fields
 * @return { SequenceType<PairType<Stone, Number>> }
 */
const indexFields = fields => zip(fields)(Range(1,9));

/**
 * Calculates possible moves for a given board.
 * @param { Board } board
 * @return SequenceType<Board>
 */
const moves = board => {
  if (hasWon(board)(Computer)) return /**@type {SequenceType<Board>} */nil;
  if (hasWon(board)(Human))    return /**@type {SequenceType<Board>} */nil;

  const otherPlayer   = opponent(board.whosTurn);
  const indexedFields = indexFields(board.fields);

  const blankIndices =
    from(indexedFields)
      .where( ([content, _]) => content === stone(NoPlayer))
      .select( ([_, i]) => i)
      .result();

  const fieldsWithPlayerPlacedAt = pos =>
    from(indexedFields)
      .select(([content, i]) => i === pos ? stone(board.whosTurn) : content)
      .result();

  const boardFieldsAfterMove = map (fieldsWithPlayerPlacedAt) (blankIndices);

  return map(fields => ({fields, whosTurn: otherPlayer})) (boardFieldsAfterMove);
};
```

Listing 8.15: Tic tac toe move function

Evaluating a Board Listing 8.16 shows the implementation of the `hasWon` function, which checks if a player has won on a given board. The evaluation function, explained in section 8.2.5: "How does the Algorithm work?", uses this function to assess a playing field. Line 39 defines this function called `staticEval`.

```
1  /**
2   * Checks, if a player has won.
3   * @type {
4   *     (board: Board)
5   *     => (player: Player)
6   *     => Boolean
7   * }
8   */
9  const hasWon = board => player => {
10   const winTriples = [
11     [1,2,3], [4,5,6], [7,8,9], // row
12     [1,4,7], [2,5,8], [3,6,9], // col
13     [1,5,9], [3,5,7]           // diag
14   ];
15
16   const checkTriple = triple => {
    const actualStone   = stone(player);
    const indexedFields = indexFields(board.fields);

    const playerOnFields =
      from(indexedFields)
      .where( ([_, i])        => triple.includes(i))
      .select( ([content, _]) => content === actualStone)
      .result();

    return foldl$((acc, cur) => acc && cur, true)(playerOnFields);
  };
  return pipe(
    map(checkTriple),
    foldl$((acc, cur) => acc || cur, false)
  )(winTriples)
};

/**
 * Evaluates a given board.
 * @param { Board } board
 * @returns Number
 */
const staticEval = board => {
  if (hasWon (board) (Computer)) return 1.0;
  if (hasWon (board) (Human))    return -1.0;
  return 0.0;
};

```

Listing 8.16: Tic tac toe hasWon function

Finding the best Move Listing 8.17 shows the implementation of the functions `maximize` (and its counterpart `minimize`) explained in section 8.2.5: "How does the Algorithm work?". These functions determine which are the best moves.

```
/**
 * Determines the best move.
 * @template _T_
 * @param { Tree<_T_>}
 * @return { _T_ }
 */
const maximize = ([a, sub]) => {
  if (sub ["=="] (nil)) return a;
  return max$ (map(minimize)(sub))
};

/**
 * Determines the best move for the opponent.
 * @template _T_
 * @param { Tree<_T_>}
 * @return { _T_ }
 */
const minimize = ([a, sub]) => {
  if (sub ["=="] (nil)) return a;
  return min$ (map(maximize)(sub))
};
```

Listing 8.17: Tic tac toe minimax algorithmus

Pruning Trees We cut off (prune) the possibly infinite tree. The following listing 8.18 shows the implementation of this pruning. Since subtrees are sequences, `nil` replaces them when pruning.

```
/**
 * Prunes a given {@link Tree} to a max depth n.
 * @template _T_
 * @type {
 *           (n: Number)
 *        => (tree: Tree<_T_>)
 *        => Tree<_T_>
 * }
 * @param n
 */
const prune = n => tree => {
  const [a, sub] = tree;
  if (n === 0) return Pair(a)(nil);
  else return Pair(a)(map(prune(n-1))(sub));
};
```

Listing 8.18: Tic tac toe prune function

Putting it all together The following functions evaluate a playing field.
The function `nextBoard` calculates the best possible next move for the computer. The number `lookaheads` defines how many turns the algorithm calculates in advance. Based on these calculations the algorithm chooses the next move.

```
/**
 * Evaluates a given Board by building a tree
 * @type {
 *           (lookahead: Number)
 *        => (board: Board)
 *        => Number
 * }
 */
const evaluate = lookahead => board => {
  const prunedTree = prune(lookahead)(gameTree(board));
  const mappedTree = treeMap(staticEval)(prunedTree);
  return minimize(mappedTree);
};

/**
 * @template _T_
 * @type {
 *           (lookahead: Number)
 *        => (board: Board)
 *        => PairSelectorType<_T_, Board>
 * }
 */
const nowValue = lookahead => board =>
  Pair(evaluate(lookahead)(board))(board);

/**
 * Calculates the next board with given board fields.
 * @type {
 *          (lookahead: Number)
 *       => (inFields: Array<Number>)
 *       => Board
 * }
 */
const nextBoard = lookahead => inFields => {
  const currentBoard = { whosTurn: Computer, fields: inFields };
  // get all possible
  const possibleMoves  = moves (currentBoard);
  if (isEmpty(possibleMoves)) return currentBoard;
  // evaluate each move by looking ahead how good it is
  let evaluatedMoves =  map (nowValue(lookahead)) (possibleMoves);

  // if the computer is sure to lose
  // only look one place ahead to prevent the "nearest" loss
  if (onlyLoses(evaluatedMoves)) {
    evaluatedMoves = map (nowValue(1)) (possibleMoves);
  }

  /**
   * Gets the highest ranked board of the passed board
   * @param {SequenceType<PairSelectorType<Number, Board>>} boards
   * @return Board
   */
  return bestOf(evaluatedMoves);
};
```

Listing 8.19: Tic tac toe - evaluating functions

The Result

One advantage of JavaScript is that it works hand in hand with websites and HTML. A website with a UI (like figure 8.6) can now use the algorithm:

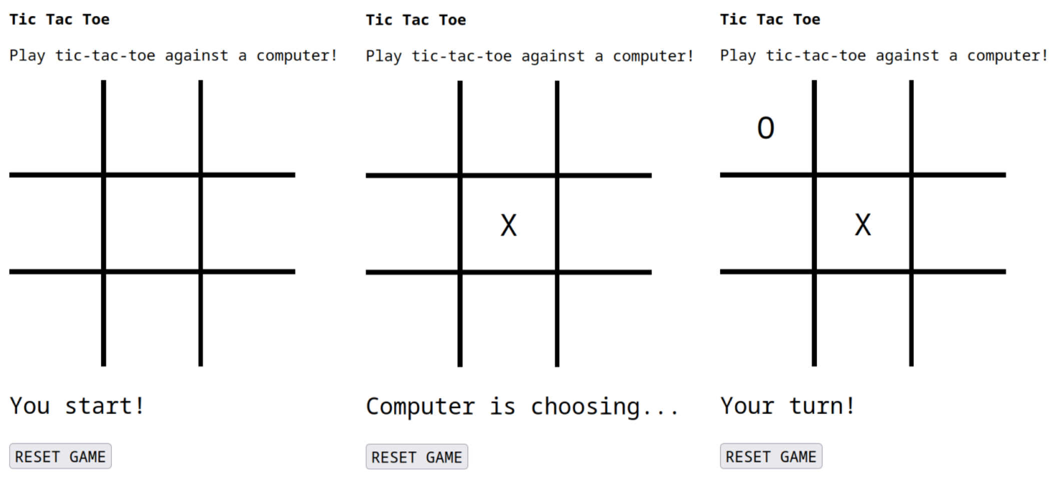


Figure 8.6.: A website using the algorithm

When a player places his stone on the board, the function `nextBoard` gets called. The algorithm automatically makes the best possible next turn based on how many moves it can look ahead and returns the new board.

Conclusion

This example shows impressively how the Sequence library enables developers to adapt an algorithm developed for functional programming languages in JavaScript.
Of course, copying the algorithm is impossible because JavaScript misses some language features like pattern matching. Nevertheless, the essential concepts and logic of the algorithm can be adopted directly.
It is also exciting how JSDoc allows constructing straightforward talking types like `Player` or `Stone` (listing 8.12).
To extend the algorithm, for example, to improve the static evaluation to decide how good a current playing field is, only one function has to be adjusted. Also, the evaluation of the algorithm can be easily improved and detached from other logic by increasing the number of lookahead moves. Further optimizations to prune away subtrees that are out of consideration anyway can be done directly in the function `maximize`.
Thus, the modularity and extensibility of the program is very high.

8.3. Future Features

During the project, some ideas emerged that were not feasible for the time being for various reasons. This section describes these possible extensions for the standard library.

8.3.1. Logging

Logging is a topic that has come up several times. On the one hand, it appears during the implementation of the Sequence library, but also through feedback from a test proband of the user test. In the predecessor project, we implemented a logging framework [24]. This would also be useful to include in the Sequence library. Especially in case of errors, analyzing debug or tracing messages from the involved functions would be helpful. Currently, the logging framework is only included in a part of the test framework. But this could be extended in a further step.

8.3.2. Operators and Operations

Following are some possible extensions for the Sequence library:

uncons for empty Sequences

`uncons` is an operation that returns the head and the rest of an iterable in a pair. The Sequence library already includes this operation. But it does not work on empty sequences. Meaning `uncons` applied to an empty iterable fails. Thus, one would have a version, which for example, wraps the result in a maybe. `uncons` on an empty list would then return `Nothing`. The API documentation page [27] describes the function in detail.

tail

`tail` [28] is a useful function that removes the first element of an iterable and returns the rest of it. By implementing `tail`, one has to pay attention to empty or single-valued iterables, which do not have a tail. An option could be to return a maybe. `Nothing`, if the list has one or zero elements, and `Just` including the result otherwise.

unfoldr

`unfoldr` is a powerful concept for creating lists in Haskell. It would supplement the options to create new sequences besides the constructor `Sequence`. In Haskell, it works using a `Maybe`. `unfoldr` creates a list that returns a pair wrapped in `Just` with the value of the current iteration and the next element. If the stop condition is fulfilled, `unfoldr` returns `Nothing`. For further information, look at the documentation on [29].

iterate

`iterate` takes two arguments. The first argument is a function, and the second is a start value. It generates an infinite iterable of repeated applications of the function to the calculated value. You will find more information about `iterate` on hoogle [30].

8.3.3. JINQ Functions

It is possible to get errors when working with `JINQ` and `JSONMonad` by grabbing not present properties. That means if a function in `select` accesses a property which is not defined, it will throw an error. This kind of null-case handling takes a lot of work. To remedy this situation, having error-safe functions like `safeSelect` to access probably not existing values would be excellent.

8.3.4. Applications

Fix Point Sequence

In the mathematical context, having a tool for approximations is helpful. A constructor for such a case could be the fix point sequence. It allows us to find an approximation with a given number of iterations or by providing a value of minimal deviation. The function `within` described in section 8.2.4: "Numerical Differentiation" already serves as a good starting point.

Use JINQ to process HTML

Scalpel [31] is a Haskell library to scrap web content. Building a similar application based on the present implementations could be possible. With `JINQ` and `JsonMonad`, the standard library already contains a scraping tool for JSON Objects. A future implementation could use a similar approach to develop such an application.

8.4. Conclusion

8.4.1. Findings and Achievements

This thesis aimed to develop a functional standard library for the Kolibri Web UI Toolkit in JavaScript. To achieve this, we delved into the iterable protocols of JavaScript, leveraging our understanding to construct a new data structure: the sequence. The characteristics of sequences are lazy evaluation and immutability, which offer powerful ways of handling iterable objects. Based on these findings, the Sequence library emerged. It can process, transform, and create sequences by decorating existing iterables. This library does not only work with sequences but also with every iterable object, such as JavaScript arrays - passing the receiver to the functions enabled this.
The power of eta reduction manifests itself in the pipe function, enabling the execution of multiple

operations sequentially, dramatically improving code readability. With the Sequence library's compatibility with all iterable objects, it became logical to make other existing data structures iterable as well. As a result, existing collections of the Kolibri, such as pair, stack, and tuple, are now iterable and, therefore, compatible with the Sequence library.
Typing the Sequence library using JSDoc ensures clarity and maintains consistency.

The work also shows that monads in JavaScript can be very helpful and enable new ways to reason about problems. For example, JINQ enables the processing of various data structures that provide a monadic API in a very declarative way. The JSON monad makes JINQ compatible with arrays of JavaScript objects, solving daily problems that web developers face.

In addition, many examples show the strengths of the standard library. Its tools enable easy transfers of Haskell algorithms based on lists to JavaScript.
Despite some challenges, a powerful functional standard library for the Kolibri Web UI Toolkit emerged, offering enhanced functionality and versatility to JavaScript developers.

8.4.2. A Closer Look to Particular Findings

Similarity to Haskell

As Haskell is a well-established and widely used functional language, it serves as a great role model for solving problems and making decisions.
However, it is essential to note that some remarkable language concepts of Haskell do not exist in JavaScript, making it harder to translate code directly from one language to the other.
Nevertheless, specific implementations are pretty similar, as the following listings 8.20 and 8.21 demonstrate:

```haskell
-- Creating a list from 0 to 4
list = unfoldr (\x -> if x < 5 then Just(x, x + 1) else Nothing) 0

-- mapping the list
map (\x -> x * 2) list
```

Listing 8.20: Haskell vs. Sequence library - Haskell implementation

```javascript
// Creating a list from 0 to 4
const list = Sequence(0, x => x < 5, x => x + 1);

// mapping the list
map(x => x * 2)(list)
```

Listing 8.21: Haskell vs. Sequence library - JavaScript implementation

Robust Programming in JavaScript

Strange situations arise in JavaScript more often than in other programming languages. One contributing factor is the nature of its type system. However, this thesis demonstrates that functional programming concepts and distinct usage of JSDoc allow the development of reasonably robust programs.
Crucially, this requires taking advantage of the functional aspects inherent to the language, such as higher-order functions and partial application. Nevertheless, certain limitations sometimes pose significant problems on the solution path. In particular, not all desired behaviours are achievable when dealing with the type system: JSDoc is very limited when combining multiple types or writing general functions that need types to abstract over another type. An illustrative example of this is the combination of a monad with an iterable: when a function returns a monad, the valuable information that this monad could also be iterable is lost. Resolving such a situation requires type-casting.
The user test shows that modern IDEs deliver excellent support for JSDoc and help the developers get along the way when working with JavaScript.

Monadic Structures

With the `MonadType`, the standard library offers a type that defines a monadic interface. Any data structures can now implement this type and are thus compatible with all functions that can handle monadic types. The standard library thus lays essential foundations for implementing monads in JavaScript. Nevertheless, proper sleights and typecasting are required to circumvent the limitations of JSDoc.
JINQ is an example that already uses this type and shows the power of such abstractions. As examples from section 8.2.3: "JINQ" show, JINQ can traverse JSON structures and assemble new lists, which are two completely different tasks. In both cases, the resulting code is straightforward to understand.

Testing

A reliable test framework is crucial for achieving a robust and sustainable library. In the case of the Sequence library, the testing table attained the stability of the Sequence library. This lies a solid foundation for incremental growth. The library's core became increasingly robust by gradually incorporating new functionalities and corresponding test cases. Moreover, novel test concepts were introduced during the development process to enhance the testing capabilities further. A notable example is the introduction of invariant tests (section 6.4.1: "Invariant Testing"), which systematically assess the behaviours of functionalities through diverse testing approaches, ensuring comprehensive validation.

8.4.3. Non-Functional Findings

Library Organization

During the development of the Sequence library, we consistently paid close attention to non-functional aspects. As part of this effort, we adjusted the organization of the project's development setup. This led to the current state, where each operation of the Sequence library is in a separate file. Adopting such a project structure has significantly impacted the codebase's overall clarity and navigability. Moreover, it contributes to a sense of order, which, in turn, enhances the overall code quality. Additionally, it helps to prevent cycling dependencies when importing particular functionalities.

Code Quality

Maintaining high code quality standards led to a clear and consistent code base. Several points were particularly important during development:

- No code duplications
- Good naming
- Standardized formatting
- Only necessary exports of functions
- Appropriate comments and JSDoc, including examples
- Project organization and structure

Strict adherence to these principles made development easier for us and helped library users find their way around more quickly. Additionally, it facilitates future developers to learn how to add new functionalities.

8.4.4. User Testing

The user testing conducted in section 8.4.4: "User Testing" was crucial for assessing the usefulness of the Sequence library. It provided valuable feedback from programmers without prior knowledge of the library. These insights allowed for addressing specific issues and offered helpful improvements. Additionally, other useful suggestions contributed to the list of potential enhancements, as outlined in section 8.3: "Future Features".

Bibliography

[1] “Data.list.” (), [Online]. Available: https://hackage.haskell.org/package/base-4.17.0.0/docs/Data-List.html (visited on 07/09/2023).

[2] “Iteration protocols - JavaScript MDN.” (May 3, 2023), [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Iteration_protocols (visited on 06/16/2023).

[3] A. Wild and T. Wyss. “IP6 functional standard library for the kolibri web UI toolkit.” (), [Online]. Available: https://github.com/wildwyss/Kolibri/tree/main/contrib/wild_wyss (visited on 08/07/2023).

[4] “Kolibri.” (), [Online]. Available: https://webengineering-fhnw.github.io/Kolibri/ (visited on 06/20/2023).

[5] “Javascript modules - javascript MDN.” (Jul. 10, 2023), [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules (visited on 07/31/2023).

[6] Bill Wagner. “Language-integrated query (LINQ) (c#).” (Mar. 9, 2023), [Online]. Available: https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/ (visited on 06/21/2023).

[7] J. Hughes, “Why functional programming matters,” *The Computer Journal*, vol. 32, no. 2, pp. 98–107, Feb. 1, 1989, ISSN: 0010-4620, 1460-2067. DOI: 10.1093/comjnl/32.2.98. [Online]. Available: https://academic.oup.com/comjnl/article-lookup/doi/10.1093/comjnl/32.2.98 (visited on 06/15/2023).

[8] “Functions - JavaScript MDN.” (Apr. 5, 2023), [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions (visited on 08/02/2023).

[9] “Introduction · frege goodness.” (), [Online]. Available: https://dierk.gitbooks.io/fregegoodness/content/ (visited on 06/15/2023).

[10] K. Claessen and J. Hughes, “QuickCheck: A lightweight tool for random testing of haskell programs,”

[11] “Lodash.” (), [Online]. Available: https://lodash.com/ (visited on 06/26/2023).

[12] “RxJS.” (), [Online]. Available: https://rxjs.dev/ (visited on 06/26/2023).

[13] P. Andermatt and B. Brodwolf. “Lambda kalkül für praktisches JavaScript - lambda kalkül für javascript.” (Nov. 3, 2022), [Online]. Available: https://mattwolf-corporation.gitbook.io/ip6-lambda-calculus/ (visited on 06/26/2023).

[14] “Python range() function.” (), [Online]. Available: https://www.w3schools.com/python/ref_func_range.asp (visited on 07/09/2023).

[15] “Ranges and progressions kotlin,” Kotlin Help. (), [Online]. Available: https://kotlinlang.org/docs/ranges.html (visited on 07/09/2023).

[16] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st ed. Addison-Wesley Professional, 1994, ISBN: 0201633612.

[17] "Stream (java platform SE 8)." (), [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html (visited on 06/20/2023).

[18] K. Eilebrecht and G. Starke, *Patterns kompakt: Entwurfsmuster für effektive Softwareentwicklung* (IT kompakt). Berlin, Heidelberg: Springer Berlin Heidelberg, 2019, ISBN: 978-3-662-57936-7 978-3-662-57937-4. DOI: 10.1007/978-3-662-57937-4. [Online]. Available: http://link.springer.com/10.1007/978-3-662-57937-4 (visited on 06/21/2023).

[19] "Object prototypes - learn web development MDN." (Feb. 24, 2023), [Online]. Available: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object_prototypes (visited on 06/19/2023).

[20] "Use JSDoc: Index." (), [Online]. Available: https://jsdoc.app/ (visited on 06/22/2023).

[21] G. Hutton, *Programming in Haskell : Graham Hutton*, 2nd ed. Cambridge University Press, Sep. 2016, ISBN: 978-1-316-62622-1.

[22] "Functors, applicatives, and monads in pictures - adit.io." (Apr. 17, 2013), [Online]. Available: https://www.adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html (visited on 06/19/2023).

[23] baeldung, *Higher-Kinded Types Baeldung on Scala*, en-US, Aug. 2020. [Online]. Available: https://www.baeldung.com/scala/higher-kinded-types (visited on 06/20/2023).

[24] A. Wild and T. Wyss. "IP5 functional standard library for the kolibri web UI toolkit." (Jan. 20, 2023), [Online]. Available: https://wildwyss.gitbook.io/ip5-funktionale-standard-library-fuer-kolibri/ (visited on 06/23/2023).

[25] A. Beutelspacher and M.-A. Zschiegner, *Diskrete Mathematik für Einsteiger: mit Anwendungen in Technik und Informatik* (Studium), 4., aktualisierte Aufl. Wiesbaden: Vieweg + Teubner, 2011, 254 pp., ISBN: 978-3-8348-1248-3.

[26] E. W. Weisstein. "Pythagorean triple." Publisher: Wolfram Research, Inc. (), [Online]. Available: https://mathworld.wolfram.com/ (visited on 06/26/2023).

[27] "Data.list." (), [Online]. Available: https://hackage.haskell.org/package/base-4.18.0.0/docs/Data-List.html#v:uncons (visited on 06/27/2023).

[28] "Prelude." (), [Online]. Available: https://hackage.haskell.org/package/base-4.18.0.0/docs/Prelude.html#v:tail (visited on 06/27/2023).

[29] "Data.list." (), [Online]. Available: https://hackage.haskell.org/package/base-4.18.0.0/docs/Data-List.html#v:unfoldr (visited on 06/27/2023).

[30] "Prelude." (), [Online]. Available: https://hackage.haskell.org/package/base-4.18.0.0/docs/Prelude.html#v:iterate (visited on 06/27/2023).

[31] "Scalpel," Hackage. (), [Online]. Available: //hackage.haskell.org/package/scalpel (visited on 06/27/2023).

Appendices

A. Declaration of Academic Integrity

We the undersigned declare that all material presented in this bachelor thesis is our own work and written independently only using the indicated sources. The passages taken verbatim or in content from the listed sources are marked as a quotation or paraphrased. We declare that all statements and information contained herein are true, correct and accurate to the best of our knowledge and belief. This paper or part of it have not been published to date. It has thus not been made available to other interested parties or examination boards.

Windisch, 18. August 2023

Name: Andri Wild

Signature: [illegible]

Name: Tobias Wyss

Signature: [illegible]

B. API

B.1. Constructors

All functions listed below create new sequences from non-sequence values. They are therefore called constructors.

nil

This constant represents a sequence containing no values.

Returns: `SequenceType<*>`

Example

```
const emptySequence = nil;

console.log(...emptySequence);
// => Logs '' (nothing)
```

PureSequence

Creates a sequence which contains just the given value.

Returns: `SequenceType<_T_>`

Parameters

Name	Type	Description
`value`	`_T_`	the single value

Example

```
const seq = PureSequence(1);

console.log(...seq);
// => Logs '1'
```

Range

Creates a range of numbers between two inclusive boundaries, that implements the JS iteration protocols. First and second boundaries can be specified in arbitrary order, step size is always the third parameter.

Returns: `SequenceType<_T_>`

Parameters

Name	Type	Description
`firstBoundary`	`Number`	the first boundary of the range
`secondBoundary`	`Number`	optionally the second boundary of the range
`step`	`Number`	the size of a step, processed during each iteration

Example

```
const range              = Range(3);
const [five, three, one]  = Range(1, 5, -2);
const [three, four, five] = Range(5, 3);

console.log(...range);
// => Logs '0, 1, 2, 3'
```

repeat

Returns a Sequence that will repeatedly yield the value of `arg` when iterated over. `repeat` will never be exhausted.

Returns: `SequenceType<_T_>`

Parameters

Name	Type	Description
`arg`	`_T_`	the value to repeat

Example

```
const ones   = repeat(1);
const result = take(3)(ones);

console.log(...result);
// => Logs '1, 1, 1'
```

replicate

`replicate(n)(x)` creates a Sequence of length `n` with `x` the value of every element.

Returns: `SequenceType<_T_>`

Parameters

Name	Type	Description
`n`	`Number`	how many times the valued should be repeated
`x`	`_T_`	the value to repeat

Example

```
const trues = replicate(3)(true);

console.log(...trues);
// => Logs 'true, true, true'
```

Sequence

The `incrementFunction` should change the value (make progress) in a way that the `ntilFunction` function can recognize the end of the sequence.

Contract:

- `incrementFunction` & `untilFunction` should not refer to any mutable state variable (because of side effect) in the closure.

Returns: `SequenceType<_T_>`

Parameters

Name	Type	Description
`start`	`Number`	the first value to be returned by this sequence
`whileFunction`	`(_T_) => Boolean`	returns false if the iteration should stop
`incrementFunction`	`(_T_) => T`	calculates the next value based on the previous

Example

```
const start      = 0;
const untilF     = x => x < 3;
const incrementF = x => x + 1;
const sequence   = Sequence(start, untilF, incrementF);

console.log(...sequence);
// => Logs '0, 1, 2'
```

StackSequence

Creates a `SequenceType` on top of the given `stack`.

Returns: `SequenceType<_T_>`

Parameters

Name	Type	Description
`stack`	`stack`	each iteration returns the next element of this stack

Example

```
const stack         = push(push(push(emptyStack)(1))(2))(3);
const stackSequence = StackSequence(stack);

console.log(...stackSequence);
// => Logs: '3, 2, 1'
```

TupleSequence

Constructs a new sequence based on the given tuple.

Returns: `SequenceType<_T_>`

Parameters

Name	Type	Description
`count`	`(f:ArrayApplierType<_T_>) => any`	each iteration returns an element of the tuple

Example

```
const [Triple]      = Tuple(3);
const triple        = Triple(1)(2)(3);
const tupleSequence = TupleSequence(triple);

console.log(...tupleSequence);
// => Logs '1, 2, 3'
```

B.2. Operators

All of the following functions operate on sequences and iterables. They all return a new sequence.

bind

Applies the given function to each element of the `Iterable` and flattens it afterward. Note: This operation adds a monadic API to the `SequenceType`.

Returns: `SequenceType<_U_>`

Parameters

Name	Type	Description
`bindFn`	`<_U_>(bindFn: (_T_) => Iterable<_U_>)`	this function will be applied to each element of the iterable
`it`	`Iterable<_T_>`	the receiver of this operator

Example

```
const numbers = [0, 1, 2, 3];
const bindFn  = el => take(el)(repeat(el));
const result  = bind(bindFn)(numbers);

console.log(...result);
// => Logs '1, 2, 2, 3, 3, 3'
```

catMaybes

The catMaybes function takes an iterable of maybes and returns a sequence of all the Just's values.

Returns: `SequenceType<_T_>`

Parameters

Name	Type	Description
`it`	`Iterable<MaybeType<_T_>>`	the receiver of this operator

Example

```
const maybes = [Just(5), Just(3), Nothing];
const result = catMaybes(maybes);

console.log(...result);
// => Logs '5, 3'
```

concat

Adds the second iterable to the first iterables end.

Returns: `SequenceType<_T_>`

Parameters

Name	Type	Description
`it1`	`Iterable<_T_>`	the receiver of this operator
`it2`	`Iterable<_T_>`	the iterable to concat

Example

```
const numbers = [0, 1, 2];
const range   = Range(2);
const concatenated = concat(numbers)(range);

console.log(...concatenated);
// => Logs '0, 1, 0, 1, 2'
```

cons

Adds the given element to the front of an iterable.

Returns: `SequenceType<_T_>`

Parameters

Name	Type	Description
`element`	`_T_`	the value to put in front
`it`	`Iterable<_T_>`	the receiver of this operator

Example

```
const numbers  = [1, 2, 3];
const element  = 0;
const consed = cons(element)(numbers);

console.log(...consed);
// => Logs '0, 1, 2, 3, 4'
```

cycle

Ties a finite iterable into a circular one, or equivalently, the infinite repetition of the original iterable.

Returns: `SequenceType<_T_>`

Parameters

Name	Type	Description
`it`	`Iterable<_T_>`	the receiver of this operator

Example

```
const numbers = [0, 1, 2];
const cycled = cycle(numbers);
const result = take(6)(cycled);

console.log(...result);
// => Logs '0, 1, 2, 0, 1, 2'
```

drop

Jumps over so many elements.

Returns: `SequenceType<_T_>`

Parameters

Name	Type	Description
`count`	`Number`	the amount of elements to drop
`it`	`Iterable<_T_>`	the receiver of this operator

Example

```
const numbers = [0, 1, 2, 3];
const dropped = drop(2)(numbers);

console.log(...dropped);
// => Logs '2, 3'
```

dropWhile

Discards all elements until the first element does not satisfy the predicate anymore.

Returns: `SequenceType<_T_>`

Parameters

Name	Type	Description
`predicate`	`Predicate<_T_>`	drops elements fulfilling this predicate
`it`	`Iterable<_T_>`	the receiver of this operator

Example

```
const numbers = [0, 1, 2, 3, 4, 5];
const dropped = dropWhile(el => el < 2)(numbers);

console.log(...dropped);
// => Logs '2, 3, 4, 5'
```

map

Transforms each element using the given function.

Returns: `SequenceType<_U_>`

Parameters

Name	Type	Description
`mapper`	`Functor<_U_, _T_>`	is applied on each element
`it`	`Iterable<_T_>`	the receiver of this operator

Example

```
const numbers = [0, 1, 2];
const mapped  = map(el => el * 2)(numbers);

console.log(...numbers);
// => Logs '0, 2, 4'
```

mconcat

Flatten an iterable of iterables.

Returns: `SequenceType<_T_>`

Parameters

Name	Type	Description
`it`	`Iterable<Iterable<_T_>>`	the receiver of this operator

Example

```
const ranges = map(x => Range(x))(Range(2));
const result = mconcat(ranges);

console.log(...result);
// => Logs '0, 0, 1, 0, 1, 2'
```

pipe

Transforms the given iterable using the passed operators.

Returns: `SequenceType<_T_> | *`

Parameters

Name	Type	Description
`transformers`	`SequenceOperation<*,*>`	the operators to apply
`it`	`Iterable<_T_>`	the receiver of this operator

Example

```
const piped = pipe(
                retainAll(n => n % 2 === 0),
               map(n => 2*n),
               drop(2)
             )([0,1,2,3,4,5]);

console.log(...piped);
// => Logs '0, 4, 8'
```

rejectAll

Only keeps elements which do not fulfil the given predicate.

Returns: `SequenceType<_T_>`

Parameters

Name	Type	Description
`predicate`	`Predicate<_T_>`	ignores elements fulfilling this predicate
`it`	`Iterable<_T_>`	the receiver of this operator

Example

```
const filtered = rejectAll(el => el % 2 === 0)([0, 1, 2, 3, 4, 5]);

console.log(...filtered);
// => Logs '1, 3, 5'
```

retainAll

Only keeps elements which fulfil the given predicate.

Returns: `SequenceType<_T_>`

Parameters

Name	Type	Description
`predicate`	`Predicate<_T_>`	keeps elements fulfilling this predicate
`it`	`Iterable<_T_>`	the receiver of this operator

Example

```
const filtered = retainAll(el => el % 2 === 0)([0, 1, 2, 3, 4, 5]);

console.log(...filtered);
// => Logs '0, 2, 4'
```

reverse$

Processes the iterable backwards.

Returns: `SequenceType<_T_>`

Parameters

Name	Type	Description
`it`	`Iterable<_T_>`	the receiver of this operator

Example

```
const reversed = reverse$([0, 1, 2]);

console.log(...reversed);
// => Logs '2, 1, 0'
```

snoc

Adds the given element to the end of the iterable. It is the opposite of `cons`

Returns: `SequenceType<_T_>`

Parameters

Name	Type	Description
`element`	`_T_`	the element to add
`it`	`Iterable<_T_>`	the receiver of this operator

Example

```
const snocced = snoc(7)([0, 1, 2, 3]);

console.log(...snocced);
// => Logs '0, 1, 2, 3, 7'
```

take

Stop after so many elements.

Returns: `SequenceType<_T_>`

Parameters

Name	Type	Description
`count`	`Number`	he amount of elements to keep
`it`	`Iterable<_T_>`	the receiver of this operator

Example

```
const taken   = take(2)([0,1,2,3]);

console.log(...taken);
// => Logs '0, 1'
```

takeWhile

Proceeds with the iteration until the predicate becomes true.

Returns: `SequenceType<_T_>`

Parameters

Name	Type	Description
`predicate`	`Predicate<_T_>`	keeps elements until this predicate fails
`it`	`Iterable<_T_>`	the receiver of this operator

Example

```
const dropped = takeWhile(el => el <= 2)([0, 1, 2, 3, 4 ,5]);

console.log(...result);
// => Logs '0, 1, 2'
```

zip

Zip takes two iterables and returns an iterable of corresponding pairs.

Returns: `SequenceType<_T_>`

Parameters

Name	Type	Description
`it1`	`Iterable<_T_>`	the receiver of this operator
`it2`	`Iterable<_U_>`	zthe second iterable

Example

```
const numbers = [0, 1, 2],
const range   = Range(3, 5);
const zipped  = zip(numbers)(range);

forEach$(x => console.log(...x))(zipped);
// => Logs '0 3, 1 4, 2 5'
```

zipWith

Generalises zip by zipping with the function given as the first argument.

Returns: `SequenceType<_T_>`

Parameters

Name	Type	Description
`zipper`	`BiFunction<_T_, _U_, _V_>`	The function to combine two elements.
`it`	`Iterable<_T_>`	The receiver of this operator.

Example

```
const numbers = [0, 1, 2];
const range   = Range(2, 4);
const zipped  = zipWith((i,j) => [i,j])(numbers)(range);
console.log(...zipped);

```

```
// => Logs '[0,2], [1,3], [2,4]'
```

B.3. Terminal operations

All of the following operations transform an iterable into a single value.

eq$

Checks the equality of two non-infinite iterables.

Note: Two iterables are considered as equal if they contain or create the exactly same values in the same order. Use `["=="]` defined on the SequencePrototype to perform a comparison in a more readable form.

Returns: `Boolean`

Parameters

Name	Type	Description
`it1`	`Iterable<_T_>`	The first iterable
`it2`	`Iterable<_T_>`	the receiver of this operation

Example

```
const numbers = [0, 1, 2, 3];
const range   = Range(3);
const result  = eq$(numbers)(range);

console.log(result);
// => Logs 'true'
```

foldr

Performs a reduction on the elements from right to left, using the provided start value and an accumulation function, and returns the reduced value.

Since `foldr` reduces the iterable from right to left, it needs O(n) memory to run the function. Therefore `reduce$` is the better alternative for most cases

Returns: `_T_`

Parameters

Name	Type	Description
`accumulationFn`	`BiFunction<_U_, _T_, _T_>`	the reduction function
`start`	`_T_`	the first value
`it`	`Iterable<_T_>`	the receiver of this operation

Example

```
const numbers = [0, 1, 2, 3, 4, 5];
const result  = foldr$((cur, acc) => cur + acc, 0)(numbers);

console.log(result);
// => Logs '15'
```

forEach$

Executes the callback for each element.

Returns: `void`

Parameters

Name	Type	Description
`callback`	`Consumer<_T_>`	the callback to execute for each element
`it`	`Iterable<_T_>`	the receiver of this operation

Example

```
const container = [];
forEach$(cur => container.push(cur))([0, 1, 2, 3, 4];);

console.log(...container);
// => Logs '0, 1, 2, 3, 4'
```

head

Return the next value without consuming it. `undefined` when there is no value.

Returns: `_T_`

Parameters

Name	Type	Description
`it`	`Iterable<_T_>`	the receiver of this operation

Example

```
const numbers = [1, 2, 3, 4];
const result  = head(numbers);

console.log(result);
// => Logs '1'
```

isEmpty

Returns `true`, if the iterables head is undefined.

Returns: `Boolean`

Parameters

Name	Type	Description
`it`	`Iterable<_T_>`	the receiver of this operation

Example

```
const empty     = []
const result = isEmpty(empty);

console.log(result);
// Logs 'true'
```

max$

Returns the largest element of a **non-empty** iterable.

Note:

To determine the largest element, a comparator function is used. This function compares two elements by default with the `< (LT)` operator, where on the left side is the current largest element when processing the iterable. If needed, a different comparator can also be passed as a second argument to `max$` and will then be used to determine the largest element.

Returns: `_T_`

throws: `Error`, if the given `Iterable` is empty

Parameters

Name	Type	Description
`it`	`Iterable<_T_>`	the receiver of this operation
`comparator`	`BiPredicate<_T_,_T_>`	an optional comparing function which returns true if the second argument is larger than the first

Example

```
const numbers = [1, 3, 0, 5];
const maximum = max$(numbers);

console.log(maximum);
// => Logs '5'
```

safeMax$

Returns the largest element of an iterable.

See the note to `max$`.

Returns: `MaybeType<_T_>`

Parameters

Name	Type	Description
`it`	`Iterable<_T_>`	the receiver of this operation
`comparator`	`BiPredicate<_T_,_T_>`	an optional comparing function which returns true if the second argument is larger than the first

Example

```
const numbers = [1, 3, 0, 5];
const maybeMax = safeMax$(numbers);

maybeMax
 (_ => console.log("iterable was empty, no max!")
 (x => console.log(x));
// => Logs '5'
```

min$

Returns the smallest element of a **non-empty** iterable

See the Note to `max$`.

throws: `Error`, if the given `Iterable` is empty

Returns: MaybeType<_T_>

Parameters

Name	Type	Description
`it`	`Iterable<_T_>`	the receiver of this operation
`comparator`	`BiPredicate<_T_,_T_>`	an optional comparing function which returns true if the first argument is smaller than the second

Example

```
const numbers = [1, 3, 0, 5];
const minimum = min$(numbers);

console.log(minimum);
// => Logs '0'
```

safeMin$

Returns the smallest element of an iterable.

See the Note to `max$`.

Returns: `MaybeType<_T_>`

Parameters

Name	Type	Description
`it`	`Iterable<_T_>`	The receiver of this operator.
`comparator`	`BiPredicate<_T_,_T_>`	An optional comparing function which returns true if the first argument is smaller than the second.

Example

```
const numbers  = [0, 1, 2, 3];
const maybeMin = safeMin$(numbers);

maybeMin
 (_ => console.log("iterable was empty, no min!")
 (x => console.log(x));
// => Logs '0'
```

reduce$ / foldl$

Performs a reduction on the elements, using the provided start value and an accumulation function, and returns the reduced value.

Note: `foldl$` is an alias for `reduce$`

Returns: `_T_`

Parameters

Name	Type	Description
`accumulationFn`	`BiFunction<_T_, _U_, _T_>`	the reduction function
`start`	`_T_`	the first value
`it`	`Iterable<_T_>`	the receiver of this operation

Example

```
const number = [0, 1, 2, 3, 4, 5];
const res = foldl$((acc, cur) => acc + cur, 0)(numbers);

console.log(res);
// => Logs '15'
```

show

Transforms the passed iterable into a `String`.

Returns: `String`

Parameters

Name	Type	Description
`it`	`Iterable<_T_>`	the receiver of this operation
`maxValues`	`Number`	optional: the amount of elements that should be printed at most

Example

```
const numbers = [0, 1, 2, 3, 4, 5];
const text    = show(numbers, 3);

```

```
console.log(text);
// => Logs '[0,1,2]'
```

uncons

Removes the first element of this iterable.

Returns: `Pair<_T_, SequenceType<_T_>` the head and the tail as a pair

Parameters

Name	Type	Description
`iterable`	`Iterable<_T_>`	the receiver of this operation

Example

```
const numbers       = [0, 1, 2, 3, 4];
const [head, tail]  = uncons(numbers);

console.log("head:", head, "tail:", ...tail);
// => Logs 'head: 0 tail: 1 2 3 4'
```

B.4. The Prototype of the Sequence

Some functions are also available via the prototype of the sequence.

and

This is the same as `bind`.

Applies the given function to each element of the sequence and flattens it afterward. Note: This operation adds a monadic API to the `SequenceType`.

Returns: `SequenceType<_U_>`

Parameters

Name	Type	Description
`bindFn`	`<_U_>(bindFn: (_T_) => Iterable<_U_>)`	this function will be applied to each element of the iterable

Example

```
const numbers = Range(3);
const bindFn  = el => take(el)(repeat(el));
const result  = numbers.and(bindFn);

console.log(...result);
// => Logs '1, 2, 2, 3, 3, 3'
```

fmap

This is the same as `map`.

Transforms each element using the given function.

Returns: `SequenceType<_U_>`

Parameters

Name	Type	Description
`mapper`	`Functor<_U_, _T_>`	is applied on each element

Example

```
const numbers = Range(2);
const mapped  = numbers.fmap(el => el * 2);

console.log(...numbers);
// => Logs '0, 2, 4'
```

empty

This is the same as `nil`.

This functions returns a sequence containing no values.

Returns: `SequenceType<*>`

Example

```
const emptySequence = Range(3).empty();

console.log(...emptySequence);
// => Logs '' (nothing)
```

pure

This is the same as `PureSequence`.

Creates a sequence which contains just the given value.

Returns: `SequenceType<_T_>`

Parameters

Name	Type	Description
`value`	`_T_`	the single value

Example

```
const seq = Range(3).pure(1);

console.log(...seq);
// => Logs '1'
```

toString

This is the same as `show`.

Transforms this sequence into a `String`.

Returns: `String`

Parameters

Name	Type	Description
`maxValues`	`Number`	optional: the amount of elements that should be printed at most

Example

```
const numbers = Range(6);
const text    = range.toString(3);

console.log(text);
// => Logs '[0,1,2]'
```

pipe

This is the same as `pipe`.

Transforms this sequence using the passed operators.

Returns: `SequenceType<_T_> | *`

Parameters

Name	Type	Description
`transformers`	`SequenceOperation<*,*>`	the operators to apply

Example

```
const numbers = Range(5);
const piped   = numbers.pipe(
                  retainAll(n => n % 2 === 0),
                  map(n => 2*n),
                  drop(2)
                );

console.log(...piped);
// => Logs '0, 4, 8'
```

==

This is the same as eq$.

Checks the equality of this iterable with the given iterable.

Note: Two iterables are considered as equal if they contain or create the exactly same values in the same order. Use `["=="]` defined on the SequencePrototype to perform a comparison in a more readable form.

Returns: `Boolean`

Parameters

Name	Type	Description
`that`	`Iterable<_T_>`	the other (finite) iterable to compare with

Example

```
const numbers = [0, 1, 2, 3];
const range   = Range(3);
const result  = range ["=="] (numbers);

console.log(result);
// => Logs 'true'
```

C. User Test Results

To minimize linguistic misunderstandings (especially with text answers), we decided to conduct the questionnaire of the user test in the native language of the participants (German). This appendix, therefore, lists all questions and answers in the original language.

C.1. Questions about Sequences

Note: Initially, the Sequence library was called Sequence framework. The questions, therefore, use the old name.

Question 1

Mir war sofort klar, was die Parameter des Sequenz-Konstruktors bedeuten und wie ich sie definieren muss.

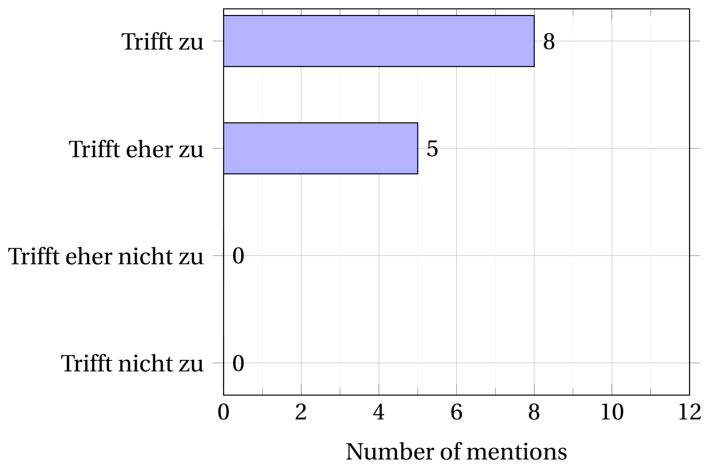


Figure C.1.: Answers to question 1

Question 2

Ich brauche Funktionen wie map, filter, take, reduce häufig wenn ich mit Listen, Arrays oder Ähnlichem arbeite.

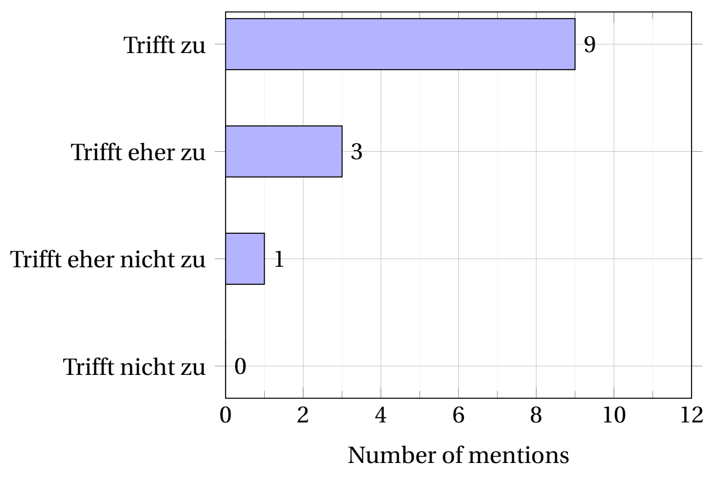


Figure C.2.: Answers to question 2

Question 3

Die Codebeispiele in der JSDoc von take, uncons usw. waren hilfreich.

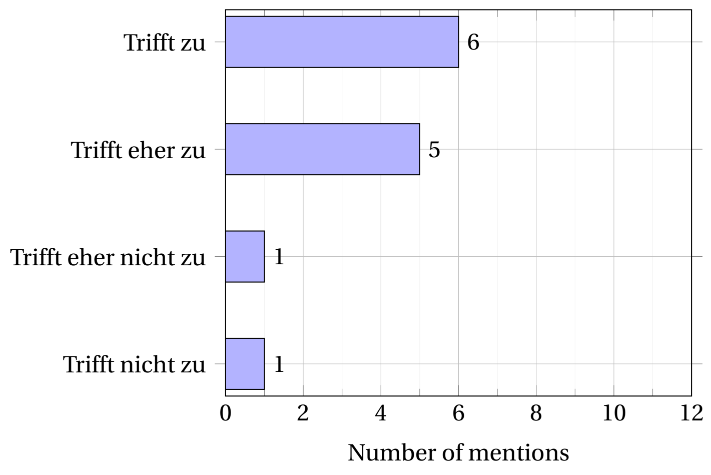


Figure C.3.: Answers to question 3

Question 4

Ich kann mir vorstellen, in einem zukünftigen Projekt das Sequenzframework zu verwenden.

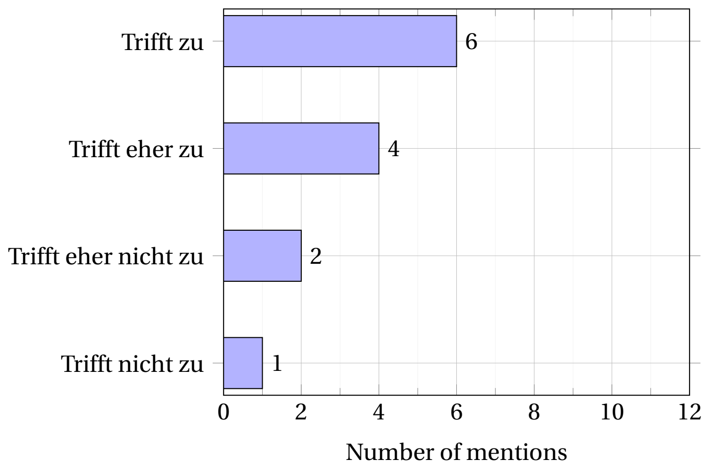


Figure C.4.: Answers to question 4

Question 5

Diese Funktionen brauche ich sonst noch oft im Zusammenhang mit Listen.

Note: This multi select question allows us to discover missing operators in the Sequence library.

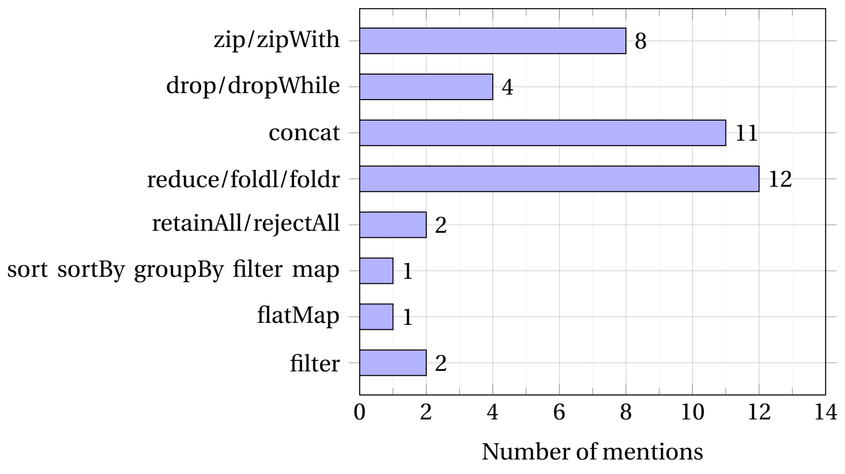


Figure C.5.: Answers to question 5

Question 6

Ich finde es sinnvoll, das Sequenzframework über einen Namen (im Test '_') zu importieren, um bessere IDE Unterstützung zu erhalten.

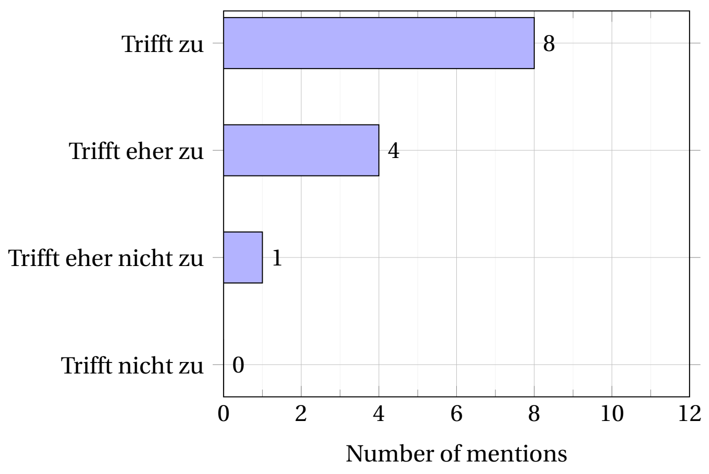


Figure C.6.: Answers to question 6

Question 7

Was gefiel Ihnen im Zusammenhang mit Sequenzen, was weniger?

Der funktionale Stil, aber das liegt bloss an persoenlicher Vorliebe. Prinzipiell eine gute Idee, aber die Funktionen sind so simpel, dass ich das lieber selber schreibe, als dafuer ein Framework zu lernen.
Sehr schnell verständlich wie es aufgebaut und somit zu verwenden ist. So machen auch Mathematische Funktionen schon fast Spass :D
Dank der Beschreibung ab Zeile 60 war mir sofort klar, was eine Sequenz ist und was man im Konstruktor angeben muss. Das Beispiel JSDoc ist mir erst aufgefallen, als ich mit der Aufgabe fertig war und den Fragebogen gestartet habe. Ein Beispiel wäre aber sicher von Anfang an hilfreich gewesen. Ich denke so ein SequenzFramework kann viel Arbeit abnehmen und auch den Code lesbarer machen. Allerdings hätte ich das Sequenzframework mit einem anderen Namen importiert anstatt mit _.
hat jetzt nicht wirklich den anreiz, da es auch mit vanilla js machbar ist und nicht wirklich schwer zu implementieren ist
Die Art wie mit diesen Sequenzen gearbeitet werden kann und die gebotene Funktionalität, gefällt mir gut. Hatte einzig etwas mühe, JSDocs in Verbindung mit dem Curring zu lesen. Die Beispiele waren da jeweils deutlich hilfricher, als der angezeigte Typ der Funktoin (z.B. bei _.map)
Mir gefiel den Einstieg und die Dokumentation. Ich habe eigentlich nichts was mir nicht so gefallen hat.
+Ähnlichkeit in der Anwendung zu Haskell und anderen funktionalen Sprachen
ist das effizient bzgl. Laufzeit und Speicher? Wie loggt man Zwischenstände?
einfach anzuwenden, aber benötigt Eingewöhnung beim Aufruf mit ()() [zb. _.map(slope(f)(x))(halves(h0))]
Tolle Sache. Funktionalität und Anwendung habe ich verstanden. Der Mathe Teil nicht ganz einfach zu verstehen, musste es mehrmals durchlesen um es zu verstehen. Ev gibt es noch ein einfacheres Beispiel.

Table C.1.: Answers to question 7

C.2. Questions about JINQ

Question 8

Es war einfach JINQ zu verwenden.

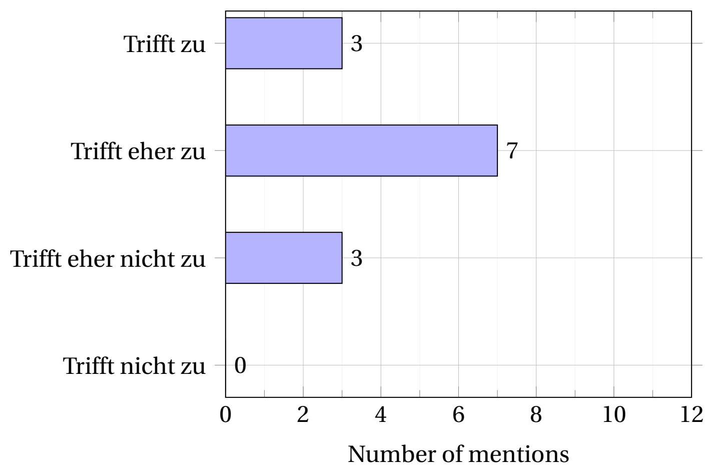


Figure C.7.: Answers to question 8

Question 9

Ich habe bereits oft mit ähnlichen Abstraktionen wie JINQ gearbeitet. (zB. LINQ).

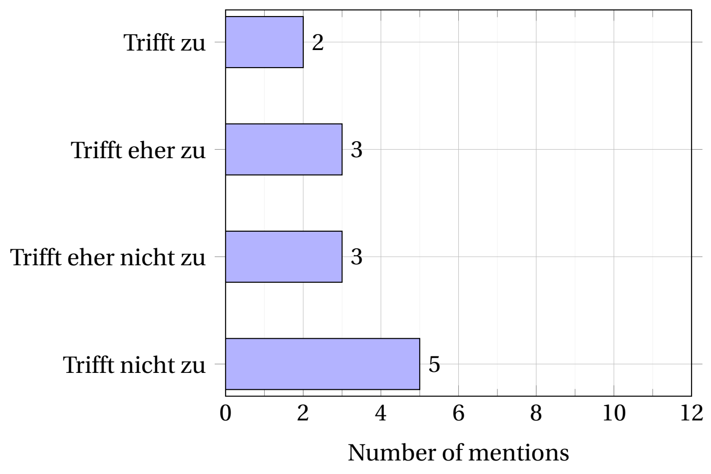


Figure C.8.: Answers to question 9

Question 10

Ich sehe Sinn darin, mittels JINQ JSON-Strukturen zu durchsuchen.

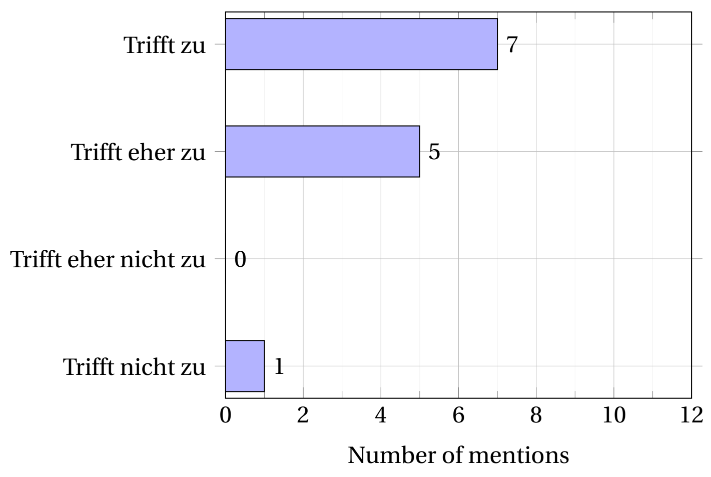


Figure C.9.: Answers to question 10

Question 11

Ich finde die Operationen von JINQ sind sinnvoll benannt.

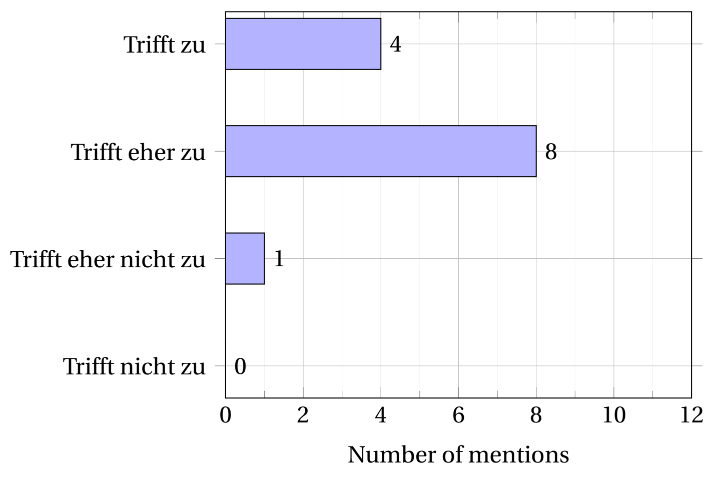


Figure C.10.: Answers to question 11

Question 12

JINQ könnte in einem zukünftigen Projekt ein Gewinn sein.

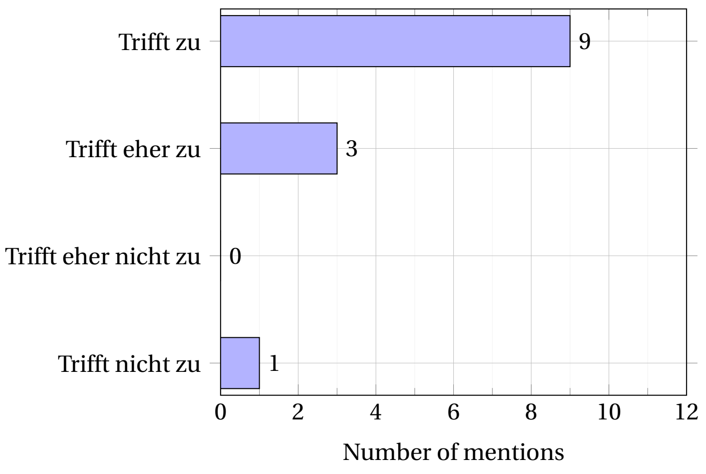


Figure C.11.: Answers to question 12

Question 13

Was gefiel Ihnen im Zusammenhang mit JINQ, was weniger?

Die Operationen from, select, where sind einleuchtend benennt und daher gut verständlich, da sie an SQL-Abfragen erinnern, wo die gleichen Begriffe verwendet werden.
from(JsonMonad(developers)) => könnte man JsonMonad nicht dirrekt ins from reinnehmen?
Aehnlich wie Sequenzen. Die Funktionalitaet von JINQ selbst ist einfach selbst umzusetzen, aber es ist schwierig sich in ein neues Framework einzubauen und unerwartete Fehler zu machen, deshalb wuerde ich sie lieber selbst umsetzen, damit ich sie nicht lernen muss.
Kannte ich vorher nicht, brauchte deshalb etwas Zeit um mich einzulesen. Hat man das Prinzip dann verstanden, ist es relativ einfach das gewünschte Ergebnis aus einer JSON-Struktur herauszufiltern.
Ich hatte vorher weder JINQ noch LINQ benutzt. Um JSON zu durchsuchen, ist es sicher sehr nützlich und ich würde es auch selbst verwenden. Hier hätte ich mir aussagekräftigere Beispiele gewünscht, z.B. ein JSON als Beispiel und dann wie man JINQ verwendet und dann das Ergebnis.
dass .include() zur string comparison einen error wirft im where war sehr anstrengend und verhindert, dass man die richtigen methoden von js verwenden kann -> unsauberer code. Ich fand die Ähnlichkeit zu der csharp implementierung ganz cool, aber die types und die errors sind halt sehr abstrakt. Ich persöhnlich finde es sowieso sinnvoller, wenn man mit maps arbeitet oder viel json parsen muss, würde ich eher richtung danfoJS gehen, da ich den Pandas Syntax von Python schon gut kenne.
Mir gefallen solche ansätze sehr. Gute Umsetzung!
Da ich bereits mit LINQ vertraut bin, konnte ich mein vorhandenes Wissen anwenden. Vielleicht sollte die Operation "result()" besser als "get()" oder "toMonad" bezeichnet werden. Ich kann dies nicht genau erklären, aber mir wäre dann klarer gewesen, dass ich den Vorgang abgeschlossen habe. Möglicherweise habe ich noch zu sehr an andere Frameworks gedacht, in denen man häufig "toList" oder Ähnliches verwendet.
- vielleicht liegt es daran, dass ich LINQ noch nie genutzt habe, aber ich finde inside() nicht intuitiv, obwohl die Funktionalität bekannt ist und ich finde pairWith() könnte beispielsweise join() heissen (angelehnt an SQL oder pandas in Python)
Mehr Beispiele wären angenehm um reinzukommen.
Der Code wirkt mit der JINQ Verwendung schön strukturiert. Typunterstützung soweit vorhanden. Die Fehlermeldungen sind nicht immer sofort klar.

Table C.2.: Answers to question 13

nach einer kleinen Eingewöhnung gut zu handhaben, anfangs noch Probleme mit den Promice wegen den null-Einträgen
Tolle sache. Gut verständlicher Code. Monade Thema eher schwer zu verstehen. Anwendung allerdings super.

Table C.3.: Answers to question 13 (continuation)

C.3. General Questions

Question 14

Ich habe Erfahrung in der funktionalen Programmierung.

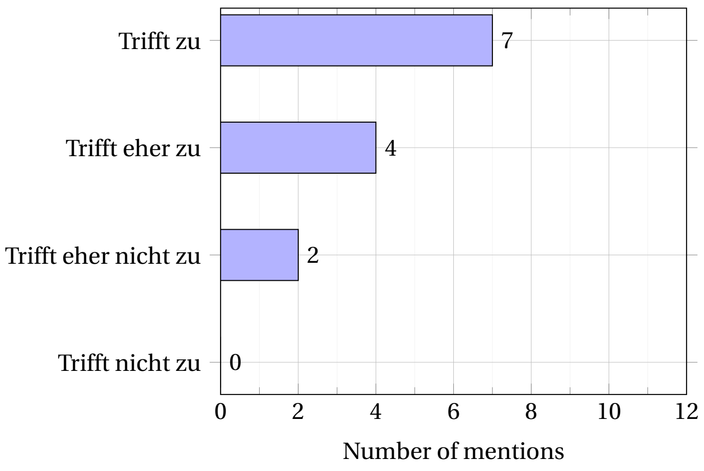


Figure C.12.: Answers to question 14

Question 15

Für diesen Test nutzte ich folgende IDE.

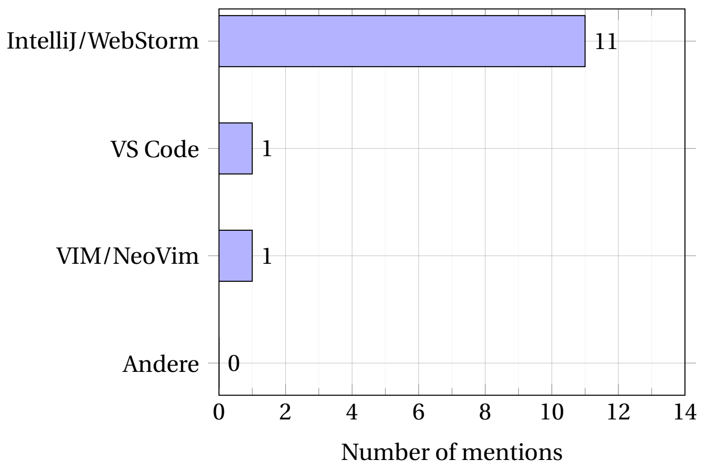


Figure C.13.: Answers to question 15

Question 16

Meine IDE hat mich während dem Usertest sinnvoll unterstützt. (Code Completion, Inline Docs, Code Navigation, Fehler/Warnungen)

Note: This question allows us to determine if it is worth the effort to write JSDoc.

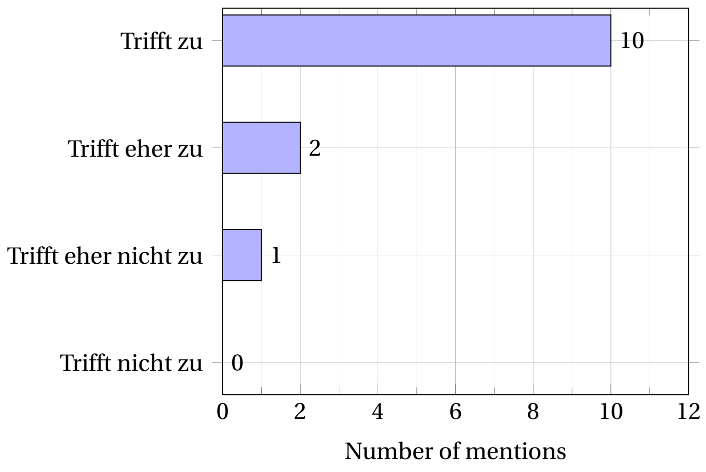


Figure C.14.: Answers to question 16

Question 17

Das möchte ich sonst noch sagen:

Sehr detailliert und gut dokumentiert!
coole Idee, spannende Umsetzung. Die erste Aufgabe war recht anspruchsvoll ohne Repetition in eana.
Bei Fragen stehe ich gerne weiterhin zur Verfuegung.
Ich finde sehr sinvoll was ihr gemacht hab. Ich habe etwa 1.5 Stunden aufgewendet und konnte Einblicke in etwas bekommen, was ich so noch nicht kannte, danke dafür!
Ich finde die Idee gut, man müsste die Errors besser machen können damit es Anklang finden könnte. Wenn ich anstatt "C++" einen Output von f(x) => g _f(x) (so in der Art wars), ist das debuggen sehr schwer.
Vielen Dank für die Möglichkeit, dabei zu sein! Es war super, mein Gehirn mal wieder ein bisschen zu quälen, äh, ich meine natürlich anzustrengen, durch die Tests.
+Spannendes Projekt, dass praktische Funktionalität bringt, die man sich beispielsweise auch durch Lodash holen kann, aber vor allem erlaubt eure Implementation Entwicklern ihre eigenen Funktionen hinzuzufügen
Tolle Arbeit!
Cool!

Table C.4.: Answers to question 17

D. Project Description

Fachhochschule Nordwestschweiz

App Web | SE RE

23FS_IMVS18: Erweiterung der Funktionalen Standard Library für das Web UI Toolkit "Kolibri"

Betreuer:	Dierk König		
		Priorität 1	**Priorität 2**
Arbeitsumfang:		P6 (360h pro Student)	---
Teamgrösse:		2er Team	---
Sprachen:	Deutsch oder Englisch		
Studiengang:	Informatik		

Ausgangslage
An der FHNW bauen wir das Web UI Toolkit "Kolibri" [1] mit Beiträgen der Studierenden. Die Library umfasst funktionale Abstraktionen wie sie etwa in der Haskell Standard Library genutzt werden, Test Unterstützung, Utilities für den sicheren Umgang mit HTML/DOM, aber auch entwicklungsnahe Aspekte wie effizientes und flexibles Logging. Im Vorgänger P5 Projekt wurden die ersten wichtigen Erweiterungen erfolgreich vorgenommen.

Ziel der Arbeit
In diesem P6 Nachfolgeprojekt geht es darum, die Library weiter auszubauen. Dabei steht die erwiesene Nützlichkeit der Erweiterungen für die tägliche Arbeit des Applikationsprogrammierers im Vordergrund. Die Studierenden müssen aus der Informatik bekannte Abstraktionen, Entwurfsmuster und Vorgehensweisen auf ihren möglichen Nutzen für das Web UI Toolkit untersuchen und mit Produktionsqualität umsetzen mit dem Ziel das Projektergebnis in die Kolibri Standard Library zu integrieren.

Kolibri Logo

Problemstellung
Es sind eine Menge an möglichen Erweiterungen zu untersuchen. Dazu gehören zum Beispiel die persistenten Datentypen, Abstraktionen aus der Haskell Standardlibrary, das Java Streams API und der Ansatz der language-integrated queries (LINQ). Zu Untersuchung gehören das Verständnis der Abstraktionen, der Versuch einer prototypischen Umsetzung in typisiertem JavaScript und die Bewertung der Nützlichkeit für den Applikationsprogrammierer. Wo die Nützlichkeit erwiesen wurde ist eine voll typisierte Umsetzung in Produktionsqualität inklusive Dokumentation, automatisierten Tests und einer Demo-Applikation zu erstellen.

Technologien/Fachliche Schwerpunkte/Referenzen
JavaScript, Design Patterns, Funktionale Programmierung, Lambda Kalkül, LINQ

Referenzen
[1] https://webengineering-fhnw.github.io/Kolibri/
[2] Simon Peyton Jones, A Taste of Haskell, https://www.youtube.com/watch?v=jLj1QV11o9g&t=224s

Bemerkung
Zwingende Voraussetzung ist der erfolgreiche Besuch des Moduls "Web Programming".
Weitere Kenntnisse der Funktionalen Programmierung und des Moduls "Web Clients" sind wünschenswert.
Im Erfolgsfall wird das Projektergebnis ein grundlegender Bestandteil des open-source toolkits "Kolilbri".

Dieses Projekt ist für Andri Wild und Tobias Wyss reserviert.

Studiengang Informatik/IMVS/Studierendenprojekte 23FS