Discovering Haskell

Documentation for the module “Project 1”

Aaron Grand	Elias Ingold	Amar Tabakovic
`grana5`	`ingoe3`	`tabaa1`

June 16, 2023

Abstract

Haskell is a purely functional programming language and is very different from other contemporary programming languages such as Java and C++. This document aims to give an unfamiliar reader an overview of Haskell and functional programming in general by covering the most important aspects of Haskell, accompanied by many use cases and example programs.

Contents

1 Introduction **4**
- 1.1 Imperative Versus Functional Programming 4
- 1.2 Goals of the Project 4
- 1.3 Literature 5
- 1.4 Document Structure 5

2 Basics of Haskell **7**
- 2.1 Basic Workflow 7
- 2.2 Basic Types 8
  - 2.2.1 Numerical types 8
  - 2.2.2 Booleans 8
  - 2.2.3 Characters and Strings 8
  - 2.2.4 Tuples 8
  - 2.2.5 Lists 9
  - 2.2.6 Type Signatures 9
- 2.3 Basic Operations 10
  - 2.3.1 Arithmetic Operations 10
  - 2.3.2 Comparison Operations 10
  - 2.3.3 Logic Operations 11
  - 2.3.4 List Operations 11
- 2.4 Functions 12
  - 2.4.1 Basic Function Definition 12
  - 2.4.2 Basic Function Application 12
- 2.5 Branching 14

3 Recursion and Pattern Matching **15**
- 3.1 Recursion 15
- 3.2 Pattern Matching 16
- 3.3 Folding 16
  - 3.3.1 `foldr` 17
  - 3.3.2 `foldl` 17
- 3.4 Some Use Cases and Examples 18
  - 3.4.1 Concatenation of Lists 19

3.4.2 Mergesort 20
3.4.3 Quicksort 22

4 Advanced Function Concepts **26**
4.1 Currying 26
4.2 Higher-order Functions 26
4.3 Anonymous Functions 27

5 Lazy Evaluation **28**
5.1 What is Lazy Evaluation? 28

6 Advanced Type Concepts **30**
6.1 Custom Types 30
6.2 Type Classes 30
6.3 Polymorphic Types 31
6.4 Recursive Types 32
6.4.1 Example of a Binary Tree 32
6.4.2 Example of a General Tree 33

7 Game of Nim **35**
7.1 Introduction 35
7.2 Implementation 35
7.2.1 Winning Condition 36
7.2.2 Data Representation and Conversion 36
7.2.3 Determining Winning and Losing Game States 37
7.2.4 Gameplay 37
7.2.5 Player and Computer Turns 38
7.2.6 Main Game Loop 39
7.2.7 Functional Aspects and Side Effects 39
7.3 Conclusion 40

8 Unification **41**
8.1 Introduction 41
8.1.1 What is Unification? 41
8.1.2 Basic Examples 41
8.2 Parsing 43
8.2.1 Grammar for Terms 43
8.2.2 `Parser` Data Type 43
8.2.3 Basic Parser Functions 44
8.2.4 `Term` Data Type 50
8.2.5 Parser Functions for Unification Terms 50
8.2.6 Examples of Parsing 54
8.2.7 Functional Aspects of the Parser 55
8.3 Unification Algorithm 56

8.3.1 Description of the Algorithm 56
8.3.2 Substitution Definition 57
8.3.3 Main Unification Function 57
8.3.4 Unification With a Variable 58
8.3.5 Unification With a Variable and a Compound 58
8.3.6 Substitution Application 59
8.3.7 Unification of List of Terms 59
8.3.8 Functional Aspects of the Algorithm 59
8.4 Conclusion 60

9 Project Management and Organisation **61**
9.1 Overview 61
9.2 Sprints 61
9.3 Roles 62
9.4 Conclusion 62

10 Conclusion **64**
10.1 Looking Back 64
10.2 Looking Forward 64

Bibliography **67**

Chapter 1

Introduction

1.1 Imperative Versus Functional Programming

Many different programming paradigms exist. "Normal" languages such as Java, C and C++ are often called *imperative programming languages*. Imperative programming languages consist of step-by-step instructions for the machine (or virtual machine) to execute and depend on the notion of mutable state. In other terms, imperative programming languages allow for variables to be modified after their definition. This mutability of variables causes side effects in a program, which can then cause unwanted behavior or even bugs.

Another family of languages are the *functional programming languages*, which in turn do not allow for mutable state. After defintion, a variable is immutable and its value cannot be changed, which eliminates the occurrence of side effects. Instead of step-by-step instructions, functional programming languages consist of mathematical functions: a function computes the same output for the same input without causing any side effects.

One such language is **Haskell**, a purely functional programming language. Haskell is named after mathematician Haskell Curry (1900 – 1982) and has been in development since 1987. The details of Haskell are going to be described later on in this documentation.

1.2 Goals of the Project

Discovering Haskell The first goal of this project is to explore the possibilities of Haskell and applying Haskell to many different scenarios. For this, some important aspects of Haskell such as recursion and pattern matching are described and for each important aspect, some examples are given. Additionally, we compare the functional implementations of our examples to imeplementations in imperative languages without using functional constructs, such as Java Streams, and show how they differ.

This document can be seen as a collection of knowledge we gained throughout a semester, with the goal to be useful to software engineers who do not have experience in functional programming and wish to gain some insights.

Learning and Applying Scrum The second goal of this project is to learn and apply Scrum and best practices in agile project management. This means that this project is conducted using Scrum as the project management method, with meetings being held with our project adviser every 1 - 2 weeks.

1.3 Literature

The main literature we used to learn Haskell and for looking up important information are the following:

- The course material of the module *Functional Programming in Java and Kotlin* taught by Dr. Olivier Biberstein at the Bern University of Applied Sciences [1].
- *Haskell: The Craft of Functional Programming* by Simon Thompson [2].
- *Programming in Haskell (First Edition)* by Graham Hutton [3].

1.4 Document Structure

This document is structured as follows:

- Chapter 2 is a longer chapter that gives a general introduction to the Haskell programming language. The most important basic concepts of the language are presented, some of which include basic types and operations, basics of functions, branching and more.
- Chapter 3 discusses *recursion* and *pattern matching* and their importance in Haskell and functional programming in general. Topics include an introduction to the concept of recursion, defining recursive functions and a few examples and comparisons which demonstrate why recursion and pattern matching is essential in Haskell.
- Chapter 4 discusses advanced concepts of functions, such as *currying*, *higher order functions* and *anonymous functions*.
- Chapter 5 goes into the topic of *lazy evaluation*, the concept of only computing values when really needed. One important application of this concept is that infinite lists can be constructed.

- Chapter 6 discusses some advanced topics of Haskell's type system. Topics include *type inference*, *type classes*, *polymorphic types* and more.
- Chapter 7 describes the first larger Haskell project, which is the *Game of Nim*, a mathematical two-player strategy game.
- Chapter 8 describes the second larger Haskell project, which is the implementation of *Unification*, a procedure used in first-order logic and type inference. The project consists of a *parser* that parses strings into unification terms and of the implementation of the unification algorithm.
- Chapter 9 goes into the organisation and project management aspects of this project, such as how we organized ourselves using Scrum and Agile.
- Chapter 10 describes the most important results and gives a short reflexion of the project.

Chapter 2

Basics of Haskell

2.1 Basic Workflow

Haskell programs are saved in `.hs` files and compiled using one of Haskell's few compilers. One of the most popular compilers is the *Glasgow Haskell Compiler (GHC)*, developed at the University of Glasgow [4]. GHC also comes with an interactive programming environment, called *GHCi* (the "i" stands for "interactive"). The version of GHC used in this project is version 8.10.7.

With GHCi, a programmer can enter Haskell expressions directly into the terminal and the expression gets evaluated and displayed to the terminal in real time. Another positive aspect of GHCi is that types of expressions can be determined, as shown in the example below:

```
1 Prelude> :t map
2 map :: (a -> b) -> [a] -> [b]
```

Listing 2.1: GHCi environment with the output of the command to show the type of the function `map`.

What this *type signature* exactly means will be explained shortly in the section "Type Signatures". GHCi supports other commands which can be shown using the command `:?`.

GHC can also be used normally as a compiler, i.e. running a command to compile a Haskell program. The output of the compiler is a binary file, which can then be executed.

For most programs in this project, the following approach can be taken:

1. In the command line, navigate to the `src/` directory and into the corresponding chapter.

2. Run `ghci` from the command line.

3. Enter `:load` followed by the `.hs` source file that you want to run and press enter (e.g. `:load basic-functions.hs`).
4. Enter any function or expression you want to execute and press enter.

2.2 Basic Types

2.2.1 Numerical types

Integers Integers are represented with multiple types in Haskell, such as `Int` for 32-bit or 64-bit integers and `Integer` for arbitrary sized integers.

Floating point numbers Like integers, floating point numbers are represented with multiple types in Haskell, such as `Float` for single-precision floating point numbers, `Double` for double-precision floating point numbers and `Rational` for rational numbers.

2.2.2 Booleans

Haskell supports the basic boolean type `Bool` with values `True` and `False`. The operations on booleans are described in the section "Logic Operations".

2.2.3 Characters and Strings

Haskell supports characters as the primitive type `Char`. A character consists of a single symbol and is denoted by single quotes `''`.

Basic strings are simply lists of characters. This means that operations on lists can be applied to strings. A concrete string type `String` is also defined with its own operations [5].

2.2.4 Tuples

Haskell supports tuples, which are similar to tuples from mathematics. They are denoted with parantheses `()` and their elements are separated by commas. The order of the elements is important. The elements of a tuple in Haskell can consist of many different types. The following example shows a tuple with lists of booleans in the first position and a boolean in the second position:

```
1 myTuple = ([True, False, True], True)
```

Listing 2.2: Example of a tuple with a list of booleans in its first position and a boolean in its second position.

Tuples can also consist of other tuples. The elements of a 2-tuple can be accessed either through the `fst` (first) function or the `snd` (second). The elements of a tuple of any size can be accessed using *tuple destructuring*:

```
(bs, b) = myTuple
```

Listing 2.3: Example of tuple destructuring with the above turple. `bs` and `b` can now be used as values.

2.2.5 Lists

Lists in Haskell are used to store multiple elements of a single type. Unlike arrays in imperative programming languages, lists have a dynamic size and differ strongly in their construction, which will be elaborated shortly.

Lists can be defined directly using brackets `[]` with their elements inside the brackets separated by commas, as shown in the following example:

```
boolList = [True, False, True]
```

Listing 2.4: Example of a list containing boolean values.

Lists are constructed using the *cons* operator, which is denoted by a colon `:`. The cons operator simply prepends an element to a list. The construction happens element-wise, beginning with the empty list, as shown below:

```
boolList = True : (False : (True : []))
         = True : (False : [True])
         = True : [False, True]
         = [True, False, True]
```

Listing 2.5: Example of the construction of the list from the above example. The listing does not contain valid Haskell syntax and is only used to demonstrate each step of the construction.

2.2.6 Type Signatures

Type signatures are used in Haskell to indicate what type a value i.e. variable has. A few basic examples are shown below:

```
23 :: Num p => p
False :: Bool
["Aaron", "Amar", "Elias"] :: [[Char]]
```

Listing 2.6: Three examples of type signatures.

The first signature signifies that `23` is a value of type `p` that belongs to the *type class* `Num`. The concept of type classes is also going to be discussed later on in the chapter "Advanced Types".

The second signature signifies that `False` is a value of type `Bool`.

The third signature signifies that `["Aaron", "Amar", "Elias"]` is a list of character lists.

Types often do not have to be annotated, but it is good practice to designate them, because then it is clear what kind of values a variable really can accept [6].

2.3 Basic Operations

2.3.1 Arithmetic Operations

Haskell supports the basic arithmetic operations addition, subtraction, multiplication and division. The addition, subtraction and multiplication are all denoted with their standard symbols (i.e. `+` , `-`, `*`) The division operation is a bit more special however, since it exists in two variants: one variant is for floating point numbers, denoted as `/`, and one variant is for whole numbers, denoted as `` `div` ``.

Exponentiation is supported through the `^` operator. The operator precedence is as follows: exponentiation has precedence over multiplication and division, which has precedence over addition and subtraction. The following example demonstrates the operator precedence:

```
n = 2 + 5 * 7 ^ 3
n' = 2 + (5 * (7 ^ 3))
b = n == n' -- evaluates to True
```

Listing 2.7: Basic example of operator precedence.

For modular arithmetic, Haskell offers the `` `mod` `` function, which calculates the remainder of the division of two numbers.

2.3.2 Comparison Operations

The comparison operations work just like in other programming languages.

Greater than The greater than comparison is denoted by `>`. The greater equals comparison is denoted by `>=`.

Less than The less than comparison is denoted by `<`. The less equals comparison is denoted by `<=`.

Equals The equality comparison is denoted by `==`.

Not equals The inequality comparison is denoted by `/=`.

2.3.3 Logic Operations

Haskell supports the following logic operations:

Logical conjunction Similar to other programming languages, the logical conjunction (i.e. logical AND) operation is denoted by double ampersands `&&`.

Logical disjunction Similar to other programming languages, the logical disjunction (i.e. logical OR) operation is denoted by double pipes `||`.

Logical negation Logical negation in Haskell is denoted by the `not` keyword.

2.3.4 List Operations

Haskell contains many operations on lists, some of which include:

- Taking the first element of a list → `head`
- Taking the last element of a list → `last`
- Removing the first element of a list → `tail`
- Taking the first n elements of a list → `take`
- Removing the first n elements of a list → `drop`
- Reversing a list → `reverse`
- Getting the length of a list → `length`
- Appending two lists → `++`

and more. An exhaustive list of the basic list manipulation functions can be found in the Haskell documentation [7].

2.4 Functions

2.4.1 Basic Function Definition

Haskell programs consist mainly of functions. A function can take one or more arguments and always compute the same output value for the same input. For example, a function to add two numbers can be defined as follows:

```
1 f x = 2 * x + 1
```

Listing 2.8: Basic function `f` to double a number and add one to it.

The above function declaration indicates that `f` is a function that takes an argument `x` and evaluates it to the double of `x` plus one. The syntax of Haskell takes some inspiration from mathematics, which is especially apparent in this example when comparing it to the equivalent linear function:

$$f(x) = 2x + 1$$

How does the compiler know that the argument of the function is indeed a number? It determines this through a process called *type inference*. This aspect of Haskell will be discussed later on in this documentation. The important part right now is that just like variables, functions have a type as well. The above function `f` for instance has the type `f :: Num a => a -> a`. That is, `f` is a function that takes in a value of the type `a` that belongs to the type class `Num` and maps it to another value of type `a`.

Another example shows a function `myAdd` that takes two arguments instead of just one:

```
1 myAdd x y = x + y
```

Listing 2.9: Function `myAdd` that returns the sum of two numbers.

In fact, the above declaration is a bit more than what was just described, but the respective topic (namely *currying*) will also be discussed later on in this documentation.

2.4.2 Basic Function Application

Functions are applied to their arguments by writing the function name first and its arguments afterwards, separated by spaces:

```
1 f 5 -- evaluates to 11
2 myAdd 2 3 -- evaluates to 5
```

Mathematics	Haskell
$f(x)$	`f x`
$f(x, y)$	`f x y`
$f(g(x))$	`f (g x)`
$f(x, g(y))$	`f x (g y)`
$f(x)g(x)$	`f x * g x`

Table 2.1: Examples of function applications from mathematics and their Haskell equivalent. The functions f and g are here arbitrary functions.

```
3 take 2 [1, 2, 3, 4, 5] -- evaluates to [1, 2]
```

Listing 2.10: Examples of function application.

An important thing to keep in mind when using functions is that function application has the highest precedence. For example, when using the above function `myAdd`, the following function application would be invalid:

```
1 -- invalid function application
2 myAdd 1 + 3 2 -- equivalent to (myAdd 1 +) 3 2
3 -- valid function application
4 myAdd (1 + 3) 2 --evaluates to 6
```

Listing 2.11: Example of applying `myAdd` incorrectly and correctly.

The following table taken directly from [3] shows a few examples of function applications in mathematics and in Haskell:

Functions can also be composed with the composition operator `.`:

```
1 -- function composition with parantheses
2 f' x = f (f x)
3 -- function composition with composition operator
4 f'' x = (f . f) x
```

Listing 2.12: Examples of function composition.

The important thing to note in the above example is that parantheses are required in the composition of the first line, because otherwise it could be interpreted that `f` takes two arguments, the first being a function and the second a number.

The main rule of function composition is that if the first function is denoted by $f : A \to B$ and the second function is denoted by $b : B \to C$, then the function composition must be of the type $(f \circ g) : A \to C$. Here

is once again the parallel between Haskell and the mathematical syntax clear (i.e. the similarity between the mathematical composition $\circ$ and the composition operator `.`).

2.5 Branching

Haskell supports basic branching with the `if` keyword. Below is an example that shows how the keyword can be used.

```
1 isEvenStr n = if even n then "even" else "odd"
```

Listing 2.13: Example of a function `isEvenStr` with branching. If the number that the function is applied to is even, it returns the string `"even"`, otherwise `"odd"`.

Another type of branching mechanism in Haskell are the *guards*, which are equivalent to *if-else* chains. The expressions are evaluated one after the other and the optional `otherwise` expression handles all the other cases. The main advantage of guards is the improved readability.

```
1 isEvenStr' n
2   | even n = "even"
3   | otherwise = "odd"
```

Listing 2.14: The function `isEvenStr` implemented with guards.

Chapter 3

Recursion and Pattern Matching

3.1 Recursion

Recursion is a powerful concept that enables elegant solutions for many problems. It is used extensively in Haskell and other functional programming languages for many different purposes and allows for concise code. Recursion stems from mathematics and a recursive function can be simply defined as a function that is applies itself somewhere in its right-hand side. When defining recursive functions, it is important to keep in mind that they must have at least one base case. Otherwise, they simply continue computing infinitely.

A basic example for recursion is the Fibonacci sequence:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$$

The formal definition of a function F that evaluates to the n-th number from the Fibonacci sequence in is as follows:

$$F(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if n} = 1 \\ F(n-1) + F(n-2) & \text{otherwise} \end{cases}$$

In Haskell, the same can be done with recursion:

```
1 fib n
2   | n == 0 = 0
3   | n == 1 = 1
4   | otherwise = fib (n - 1) + fib (n - 2)
```

Listing 3.1: Example of a function `fib` that computes the `n`-th Fibonacci number.

One thing to note is that this definition of the Fibonacci function uses binary recursion, i.e. the recursive function call happens twice in its definition. This leads to a major problem of this definition of the Fibonacci function: It has an exponential runtime complexity.

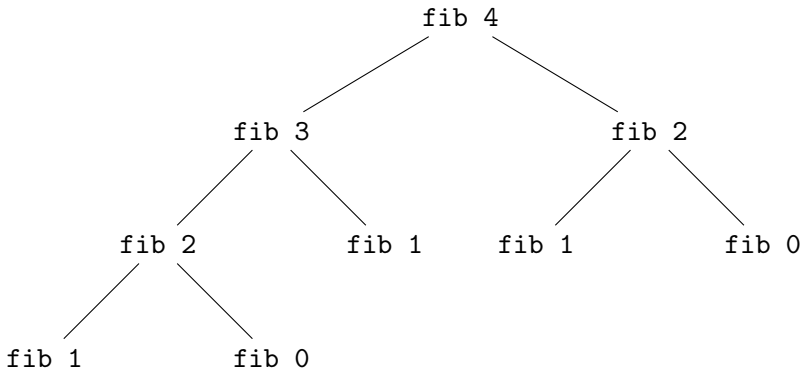

Figure 3.1: Tree representing the recursive function calls of `fib 4`.

For more optimal solutions of computing the Fibonacci sequence using more advanced methods, we refer to the Haskell wiki [8].

3.2 Pattern Matching

Pattern matching is a powerful concept in functional programming that allows for functions with certain defined arguments to be defined in a simple manner. Pattern-matched functions contain multiple function definitions for different patterns. Patterns are evaluated from top to bottom and must be defined exhaustively (i.e. for all possible inputs).

As a basic example, we take the Fibonacci function `fib` from the previous section and define it now using pattern matching instead of using guards:

```
1 fib' 0 = 0
2 fib' 1 = 1
3 fib' n = fib' (n - 1) + fib' (n - 2)
```

Listing 3.2: Function `fib'` that computes the `n`-th Fibonacci number defined with pattern matching.

3.3 Folding

Folding is a concept that can be used for common recursive patterns on lists. In Haskell, the basic folding functions are `foldr` and `foldl` for folding right

and left respectively. Folding is closely related to the concept of reduction (i.e. `reduce`) in other programming languages that support functional concepts, such as JavaScript. This relation will be clear when we show what `foldr` and `foldl` exactly do.

3.3.1 `foldr`

The function `foldr` can be best explained using a concrete introductory example: If we want to sum up all values in a list, we can do this with pattern matching as follows:

```
1 mySum [] = 0
2 mySum (x : xs) = x + mySum xs
```

Listing 3.3: Function `mySum` that sums up all values in a list using pattern matching.

With `foldr`, we can rewrite this function into another function `mySumFoldr` that looks like this:

```
1 mySumFoldr xs = foldr (+) 0 xs
```

Listing 3.4: Function `mySumFoldr` that sums up all values in a list using `foldr`.

In other terms, `foldr` continuously applies a function to values of a list and does this with a right-associativity (hence fold *right*) until an initial value is reached. The function that is applied to the elements is passed to `foldr` as the first argument and the initial value is passed as the second argument. With the above `mySum'` example, a list of numbers would get evaluated as follows:

```
1 foldr (+) 0 [8,3,4,6,7]
2 -- evaluates to 8 + (3 + (4 + (6 + (7 + 0))))
```

Listing 3.5: Demonstration of the `foldr` function that sums up numbers in a list.

3.3.2 `foldl`

The function `foldl` works similarly to `foldr`, but instead associates to the left. If we take our previous example of `mySum`, we can rewrite it into a

function `mySum'` such that it associates to the left instead:

```
1  mySum' xs = mySum'' 0 xs
2  where
3    mySum'' v [] = v
4    mySum'' v (x : xs) = mySum'' (v + x) xs
```

Listing 3.6: Function `mySum'` that sums up all values in a list using an auxiliary function `mySum''`.

The above function definition states that `mySum'` is a function that takes a list of numbers and applies the function `mySum''` to the list and the number zero. The function `mySum''` on the other hand takes an initial numerical value `v` and a list as its arguments. This value `v` also acts as the accumulator, i.e. we accumulate the values from the list to `v`. If the function is applied to an empty list, the accumulator value `v` simply gets returned with the final result. Otherwise, the head of the list is taken, added to the accumulator value and we recursively call `mySum''` again with the new value and the tail of the list.

We can rewrite this now with `foldl`:

```
1 mySumFoldl xs = foldl (+) 0 xs
```

Listing 3.7: Function `mySumFoldl` that sums up all values in a list using `foldl`.

The function `foldl` works exactly the same as `foldr` except for the fact that it associates to the left instead of the right. If we take the list from the `foldr` example and apply `foldl` to it, it gets computed as follows:

```
1 foldl (+) 0 [8,3,4,6,7]
2 -- evaluates to ((((0 + 8) + 3) + 4) + 6) + 7
```

Listing 3.8: Demonstration of the `foldl` function that sums up numbers in a list.

3.4 Some Use Cases and Examples

As mentioned previously, recursion is used (almost) everywhere in Haskell since there are no looping constructs commonly found in imperative languages. It can be used for the most basic of tasks, but also for more complex

problems, some of which are going to be demonstrated below.

3.4.1 Concatenation of Lists

For a first example, let us consider the use case of concatenating all elements of a list into a single list. For example, the list of lists consisting of `[[1,3,5],[2,7],[9,4,8]]` concatenated results in `[1,3,5,2,7,9,4,8]`. Such an implementation in Haskell could be:

```
1 myCat :: [[a]] -> [a]
2 myCat [] = []
3 myCat (xs : xss) = xs ++ myCat xss
```

Listing 3.9: Example of a Haskell function `myCat` that concatenates all elements in a list into a single list.

We can simplify the definition even further by using the `foldr` function:

```
1 myCat' :: [[a]] -> [a]
2 myCat' xs = foldr (++) [] xs
```

Listing 3.10: The function `myCat` rewritten into `myCat'` with `foldr`.

The further simplification would be to simply use the built-in library function `concat`.

If we compare our Haskell implementation with an implementation in Java, we get the following:

```
1 public static <T> List<T> myCat(List<List<T>> lists) {
2     List<T> result = new ArrayList<>();
3
4     for (List<T> list : lists) {
5         result.addAll(list);
6     }
7
8     return result;
9 }
```

Listing 3.11: Example of a Java method `myCat()` that concatenates all elements in a list into a single list.

Here we already see some minor differences between these two implementations:

- The Haskell implementation has a simpler type definition. The type of `myCat` is `myCat :: [[a]] -> [a]`, which means that it is a function that maps from a list of lists containing elements of any type `a` to a list that contain elements of the same type `a`. This is a very simple and concise type definition. The Java implementation on the other hand uses the concept of generics. It states something similar as the Haskell definition, but the type parameter `T` has to be additionally specified and used in the function body for variable definitions.
- The Haskell implementation has a more concise "function body". It simply consists of the base case (i.e. the function is applied to an empty list) and the recursive case that takes the first element `xs` of the list `xss` and concatenates the result of `myCat xss`. Here we can see a minor advantage of pattern matching. If the empty list gets matched, nothing else happens, it simply gets returned. Nothing else needs to be done.

3.4.2 Mergesort

Mergesort is a sorting algorithm that sorts an unordered list using divide-and-conquer. It first recursively splits up the list into different sublists until no more sublists can be constructed and merges them afterwards into an ordered list. Mergesort has a time complexity of $\mathcal{O}(n \log n)$ in the worst case, $\mathcal{O}(\log n)$ for splitting the list up into different sublists and $\mathcal{O}(n)$ for the merging operation.

In Haskell, we implement mergesort using two functions `merge` and `mergeSort`. The function `merge` takes two ordered lists and merges them recursively such that the new list is ordered as well.

```
1 merge :: Ord a => [a] -> [a] -> [a]
2 merge xs [] = xs
3 merge [] ys = ys
4 merge (x : xs) (y : ys)
5   | x <= y = x : merge xs (y : ys)
6   | otherwise = y : merge (x : xs) ys
```

Listing 3.12: Example of a Haskell function `merge` that merges two ordered lists into a single ordered list.

We use pattern matching for the base cases, which is that one of both lists is empty, which simply returns the other list. The recursive pattern at the bottom matches two lists such that `x` and `y` are taken from the first and second list respectively. It then checks using guards whether `x` is smaller or equal to `y` and appends `x` to the result of the recursive call of the tail `xs` of

the first list and the second list. Otherwise if `y` is greater than `x` it does the same but for `y` and `ys`.

We can now use the function `myMerge` to construct the mergesort algorithm:

```
mergeSort :: Ord a => [a] -> [a]
mergeSort [] = []
mergeSort [a] = [a]
mergeSort xs = myMerge (mergeSort xs') (mergeSort xs'')
  where
    xs' = take n xs
    xs'' = drop n xs
    n = length xs `div` 2
```

Listing 3.13: Example of a Haskell function `mergeSort` that performs the mergesort algorithm using `merge`.

The function `mergeSort` takes a list of any orderable type and maps it to another list of the same type. Here we use pattern matching as well. The base cases are the empty list and the singleton list (i.e. the list that consists of a single element). The recursive pattern matches a non-empty and non-singleton list and recursively applies the function `mergeSort` to the first and second halves of the list and merges these two halves using `merge`. We compare our implementation in Haskell with the Java implementation from [9]:

```
public static <K> void merge(K[] S1, K[] S2, K[] S,
    Comparator<K> comp) {
    int i = 0, j = 0;
    while (i + j < S.length) {
        if (j == S2.length || (i < S1.length && comp.
    compare(S1[i], S2[j]) < 0))
            S[i+j] = S1[i++];
        else
            S[i+j] = S2[j++];
    }
}

public static <K> void mergeSort(K[] S, Comparator<K>
    comp) {
    int n = S.length;
    if (n < 2) return;

    int mid = n/2;
    K[] S1 = Arrays.copyOfRange(S, 0, mid);
    K[] S2 = Arrays.copyOfRange(S, mid, n);
```

```
18
19      mergeSort(S1, comp);
20      mergeSort(S2, comp);
21
22      merge(S1, S2, S, comp);
23  }
```

Listing 3.14: Example of a Java method `mergeSort()` that performs the mergesort algorithm using `merge()`.

Here, some differences between the two implementation are observered:

- Once again, Haskell's type system shows how it allows for a relatively simple definition of types. The function `myMergeSort` simply depends on the fact that the two lists are of a type that is orderable. The implementation in Java, on the other hand, depends not only on generics, but also on a comparator method.
- Another differnece is that the Java implementation is intuitively more difficult to understand than the declarative Haskell implementation.

3.4.3 Quicksort

Quicksort is another sorting algorithm that also sorts unordered lists using divide-and-conquer with a worst-case time complexity of $\mathcal{O}(n^2)$ and an average-case time complexity of $\mathcal{O}(n \log n)$ It is often used as an example to show Haskell's expressiveness through its declarative style.

Our Haskell implementation is based on the implementation in [3] and looks as follows:

```
1  quickSort :: Ord a => [a] -> [a]
2  quickSort [] = []
3  quickSort (x : xs) = quickSort smaller ++ [x] ++
    quickSort larger
4    where
5      smaller = [a | a <- xs, a <= x]
6      larger = [b | b <- xs, b > x]
```

Listing 3.15: Example of a Haskell function `quickSort` that sorts an unordered list using the quicksort algorithm.

The function definition states that `quickSort` is a function that takes a list containing orderable elements as its argument and returns another list with the same type. The first pattern simply maps the empty list to itself. The second pattern takes the head `x` of the list and constructs a list such

that the singleton list containing `x` is in the middle between the result of `quickSort` applied to the two lists `smaller` and `larger`. The list `smaller` is constructed from all elements of the tail `xs` of the list that are smaller or equal to `x`. This concept is called *list comprehension* and will be described later. The list `larger` contains all elements from `xs` that are greater than `x`.

The following tree visualizes how `quickSort` works on an example input:

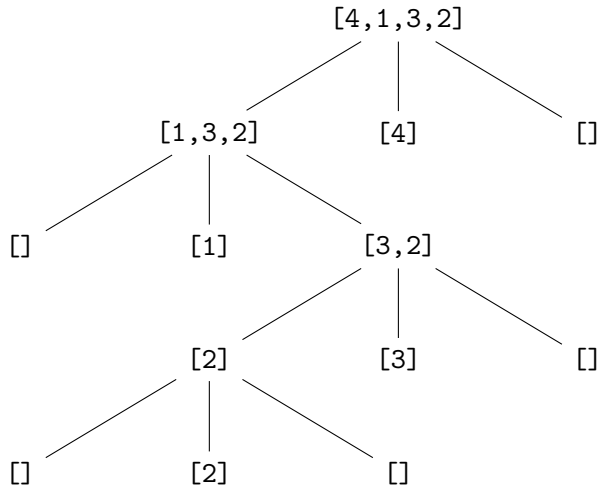

Figure 3.2: Tree showing how `quickSort` sorts the list `[4,1,3,2]`.

In the tree of the figure above, every left edge represents a recursive call with the smaller list (i.e. `quickSort smaller`), every middle edge represents the singleton list consisting of the head (i.e. `[x]`) and every right edge represents a recursive call with the larger list (i.e. `quickSort larger`). On each level below the root, the three siblings get concatenated from left to right, which corresponds to `quickSort smaller ++ [x] + quickSort larger`. The result of that concatenation then replaces the parent node and the steps get repreated again until the root is reached.

For example, if we start at the lowest level and concatenate all siblings (i.e. `[] ++ [2] ++ []`), we get `[2]` which we then use to overwrite the parent node. At the second-lowest level, we concatenate all siblings (i.e. `[2] + [3] + []`) and get `[2,3]`. We then overwrite the parent node of that level with `[2,3]`. On the third-highest level, we concatenate all siblings (i.e. `[] + [1] + [2,3]`) and replace the parent node with `[1,2,3]`. At the second-highest level, we concatenate all siblings (`[1,2,3] ++ [4] ++ []`) and finally receive the sorted list `[1,2,3,4]`.

We now compare the quicksort algorithm with a reference implementation in Java found in [9]:

```java
public static <K> void quickSort(Queue<K> S, Comparator<K
    > comp) {
     int n = S.size();
     if (n < 2) return;

     K pivot = S.first();
     Queue<K> L = new LinkedQueue<>();
     Queue<K> E = new LinkedQueue<>();
     Queue<K> G = new LinkedQueue<>();

     while (!S.isEmpty()) {
         K element = S.dequeue();
         int c = comp.compare(element, pivot);
         if (c < 0)
             L.enqueue(element);
         else if (c == 0)
             E.enqueue(element);
         else
             G.enqueue(element);
     }

     quickSort(L, comp);
     quickSort(G, comp);

     while (!L.isEmpty())
         S.enqueue(L.dequeue());
     while (!E.isEmpty())
         S.enqueue(E.dequeue());
     while (!G.isEmpty())
         S.enqueue(G.dequeue());
}
```

Listing 3.16: Example of a Java method `quickSort` that performs the quicksort algorithm.

We make some observations:

- Just like in the previous examples, Haskell's type system allows for simple function type definitions in contrast to Java. While the Haskell implementation depends on simple lists that contain orderable elements, the Java implementation depends on objects that implement the `Queue` interface and a comparator object.
- The Haskell solution is easier to digest and understand. Using pattern matching, recursion and list comprehension, we have a solution that is not only more compact, but also easier to reason about. The Java solution, however, upon first glance, may cause confusion. It is slightly

more complex to understand, since it uses three `LinkedQueue` instances for storing the elements smaller, equal or greater than the current element, each of which gets manipulated during the recursive steps.

Chapter 4

Advanced Function Concepts

4.1 Currying

The concept of *Currying* is named after Haskell Curry (who is also the namesake of *Haskell*). Currying is also found in other programming languages that support functional features, such as JavaScript. One advantage of Currying is that we do not have to depend on tuple types in order to pass multiple arguments to a function. Another advantage of Currying is that it allows for *partial application* of functions, i.e. construcing functions that are already defined for certain values. Since we often work with functions that use multiple arguments, currying is found everywhere in Haskell.

4.2 Higher-order Functions

Higher-order functions are a very important part of functional programming. In simple terms, a higher-order function is a function that can take another function as its argument.

A well-known example of a higher-order function that is found in other programming languages as well is the function `map`, which takes a function and a list as its arguments and applies the function to all elements of the list. A very basic example of `map` using a function that returns the square of a number is given here:

```
1 square x = x * x
2 map square [1..10]
3 -- evaluates to [1,4,9,16,25,36,49,64,81,100]
```

Listing 4.1: Example of applying `map` to a function `square` that squares a given number and to a list of numbers.

The list `[1..10]` generates the list of all integers from 1 to 10.

4.3 Anonymous Functions

Anonymous functions are another concept that are commonly found in other programming languages borrowing elements from functional programming. Simply put, an anonymous function is simply a function that does not have a name. It instead consists simply of its body.

We can rewrite the previous example using an anonymous function instead:

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

Listing 4.2: The function `square` rewritten as an anonymous function.

The backslash \ at the beginning of the anonymous function is called a *lambda*. After the lambda, the arguments of the anonymous functions are defined. Multiple arguments are separated using spaces. After the arguments, an arrow `->` is written and the expression of the function follows.

Anonymous functions are often also called *lambda expressions*. The theory behind anonymous functions and functional programming in general stems from the area of *lambda calculus*. Lambda calculus is a model of computation (such as e.g. Turing machines) that is based on functions.

Chapter 5

Lazy Evaluation

5.1 What is Lazy Evaluation?

The concept of *lazy evaluation* allows for values to be computed only when they are really needed. The converse of lazy evaluation is *strict evaluation* (also called *eager evaluation*). Most imperative programming languages are strictly evaluated and some languages support both lazy and strict evaluation (such as Scala). Lazy evaluation can be a big advantage for certain use cases.

A particularly popular use case for lazy evaluation is the construction and usage of infinite lists. Using the basic `fib` function from the third chapter, we can create the a function `fibs` computes an infinite list consisting of Fibonacci numbers:

```
1 fibs = map fib [2..]
```

Listing 5.1: Example of a function `fibs` that generates an infinite list of Fibonacci numbers.

The above function definition simply maps the function `fib` to the infinite list consisting of all numbers greater than one. Now, using a second function `nFibs`, we can define a function that returns the first n Fibonacci numbers:

```
1 nFibs n = take n fibs
```

Listing 5.2: Example of a function `nFibs` that returns the first n Fibonacci numbers using the function `fibs`

The function `nFibs` simply takes the first n elements from the infinite list

returned by `fibs`. For example, if we call `nFibs 8`, the function returns `[1,2,3,5,8,13,21,34]`.

There is however a major flaw with this implementation of `fibs`: It is computationally inefficient. For every list entry i, we have to compute the i-th Fibonacci entry from the beginning using `fibs`. This implementation does not have any concept of "remembering the previous values".

An alternative implementation would be the following function `fibs'`:

```
1 fibs' = fibsAux 1 2
2   where fibsAux n1 n2 = n1 : fibsAux n2 (n1 + n2)
3
4 nFibs' n = take n fibs'
```

Listing 5.3: Function `fibs'` that computes an infinite list of Fibonacci numbers using the basic definition of Fibonacci numbers and `nFibs'` that returns a list of n Fibonacci numbers.

The function definition above states that the function `fibs'` returns the result of an auxiliary function `fibsAux` applied to the arguments 1 and 1, which recursively computes the Fibonacci numbers without a base case, i.e. effectively infinitely. The function `nFibs'` works the same as `nFibs` and simply returns the n first Fibonacci numbers.

We can compare the (first time) execution speed of `nFibs` and `nFibs'` applied to the number 30. The functions were executed on a MacBook Air 2020 (macOS 12.6 21G115 x86_64, Intel i3-1000NG4 at 1.10GHz, 8 GB RAM) and timed using the command `:set +s` on GHCi:

- `nFibs`: The function `nFibs` computed the first 30 Fibonacci numbers in 11.22 seconds whilst allocating up to 5,059,236,024 bytes.
- `nFibs'`: The function `nFibs'` computed the first 30 Fibonacci numbers in 0.07 seconds whilst allocating up to 177,952 bytes.

There is a drastic difference between both implementations. The key takeaway from this is that great care has to be taken when defining infinite lists. One should always check whether more complicated solutions can be broken down using first principles.

Chapter 6

Advanced Type Concepts

This chapter discusses various advanced aspects of typing in Haskell.

6.1 Custom Types

Custom types can be defined using the keyword `type`. For example, if one wants to define a custom type for a key-value list, one can define the following type:

```
1 type KeyValueList k v = [(k,v)]
```

Listing 6.1: Example of a type definition for `KeyValueList`.

The above definition states that `KeyValueList` is a type of a list that can take tuples with the first element being of any type `k` and the second element being of any type `v`.

6.2 Type Classes

In Haskell, the concept of *type classes* exists. The term class does not refer to the concept of classes found in object-oriented programming languages, but rather on the classification of types into distinct classes. A type class is essentially a collection of types with functionality that is associated to them in the form of *methods*. A type is said to belong to one or multiple type classes.

A basic type class that was already mentioned is the `Num` class. It has the methods `+`, `-`, `*`, `negate`, `abs` and `signum`. Other examples include the class of orderable types `Ord`, the class of equality-comparable types `Eq` and the class of showable types (i.e. types that can be converted into a string representation) `Show`.

Custom type classes can be defined as well. For this, the keyword `class` is used. A good introductory example that is mentioned in [3, p. 111] is the definition of `Eq` in the standard prelude:

```
class Eq a  where
  (==), (/=) :: a -> a -> Bool
  x /= y = not (x == y)
```

Listing 6.2: Definition of `Eq` in the standard prelude.

6.3 Polymorphic Types

The concept of *polymorphic types* allow more generic implementations of functions, i.e. instead of having to define multiple different functions for each type, we can instead define a single function that works for all (allowed) types.

Polymorphic types were already encountered in previous chapters, e.g. in the sorting functions, which allowed for lists with values of any type `a` that belongs to the class `Ord` to be used with the sorting functions.

Another example that demonstrates the concept of polymorphic types is the custom implementation of a function `curry` that converts a given uncurried function with two arguments into a curried function:

```
curry :: ((a, b) -> c) -> (a -> b -> c)
curry f = \x -> \y -> f (x, y)
```

Listing 6.3: Example of a function `curry` that converts a two-argument uncurried function into a curried function.

In the above example we have three polymorphic types, namely `a`, `b` and `c`. The function `curry` simply maps a given function `f` into a curried function by returning lambda functions that map to the original functions, thereby turning the uncurried function into a curried one.

Another example of a function with polymorphic types is a custom implementation of the `map` function:

```
map' :: Foldable a => (b -> c) -> (a b -> [c])
map' f = foldr (\a xs -> f a : xs) []
```

Listing 6.4: Example of a function `map'` that maps a given function to a list and returns the resulting list.

The function takes as its arguments a function that maps from a type `b` to a type `c` and `a` that belongs to the type class `Foldable` and returns a function that takes two arguments of type `a` and `b` that maps to a list of type `c`. The function takes a function `f` as its argument and applies (foldr) to a lambda function with arguments `a` and `xs` that applies `f` to `a` and calls the list constructor on `xs`. The initial value is the empty list.

6.4 Recursive Types

Types in Haskell can also be defined in a recursive manner. This is especially useful in e.g. tree-like structures, which are by their nature recursive.

6.4.1 Example of a Binary Tree

The following example shows how a binary tree and a general tree can be defined in Haskell using recursive types:

```
1 data BinTree a
2   = BinTreeLeaf a
3   | BinTreeNode (BinTree a) (BinTree a)
```

Listing 6.5: Example of a recursive type `BinTree` to model binary trees.

The first data definition states that `BinTree` is a type containing a polymorphic type `a`. The bar symbol `|` represents choice, a notation that is also found in formal language theory. A value of type `BinTree` either takes form as `BinTreeLeaf a`, i.e. as the label `BinTreeLeaf` with an associated value of type `a`, or it takes form as `BinTreeNode (BinTree a) (BinTree a)`, i.e. as the label `BinTreeNode` with two assoicated values of the type `BinTree a`, both of which can once again take form as either a `BinTreeLeaf` or a `BinTreeNode`.

A basic example is given of a binary tree defined using the above data type:

```
1 binTreeExample :: BinTree [Char]
2 binTreeExample =
3   BinTreeNode
4     (BinTreeLeaf "root")
5     ( BinTreeNode
6         (BinTreeLeaf "left child")
7         (BinTreeLeaf "right child")
8     )
```

Listing 6.6: Example of a value of the type `BinTree`.

This binary tree corresponds to the following binary tree:

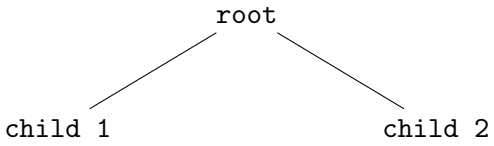

Figure 6.1: Tree representing the Haskell value `binTreeExample`.

6.4.2 Example of a General Tree

The following example shows how a general tree can be defined in Haskell using a similar approach to the binary tree. The approach is in fact the same, except for the fact that instead of having two children, a tree node can have any number of children.

```haskell
data Tree a
  = TreeLeaf a
  | TreeNode [Tree a]
```

Listing 6.7: Example of a recursive type `Tree` to model trees.

The following listing demonstrates a basic example of a tree defined with the type `Tree`:

```haskell
treeExample :: Tree [Char]
treeExample =
  TreeNode
    [ TreeLeaf "root",
      TreeNode
        [ TreeLeaf "child 1",
          TreeLeaf "child 2",
          TreeNode
            [ TreeLeaf "grandchild 1",
              TreeLeaf "grandchild 2",
              TreeLeaf "grandchild 3"
            ],
          TreeLeaf "child 3"
        ]
    ]
```

Listing 6.8: Example of a value of the type `Tree`.

This tree corresponds to the following tree:

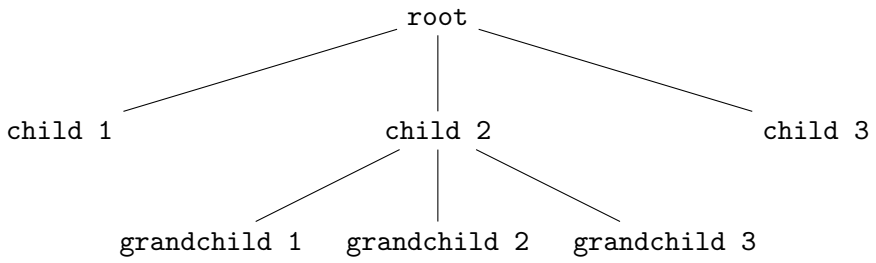

Figure 6.2: Tree representing the Haskell value `treeExample`.

The interesting aspect of such recursive types is the fact that their definition is relatively simple and that values of such types ressemble trees as well. Another aspect that is noteworthy is the fact that it uses the choice symbol `|` to model alternatives. This can be very useful to define syntax trees, which will be shown later on.

Chapter 7

Game of Nim

7.1 Introduction

The *Game of Nim* is a mathematical strategy game. The game consists of four rows of matches, with each row containing 1,3,5 and 7 matches respectively. It is played by two players.

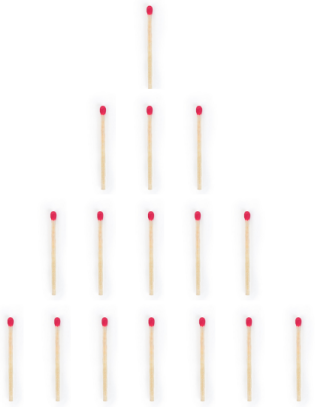

Figure 7.1: Game of Nim setup with 4 matches.

The objective is to be the last player to remove a match from the last row. The game ends when a player takes the last match from the last row, resulting in a winning condition for that player. Each row contains a number of matches. Players take turns removing matches from any row.

7.2 Implementation

This section is about an in-depth exploration of the Game of Nim, implemented in Haskell. The Game of Nim is a well-known mathematical game

that has been widely studied due to its intriguing properties and practical applications. The objective is to showcase the elegance and efficiency of Haskell in implementing the Game of Nim.

7.2.1 Winning Condition

To understand how to achieve the winning condition, we must examine the mathematical properties of the game. In particular, we can use binary representation to analyze the row sizes and develop optimal strategies for removing objects.

One important property of the Game of Nim is that the XOR operation is commutative, associative, and has the property that a number XORed with itself equals 0. Using these properties, we can derive the following strategy for removing matches:

1. Find the binary representation of the row sizes.
2. Compute the XOR of all the row sizes.
3. If the computed value is 0, we are in a losing position and the opponent has a winning strategy, and we can only hope to delay the inevitable.
4. If the value is not equal to 0, find the row that we should remove matches from.

7.2.2 Data Representation and Conversion

The game state is represented as a list of integers, where each integer corresponds to the number of matches in each row. For example, the initial game state is represented as [1,3,5,7].

To determine whether a game state is winning or losing, the code uses bitwise XOR operations on the binary representations of the numbers in each row. The `int2Bin3Bits` function converts an integer into its 3-bit binary representation as a list of integers (0s and 1s), using pattern matching. The `game2Bin` function then uses the higher order function `map` to convert the entire game state into its binary representation.

```
1 int2Bin3Bits :: Int -> [Int]
2 int2Bin3Bits 0 = [0, 0, 0]
3 int2Bin3Bits 1 = [0, 0, 1]
4 int2Bin3Bits 2 = [0, 1, 0]
5 int2Bin3Bits 3 = [0, 1, 1]
6 int2Bin3Bits 4 = [1, 0, 0]
7 int2Bin3Bits 5 = [1, 0, 1]
8 int2Bin3Bits 6 = [1, 1, 0]
9 int2Bin3Bits 7 = [1, 1, 1]
```

```
10
11 game2Bin :: [Int] -> [[Int]]
12 game2Bin game = map int2Bin3Bits game
13
```

Listing 7.1: The GameofNim functions `int2Bin3Bits` and `game2Bin` that convert a list of integers into its 3-bit binary representation

7.2.3 Determining Winning and Losing Game States

This is achieved using mostly higher order functions like `foldr`, `map` and `sum`. The `isWinning` function checks whether a game is in a winning or losing state, by calculating the bitwise XOR of the binary representations of each row. Then `rowCol` function is used to transpose the binary representations, followed by the `verticalSumMod2` function to calculate the bitwise XOR.

```
1 rowCol :: [[Int]] -> [[Int]]
2 rowCol [] = []
3 rowCol ([]:rows) = []
4 rowCol rows = (map head rows) : (rowCol (map tail rows))
5
6 verticalSumMod2 :: [[Int]] -> [Int]
7 verticalSumMod2 lstlst = map ((`mod` 2) . sum) lstlst
8
9 isWinning :: [Int] -> Bool
10 isWinning game = foldr ((||) . ((/=) 0)) False
11                       (verticalSumMod2 (rowCol (game2Bin
    game)))
```

Listing 7.2: The GameofNim functions `rowCol`, `verticalSumMod2` and `isWinning`, which calculate if the current gamestate is winning or loosing

7.2.4 Gameplay

The `remove` function updates the game state by removing a given number of matches from a specified row, while the `isFinished` function checks whether the game has ended by verifying if all rows are empty. These two use a combination of pattern matching and higher order functions, such as `foldr`.

```
1 remove :: Int -> Int -> [Int] -> [Int]
2 remove 1 matches game = ((head game)-matches) : (tail
    game)
3 remove 2 matches game = (take 1 game) ++ (((game!!1)-
    matches) : (drop 2 game))
```

```
remove 3 matches game = (take 2 game) ++ (((game!!2)-
    matches) : (drop 3 game))
remove 4 matches game = (take 3 game) ++ (((game!!3)-
    matches) : [])

isFinished :: [Int] -> Bool
isFinished game = foldr ((&&) . ((==) 0)) True game
```

Listing 7.3: The GameofNim functions `remove` and `isFinished`

To display the game state, the `displayGame` function prints the number of matches in each row. The `inputRow` and `inputMatches` functions prompt the user to input the row and the number of matches they want to remove, respectively.

```
displayGame :: [Int] -> IO()
displayGame game =
   do putStrLn ""
      displayGame' game 1
      putStrLn ""
```

Listing 7.4: The GameofNim function `displayGame` which prints the gamestate using the `IO()` monad.

7.2.5 Player and Computer Turns

The `playerTurn` function implements the player's turn, while using the `IO()` monad to read from the input. The computer's strategy, used in the function `computerTurn` is based on whether the current game state is winning or losing, which is computed using the `game2Bin` function seen above. If the state is winning, the computer searches for a move that results in a losing state for the player. If the state is losing, the computer removes one match from the last possible row.

```
playerTurn :: [Int] -> IO (Int, Int, [Int])
playerTurn game =
   do row <- inputRow game
      matches <- inputMatches row game
      return (row, matches, remove row matches game)

      computerTurn :: [Int] -> (Int, Int, [Int])
computerTurn game =
   if isWinning game then
      case searchRowMatches 1 game of
```

```
11          (row, matches) -> (row, matches, remove row
    matches game)
12      else
13          case searchOneRowMatch (length game) game of
14              (row, matches) -> (row, matches, remove row
    matches game)
```

Listing 7.5: The GameofNim functions `playerTurn` and `computerTurn`.

7.2.6 Main Game Loop

The `playInTurn` function alternates between the computer's and the player's turns until the game ends. The `nim` function initializes the game and prompts the user to choose whether they want to start first, using the `IO()` monad. It then calls the `playInTurn` function with the appropriate boolean value.

7.2.7 Functional Aspects and Side Effects

In this section, we will further analyze the functional aspects of the Game of Nim implementation in Haskell, with an emphasis on the importance of side-effect free functions.

Purity and side effect free functions Functional programming languages like Haskell rely on the concept of pure functions, which are functions that, for the same input, always produce the same output and have no side effects. This property contributes significantly to the predictability and readability of the code, making it easier to understand, test, and debug.

In our Nim implementation, functions like `int2Bin3Bits`, `game2Bin`, `isWinning`, `remove`, and `isFinished` are all examples of pure functions. They take inputs and produce outputs without modifying any external state or causing side effects.

Monads and side effects Haskell uses monads to manage side effects without sacrificing the purity of functions. Side effects refer to changes in state that do not result from the function's return values. Examples of side effects include reading or writing to global variables, modifying data in place, or performing I/O operations.

In our implementation, we use the IO monad to handle the game's input and output operations. The `IO()` monad is used in functions like `displayGame`, `inputRow`, `inputMatches`, and `playerTurn`. For instance, the `displayGame` function uses the `IO()` monad to print the current game state, while `playerTurn` uses it to read the player's input.

Higher-order functions Higher order functions, which can take other functions as arguments and/or return functions as results, are a core concept in functional programming. They allow for more abstract and flexible code. Haskell's `map`, `foldr`, and `filter` are examples of higher order functions used extensively in our implementation.

In the `game2Bin` function, we use the `map` function to apply `int2Bin3Bits` to each element of the game state. In the `isWinning` function, we use `foldr` to reduce the list of integers to a single boolean value.

Recursion Recursion is a fundamental concept in functional programming, replacing loops found in imperative languages. In our implementation, we use recursion in various places, such as in the `rowCol` function to transpose the binary representations of the game state.

7.3 Conclusion

We implemented the Game of Nim using Haskell, taking advantage of language features such as pattern matching, higher-order functions, and the IO monad. These constructs allowed us to create a concise representation of the game logic, resulting in a smooth and engaging player experience.

The game state was depicted as a list of integers, and we used bitwise XOR operations to analyze the game's mathematical properties and develop optimal strategies. This implementation showcases Haskell's ability to tackle complex game logic with clarity and expressiveness.

Furthermore, we integrated user input and output using the IO monad, highlighting Haskell's skill in managing side effects while maintaining the purity of the functional paradigm.

Chapter 8

Unification

8.1 Introduction

8.1.1 What is Unification?

Unification is a concept in computer science that is found in various subfields, such as first-order logic and type theory. The main purpose of unification is the automatic solving of equations of terms that consist of variables, constants and compound terms.

The main goal of this chapter is to show how Haskell can be used to easily create parsers for context-free grammars and how the unification problem can be solved elegantly in Haskell.

8.1.2 Basic Examples

The following basic examples demonstrate the main concept behind unification and the problems that can occur during the process. The examples are taken from [1].

Example 1: Food and Fruit

Let X and Y be two variables and T_1 and T_2 be two terms defined as follows:

$$T_1 : food(X, fruit(apple, X))$$
$$T_2 : food(kiwi, fruit(Y, X)).$$

The goal is to find substitutions for the variables above such that the terms are equal. We can find two such substitutions, namely we can substitute the variable X by the constant $kiwi$ and the variable Y by the constant $apple$. By applying these substitutions, T_1 and T_2 are the same. Formally, the resulting substitutions are

$$UNIFY(T_1, T_2) = \{X \mapsto kiwi, Y \mapsto apple\}.$$

Example 2: Inferring Knowledge

Let X and Y be two varibles and $T_1, \ldots, T_4$ be four terms defined as follows:

$$T_1 : Knows(John, Jane)$$
$$T_2 : Knows(Y, Bill)$$
$$T_3 : Knows(Y, Mother(Y))$$
$$T_4 : Knows(X, Elizabeth).$$

Suppose we have an additional term $T = Knows(John, X)$ and we want to find the unification of it with all other terms. This results in the following unification:

$$UNIFY(T, T_1) = \{X \mapsto Jane\}$$
$$UNIFY(T, T_2) = \{X \mapsto Bill, Y \mapsto John\}$$
$$UNIFY(T, T_3) = \{Y \mapsto John, X \mapsto Mother(John)\}$$
$$UNIFY(T, T_4) = fail.$$

The unification of T and T_1 is trivial, since we can substitute X by $Jane$ in order to get two equal terms. Similarly, the unification of T and T_2 indicates that in order to get two equal terms, we have to replace X by $Bill$ and Y by $John$. The same goes for T and T_3. However, the unification of T and T_4 fails. The reason for this is that the variable X can only take one value at a time. It cannot take both $John$ and $Elizabeth$ at the same time. This can be fixed using a different variable name, such as Z. If we redefine T_4 to be $T_4' : Knows(Z, Elizabeth)$, we get the following:

$$UNIFY(T, T_4') = \{X \mapsto Elizabeth, Z \mapsto John\}.$$

Another problem that arises is that there could be more than a single unifier. The following unification serves as an example:

$$UNIFY(Knows(John, X), Knows(Y, Z))$$

The returned value of $UNIFY()$ could either be $\{Y \mapsto John, X \mapsto Z\}$ or $\{Y \mapsto John, X \mapsto John, Z \mapsto John\}$. The result of the unification of the first unifier would then be $Knows(John, Z)$ and the that of the second unifier would be $Knows(John, John)$. The second result can also be obtained using the first result by adding an additional substitution $\{Z \mapsto John\}$. The first unifier is considered to be more general than the second rule, since there are fewer restrictions on which values the variables can take. For every unifiable pair of terms, there exists a *most general unifier* (MGU) that is unique up to a renaming of the variables, which in this example is the first unifier $\{Y \mapsto John, X \mapsto Z\}$.

8.2 Parsing

Our approach for parsing is based on the concept of *parser combinators*, a series of smaller parsing functions that can be composed to create larger, more complex parsers. The implemented parser is a *recursive-descent parser*. Our implementation is heavily based on *Parsing with Haskell* by Andersson [10], in which the parsing functions are all hand-written. Another possible approach is to use *Parsec* [11], a parsing library for Haskell.

The parser is encapsulated in the module `Parser` which exposes only the most important functions and types for outside usage. The module is defined in the file `parser.hs`.

8.2.1 Grammar for Terms

The unification terms can be represented as strings. For example, the term $Knows(John, X)$ can be represented as `"Knows(John,X)"`. The following grammar recognizes such strings and is given in the EBNF notation:

$$
\begin{aligned}
\langle term \rangle &::= \langle constant \rangle \\
&\mid \langle variable \rangle \\
&\mid \langle compound \rangle \text{ `(' } \langle term \rangle \text{ } \{\text{`,' } \langle term \rangle\} \text{ `)'}
\end{aligned}
$$

$$
\langle constant \rangle ::= \langle letter \rangle \ \langle letter \rangle \ \{\langle letter \rangle\}
$$

$$
\langle variable \rangle ::= \langle letter \rangle
$$

$$
\langle compound \rangle ::= \langle letter \rangle \ \langle letter \rangle \ \{\langle letter \rangle\}
$$

$$
\langle letter \rangle ::= \text{`a'} \mid \ldots \mid \text{`z'} \mid \text{`A'} \mid \ldots \mid \text{`Z'}
$$

The dots $(\ldots)$ in the rule $\langle letter \rangle$ denote the rest of the lowercase and uppercase letters of the alphabet.

8.2.2 `Parser` Data Type

The parser is constructed such that it takes a string, processes it and returns a tuple of the parsed expression and the rest of the unparsed string. The main advantage of such a construction is that parsers can easily be combined together. The basic parser type is defined as follows:

```
1 type Parser a = String -> Maybe (a, String)
```

Listing 8.1: Definition of the type `Parser`.

The above type definition states that `Parser a` corresponds to the type `String -> Maybe (a, String)`. The right-hand side can be seen as a synonym of the left-hand side. The `Maybe` type allows for optional types and is defined in the prelude as follows:

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

Listing 8.2: Definition of the type `Maybe` in the prelude.

A value of the type `Maybe` can either take a value of type `a` (which indicates that a value exists), or `Nothing` (which indicates that a value does not exist). The main advantage of the `Maybe` type is that it can be used to indicated that a parse has failed.

8.2.3 Basic Parser Functions

There are two types of parser functions: The parser operators and the parser functions. Both will be used extensively to construct the parser for the grammar defined above. The parser functions can be seen as the basic building blocks of parser while the parser operators can be seen as the linking parts that link the building blocks together. Our own parser functions for the grammar will be explained shortly.

Parser Operators

The basic parser operators are defined as *infix* functions (i.e. functions that can be written between their two arguments). They take and return values of the type `Parser`. The following parser operators were used:

- `(?)`: The function `(?)` parses a string using a given parser and checks whether the parsed string satifies a predicate (i.e. a function that returns true or false). If the input satifies the predicate, then the parsed input is returned along with the rest of the string. Otherwise, it returns `Nothing`. It is defined as follows:

```
1 infix 7 ?
2 (?) :: Parser a -> (a -> Bool) -> Parser a
3 (m ? p) cs =
4   case m cs of
5     Nothing -> Nothing
6     Just (a, cs) -> if p a then Just (a, cs) else
    Nothing
7
```

Listing 8.3: Definition of the parser operator `(?)`.

- `(!)`: The function `(!)` takes two parsers. If the first parser is successful, it returns the parsed input along with the rest of the string. Otherwise, it returns the result of the second parser applied to the input. This parser operator represents the notion of choice in formal grammars. It is defined as follows:

```
infixl 3 !
(!) :: Parser a -> Parser a -> Parser a
(m ! n) cs =
  case m cs of
    Nothing -> n cs
    mcs -> mcs

```

Listing 8.4: Definition of the parser operator `(!)`.

- `(#)`: The function `(#)` simply applies two parsers in sequence. The end result is a tuple `((p, k), cs'')`, where `p` is the result of the first parse, `k` is the result of the second parse and `cs''` is the rest of the string after the second parse. It is defined as follows:

```
infixl 6 #
(#) :: Parser a -> Parser b -> Parser (a, b)
(m # n) cs =
  case m cs of
    Nothing -> Nothing
    Just (p, cs') ->
      case n cs' of
        Nothing -> Nothing
        Just (q, cs'') -> Just ((p, q), cs'')

```

Listing 8.5: Definition of the parser operator `(#)`.

- `(>->)`: The function `(>->)` transforms the result of a parse using a given function. In case of a successful parse, the given function is applied to the result of the parse and returned along with the rest of the string. It is defined as follows:

```
infixl 5 >->
(>->) :: Parser a -> (a -> b) -> Parser b
(m >-> k) cs =
  case m cs of
```

```
5       Just (a, cs') -> Just (k a, cs')
6       Nothing -> Nothing
7
```

Listing 8.6: Definition of the parser operator `(>->)`.

- `#>`: Similarly to `>->`, the function `#>` passes the result of a parse to a given function. The difference here is that the function is of the type `a -> Parser b` instead. It is defined as follows:

```
1 infix 4 #>
2 (#>) :: Parser a -> (a -> Parser b) -> Parser b
3 (m #> k) cs =
4   case m cs of
5     Nothing -> Nothing
6     Just (a, cs') -> k a cs'
7
```

Listing 8.7: Definition of the parser operator `(#>)`.

- `-#` and `#-`: The functions `-#` and `#-` simply ignore the result of the first and second parse respectively. They are defined as follows:

```
1  infixl 6 -#
2  (-#) :: Parser a -> Parser b -> Parser b
3  (m -# n) cs =
4    case m cs of
5      Nothing -> Nothing
6      Just (p, cs') ->
7        case n cs' of
8          Nothing -> Nothing
9          Just (q, cs'') -> Just (q, cs'')
10
11 infixl 6 #-
12 (#-) :: Parser a -> Parser b -> Parser a
13 (m #- n) cs =
14   case m cs of
15     Nothing -> Nothing
16     Just (p, cs') ->
17       case n cs' of
18         Nothing -> Nothing
19         Just (q, cs'') -> Just (p, cs'')
20
```

Listing 8.8: Definition of the parser operators `-#` and `#-`.

The `infix` and `infixl` functions declares that a given function is an infix operator with a given paranthesis precedence. The `infixl` function additionally declares a left-association of the given function.

Parser Functions

The parser functions are defined as normal functions. The following parser functions were used:

- `return`: The function `return` simply returns the argument and the input string without inspecting it. A function named `return` is already defined in the standard prelude, so it needs to be hidden in order to avoid naming conflicts:

```
1 import Prelude hiding (return)
2
```

Listing 8.9: Hiding `return` from the standard prelude.

Our function `return` is defined as follows:

```
1 return :: a -> Parser a
2 return a cs = Just (a, cs)
3
```

Listing 8.10: Definition of the parser function `return`.

- `cons`: The function `cons` takes a tuple consisting of an element and a list of elements, both with the same type `a`, and prepends the element to the list. It is defined as follows:

```
1 cons :: (a, [a]) -> [a]
2 cons (hd, tl) = hd : tl
3
```

Listing 8.11: Definition of the parser function `cons`.

- `iter`: The function `iter` applies a parser to itself recursively until a `Nothing` appears The results of the recursive calls get constructed into a list with the `cons` function. It is defined as follows:

```
1 iter :: Parser a -> Parser [a]
2 iter m = m # iter m >-> cons ! return []
3
```

Listing 8.12: Definition of the parser function `iter`.

- `char`: The function `char` is a very simple parser that takes an input string and returns the tuple that consists of the first character of the input and the rest of the string. If the string is empty, it returns `Nothing`. It is defined as follows:

```
1 char :: Parser Char
2 char (c : cs) = Just (c, cs)
3 char [] = Nothing
4
```

Listing 8.13: Definition of the parser function `char`.

- `lit`: The function `lit` takes a character and returns a parser that checks whether the first character of a given string is equal to the given character. It is defined as follows:

```
1 lit :: Char -> Parser Char
2 lit c = char ? (== c)
3
```

Listing 8.14: Definition of the parser function `lit`.

The function `== c` is the infix form for the lambda expression `(\x -> x == c)`.

- `letter`: The function `letter` is the parser that checks whether the first character of a given string is an alphabetic character. It does so using the function `isAlpha` from the standard prelude.

```
1 letter ::  Parser Char
2 letter = char ? isAlpha
3
```

Listing 8.15: Definition of the parser function `letter`.

- `letters`: The function `letters` parses multiple letters in a string until no more can be parsed. It makes use of the `letter`, `iter` and `cons` functions that were previously defined. It is defined as following:

```
1 letters :: Parser String
2 letters  = letter # iter letter >-> cons
3
```

Listing 8.16: Definition of the parser function `letters`.

- `space`: The function `space` is similar to the function `char`, except that it checks whether the first character of the given string is a space using the standard prelude function `isSpace`. It is defined as follows:

```
1 space ::  Parser Char
2 space = char ? isSpace
3
```

Listing 8.17: Definition of the parser function `space`.

- `token`: The function `token` takes a parser and returns another parser that ignores all whitespace characters after the string. It is defined as follows:

```
1 token :: Parser a -> Parser a
2 token m  = m #- iter space
3
```

Listing 8.18: Definition of the parser function `token`.

- `word`: The function `word` simply parses multiple letters in sequence whilst ignoring any whitespace characters after the parsed letters. It is defined as follows:

```
1 word :: Parser String
2 word = token letters
3
```

Listing 8.19: Definition of the parser function `word`.

8.2.4 `Term` Data Type

Our parser needs to conform to the grammar defined above. We can model the rules of the grammar by defining a data type `Term` as follows:

```
data Term
  = Constant String
  | Variable Char
  | Compound String [Term]
  deriving Show
```

Listing 8.20: Definition of `Term` to model the rules of the grammar.

The above definition states that a term can take form as one of the following:

- As a constant that consists of a single string.
- As a variable that consists of a single character.
- As a compound that consists of a string (its name) and a list of terms.

For example, passing the string `"Knows(John,X)"` to the parser should return the parsed term `Compound "Knows" [Constant "John", Variable 'X']`.

8.2.5 Parser Functions for Unification Terms

We defined several parser functions for parsing the unification terms. All parser functions make heavy use of the previously defined basic functions. The following functions were used:

- `term`: The function `term` corresponds to the rule $\langle term \rangle$. It is defined as follows:

```
term :: Parser Term
term = (compound #> compound')
  ! constant
  ! variable

```

Listing 8.21: Definition of the parser function `term`. function

It first maps to the parser `(compound #> compound')`, which would correspond. If that parser returns `Nothing`, it continues with the parser `constant` and if `constant` returns `Nothing`, it finally continues with the parser `variable`.

One important aspect of this function is the order of the alternatives. The parser `term` needs to first check whether the given string is a compound, then whether it is a constant and finally whether it is a variable. If the order was reversed, i.e. if the top-most parser was `Variable`, the parser would not work properly. It would prematurely detect a variable, when in reality it could be a compound or a constant. For example, when given the string `"Knows(John,X)"`, the parser would simply return `Variable 'K'` with the rest of the string `"Knows(John,X)`.

- `term'`: The function `term'` is responsible for parsing terms inside the compound brackets. It is defined as follows:

```
1 term' :: Term -> Parser Term
2 term' t = term >-> addTermToCompound t #> term''
3
```

Listing 8.22: Definition of the parser function `term'`.

It takes a term (more specifically a compound), parses another term, then adds the newly parsed term to the original compound and passes the result to the function `term''`, which checks whether there is a comma character and repeats those steps.

- `term''`: The function `term''` is used to detect commas inside compound terms It is defined as follows:

```
1 term'' :: Term -> Parser Term
2 term'' t
3   = lit ',' term' t
4   ! return t
5
```

Listing 8.23: Definition of the parser function `term''`.

It takes a term (more specifically a compound) as its argument and is split up in two rules:

  - In the first rule, the function detects and ignores a comma character. It then adds parses a term and adds the term to the compound list using `addTermToCompound`. Afterwards, it recursively calls itself and performs the same steps using the new term returned by `addTermToCompound`.
  - The second rule represents the base case of the recursion (i.e. the first rule returned `Nothing`) and simply returns the term that was

passed. What finally gets returned is the built up compound with the parsed terms inside its list. It is only used by the function `compound'` to build up the compound list when a compound is detected by `term`.

- `compound`: The function `compound` corresponds to the rule ⟨*compound*⟩. It is defined as follows:

```
1 compound :: Parser Term
2 compound = word ? (\w -> length w >= 2) >-> (\w ->
      Compound w [])
3
```

Listing 8.24: Definition of the parser function `compound`.

It first parses a word using the function `word` and checks whether its length is greater than or equal to 2, analogue to the definition in the grammar. It does this using an anonymous function `(\w -> length w >= 2)`. Afterwards, it transforms the parsed string into a compound using the anonymous function `(\w -> Compound w [])`, which simply returns a compound term with its string component set to the argument `w`. In other words, the function `compound` detects the name of the parsed compound and returns a compound consisting of its name and an empty list.

- `compound'`: The function `compound'` corresponds to the brackets and the inner terms after the ⟨*compound*⟩ in the third alternative of the rule ⟨*term*⟩. It is defined as follows:

```
1 compound' :: Term -> Parser Term
2 compound' t = lit '(' -# term' t #- lit ')'
3
```

Listing 8.25: Definition of the parser function `compound'`.

It takes a term and first parses a left paranthesis and ignores it. It then parses a term using `term'` and then ignores a right paranthesis. The returned value is the term returned from `term' t`.

- `addTermToCompound`: The function `addTermToCompound` is a helper function that takes a compound and a term and pushes the term to the list of the compound. It is defined as follows:

```
addTermToCompound :: Term -> Term -> Term
addTermToCompound (Compound m n) x = Compound m (n
    ++ [x])
addTermToCompound _ _ = error "Not a compound."

```

Listing 8.26: Definition of the helper function `addTermToCompound`.

If the first argument is not a compound, an error gets triggered using the `error` function.

- `constant`: The function `constant` corresponds to the rule ⟨*constant*⟩. It parses a word and checks with an anonymous function whether the length of the parsed word is greater than or equal to 2. It then simply gets transformed into a constant with the `Constant` constructor. It is defined as follows:

```
constant :: Parser Term
constant = word ? (\w -> length w >= 2) >-> Constant

```

Listing 8.27: Definition of the parser function `constant`.

- `variable`: The function `variable` corresponds to the rule ⟨*variable*⟩ and simply parses a single letter using the function `letter` and transforms the letter to a variable using the `Variable` constructor. It is defined as follows:

```
variable :: Parser Term
variable = letter >-> Variable

```

Listing 8.28: Definition of the parser function `variable`.

- `parse`: The function `parse` is the top-level parsing function that is intended for usage outside of the module. It takes a string and returns a value of the type `Term`. It is defined as follows:

```
parse :: String -> Term
parse s = case term s of
  Just(t, "") -> t
  _ -> error "Syntax error."
```

```
5
```

Listing 8.29: Definition of the parser function `parse`.

The string passed to the function gets evaluated with the function `term`. If the returned tuple contains the term and the remaining string is empty, the term is returned. Otherwise, an error gets triggered using the `error` function, as a non-empty remaining string would imply that an error that occurred during the parsing.

8.2.6 Examples of Parsing

The following are a few basic examples that demonstrate how certain strings get parsed by the unification parser:

1. Let the string to be parsed be `"X"`. The parser first attempts to parse the string with the parser `compound`, which fails and returns `Nothing`, since the input string has a size of 1. The parser then proceeds to attempt to parse the string with the parser `constant`, which also fails for the same reason.

   It then finally parses the string using `variable`, which succeeds since the input string is a single alphabetic character. The value returned by the parser `term` is `(Variable 'X', "")`.

2. Let the string to be parsed be `"Knows(Y,Bill)"`. The parser first attempts to parse the string with the parser `compound`, which succeeds and returns `Just (Compound "Knows" [], "(Y,Bill)")`.

   Next it passes this value to the function `compound'`, ignores the first bracket and applies `term'` to the argument. The parser `term'` attempts to parse a term using `term`. The first two parsers `compound` and `constant` in `term` fail, as the second character of the input string is a comma (and thus not alphabetic). The third parser `variable` succeeds. The returned value of `term` is `Just (Variable 'Y', ",Bill)")`. That value gets passed to `addTermToCompound` along with the term `Compound "Knows" []` and `Variable 'Y'` is pushed to the back of the empty list of the compound, all of which results in a new compound `Compound "Knows" [Variable 'Y']`.

   Afterwards, this new compound gets passed to `term''`, and a comma character gets parsed and ignored. Next it passes the compound to `term'` and and the function `term` gets invoked again. Inside `term`, the first parser `compound #> compound'` fails. Ths second parser succeeds, as the input string consists only of an alphabetic string of length greater or equal than two. The parsed constant `Constant "Bill"` gets

returned and back in the function `term'`, the constant gets pushed into the list of the compound, which results in the compound `Compound "Knows" [Variable 'Y', Constant "Bill"]`. This new compound gets passed to `term''`. The first parser fails, as there is no comma to parse, which means that the second parser gets invoked (i.e. the parser `return t`, which simply returns the term). The only thing that is parsed now is the closing paranthesis `)` by `compound'`, and the compound term `Compound "Knows" [Variable 'Y', Constant "Bill"]` is returned, which is what is the expected output.

8.2.7 Functional Aspects of the Parser

Data types Custom data types are used in the parser for the definition of the parser type `Parser` and for the definition of the type representing unification terms `Term`. The algebraic data type `Term` is defined such that it can take form as multiple variants: `Constant`, `Variable` and `Char`. Each variant is allowed to have associated values. This makes modelling tree-like structures, such as grammars for languages, particularly easy.

Most imperative programming languages do not support algebraic data types. If one were to implement the type `Term` in Java, one approach would be defining an abstract class or an interface named `Term` and create subclasses called `Constant`, `Variable` and `Compound`. This solution would however not be as simple as the one in Haskell and would require a lot more boilerplate code.

Higher-order functions The parser functions use higher-order functions extensively. Some of the basic parser operators, such as `(?)` and `>->`, take functions as arguments and process data using these functions.

Constructing functions using other functions The very nature of the parser is such that the functions are essentially building blocks. The parser is composed of many smaller parsing functions that combined result in a complex parser.

Pattern matching The parser functions make heavy use of pattern matching, which allows the code to be more concise. An implementation of the parser functions using if-statements would have resulted in less readable code. Another positive aspect to pattern matching is the possibility to match data types and their values, such as in the function `addTermToCompound` shown in listing 8.26, which matches the first pattern of a value of type `Term` with the variant `Compound` and the second pattern of everything else.

8.3 Unification Algorithm

8.3.1 Description of the Algorithm

The following pseudocode describes the unification algorithm our implementation is based on. The pseudocode is based on figure 9.1 from *Artificial Intelligence: A Modern Approach* by Russel and Norvig [12].

Algorithm 1 Unification algorithm

```
1: function UNIFY(x, y, θ = ∅)
2:     if θ = failure then return failure
3:     else if x = y then return θ
4:     else if ISVARIABLE(x) then return UNIFYVAR(x, y, θ)
5:     else if ISVARIABLE(y) then return UNIFYVAR(y, x, θ)
6:     else if ISCOMPOUND(x) and ISCOMPOUND(y) then
7:         return UNIFY(ARGS(x), ARGS(y), UNIFY(OP(x), OP(y), θ))
8:     else if ISLIST(x) and ISLIST(y) then
9:         return UNIFY(REST(x), REST(y), UNIFY(FIRST(x), FIRST(y),
   θ))
10:    else return failure
11:    end if
12: end function

13: function UNIFYVAR(var, x, θ)
14:    if {var/val} ∈ θ for some val then return UNIFY(val, x, θ)
15:    else if {x/val} ∈ θ for some val then return UNIFY(var, val, θ)
16:    else if OCCURCHECK(var, x) then return failure
17:    else return ADD({var/x}, θ)
18:    end if
19: end function
```

The variables x and y can be a variable, a term, a compound term or a list of compound terms. The variable θ is the initially empty substition that is used to hold the built-up $\{var/val\}$ pairs during the process of the algorithm. Summarized, the algorithm primarily involves matching syntax trees so that a successful match results in a unifier. Unsuccessful matches result in a $failure$. The functions used in the algorithm are as follows:

- The functions ISVARIABLE(x), ISCOMPOUND(x) and ISLIST(x) check whether the input is a variable, a compound or a list respectively.
- The function OP(x) takes the function name of a compound term and the function ARGS(x) takes the argument list of a compound term. For example, if the compound term is $T_2 = Knows(Y, Bill)$, then the

following holds:

$$\text{OP}(T_2) = Knows$$
$$\text{ARGS}(T_2) = (Y, Bill).$$

- The function FIRST(x) returns the first element of a list and the function REST(x) returns the rest of the list without the first element.
- The function OCCURCHECK(var, x) checks whether the variable var itself occurs inside the term x. If this is the case (i.e. if OCCURCHECK returns true), this means that no consistent unifier can be built and the algorithm must return $failure$.
- The function ADD($\{var/x\}, \theta$) adds the pair $\{var/x\}$ to the substitution θ.

8.3.2 Substitution Definition

A substitution represents the assignment of a variable to a term. This can be either a successful assignment or a failure, i.e. a failed unification attempt.

```
1 data Substitution = Substitution (Term, Term) | Failure
```

Listing 8.30: The Substitution data structure used in the Unification algorithm.

8.3.3 Main Unification Function

The `unify` function is the main part of the unification algorithm. It accepts two terms and a list of substitutions and returns an updated list of substitutions. This function has different cases depending on the structure of the terms to be unified.

```
1 unify :: Term -> Term -> [Substitution] -> [Substitution]
2 unify (Constant a) (Constant b) theta
3 | a == b = theta
4 | otherwise = [Failure]
5 unify (Variable var) x theta = unifyVar (Variable var) x
    theta
6 unify x (Variable var) theta = unifyVar (Variable var) x
    theta
7 unify (Compound m n) (Compound m' n') theta
8 | m == m' = unifyList n n' theta
9 | otherwise = [Failure]
```

```
10 unify _ _ _ = [Failure]
```

Listing 8.31: The main unification function.

8.3.4 Unification With a Variable

The `unifyVar` function handles unification when one of the terms is a variable. It behaves differently depending on the structure of the other term. If the other term is a constant or a variable, it adds the new substitution to the list. If the other term is a compound, it is delegated to the `unifyVarHelper` function.

```
1 unifyVar :: Term -> Term -> [Substitution] -> [
    Substitution]
2 unifyVar var x@(Constant _) theta
3 | varNotInTheta var theta = Substitution (var, x) : theta
4 | otherwise = [Failure]
5 unifyVar var x@(Variable _) theta = Substitution (var, x)
     : theta
6 unifyVar (Variable var) x theta = unifyVarHelper (
    Variable var) x theta
7 unifyVar _ _ _ = [Failure]
```

Listing 8.32: The unification function when a variable is involved.

8.3.5 Unification With a Variable and a Compound

The `unifyVarHelper` function is a helper function used to unify a variable with a compound term. It checks whether the variable occurs in the compound term (OccurCheck) to prevent infinite recursion. If the variable does not occur in the compound term, it inserts a new substitution into the list.

```
1 unifyVarHelper :: Term -> Term -> [Substitution] -> [
    Substitution]
2 unifyVarHelper var (Compound m n) theta
3 | varInTerm var (Compound m n) = [Failure] -- Occurs
    check
4 | otherwise = Substitution (var, applySubstitutions theta
     (Compound m n)) : theta
5 unifyVarHelper _ _ _ = [Failure]
```

Listing 8.33: The helper function for unifying a variable with a compound term.

8.3.6 Substitution Application

The `applySubstitutions` function is used to apply all substitutions in the list to a term. It applies the substitutions recursively to all subterms of a compound term.

```
1 applySubstitutions :: [Substitution] -> Term -> Term
2 applySubstitutions theta (Variable var) =
3 case lookupSubstitution (Variable var) theta of
4 [] -> Variable var
5 (term:_) -> term
6 applySubstitutions theta (Compound m n) =
7 Compound m (map (applySubstitutions theta) n)
8 applySubstitutions _ term = term
```

Listing 8.34: The function to apply substitutions to a term.

8.3.7 Unification of List of Terms

The `unifyList` function unifies two lists of terms. It calls the `unify` function for each pair of corresponding terms in the two lists. It is used to unify compound terms that are represented as lists of terms.

```
1 unifyList :: [Term] -> [Term] -> [Substitution] -> [
    Substitution]
2 unifyList [] [] theta = theta
3 unifyList (t:ts) (t':ts') theta = unifyList ts ts' (unify
     t t' theta)
4 unifyList _ _ _ = [Failure]
```

Listing 8.35: The function to unify a list of terms.

8.3.8 Functional Aspects of the Algorithm

Data types The data type `Substitution` represents a single substitution entry inside θ. It can either take form as a tuple of two terms, or as a failure label. This is useful as the Haskell functions can then be defined to match certain patterns based on the type variant of `Substitution`. Similarly to the paragraph "Data types" in the section "Functional Aspects of the Parser", the closest Java equivalent would be the `Optional<SubstitutionPair>` class, where `SubstitutionPair` is another class that models a tuple of two terms. A Java implementation would contain a lot more boilerplate code in comparison to this Haskell implementation.

Pattern matching Pattern matching is used very extensively in the unification algorithm for matching certain types to certain results, especially the `Term` variants. The pattern matching inside the Haskell functions partially performs the job of the functions IsVariable, IsCompound and IsList in the pseudocode.

Recursion The algorithm itself is defined recursively, which is reflected in the Haskell implementation.

8.4 Conclusion

The implemented parser for unification terms is based on parser combinators, i.e. smaller parsing functions that are combined to create larger more complex parsers. It parses input strings using a recursive-descent approach. The resulting parser code strongly ressembles the grammar for unification terms and can easily be extended.

The implemented unification algorithm is based on the description of unification in *Artificial Intelligence: A Modern Approach* by Russel and Norvig [12] and unifies terms in a recursive fashion. The implementation relies on pattern matching and recursion, both central concepts of functional programming.

Chapter 9

Project Management and Organisation

9.1 Overview

For our project, we were highly advised to utilize the *Scrum* methodology. Scrum is an Agile framework, which is commonly used in project management due to its emphasis on the iterative workflow. It allows for the decomposition of large and complex tasks into smaller, manageable units, which significantly enhances the ability to monitor progress closely.

9.2 Sprints

Our implementation of the Scrum framework involved the use of sprints, which we at first decided to make two weeks long. Over time, we decided to work with three week long sprints, as we found the longer duration to be more suitable for us.

During our sprint meetings, we reflected on our progress and set strategies for the next steps to move our project forward. The most important information and further steps that were discussed were noted down in protocols, so that every party was able to look up what was discussed in each meeting.

The distribution of the requirements into the sprints is shown in table 9.1:

Sprint	Topics	Time frame
1	Recursion, pattern matching	SW4 - SW7
2	Advanced function concepts, lazy evaluation, Game of Nim	SW7 - Break
3	Advanced types	SW9 - SW11
4	Unification parser, unification algorithm, results	SW12 - SW16

Table 9.1: Distribution of our main requirements into the sprints. SWx denotes semester week x.

9.3 Roles

Our scrum roles were as follows:

- Product Owner: O. Biberstein
- Scrum Master: A. Tabakovic
- Developer: A. Grand, E. Ingold, A. Tabakovic

Since learning is an individual process and each of us spent a lot of time acquiring knowledge independently, we mainly collaborated on the implementation of the project. The tasks were divided among the three developers and thus ensured a balanced distribution of work.

We used GitLab to organise our user stories, product backlog and sprints, following the recommendations found in a blog post by GitLab titled *How to use GitLab for Agile software development* [13]. However, our approach differed from the traditional Scrum methodology as we did not hold daily Scrum meetings due to the more sporadic nature of the project.

9.4 Conclusion

Following the Scrum methodology for our project presented us with a number of unique challenges. We initially decided to align the chapters with the sprints, but soon realised that the uneven distribution of chapters led to an imbalance of workload in the sprints. This resulted in some sprints remaining incomplete while others had extra time.

Despite these challenges, we tried to incorporate Scrum principles as much as possible. Overall, the experience was satisfying and served as a learning opportunity for us.

After our midterm presentation, we received valuable feedback from our Scrum tutor, which led to the strategic decision to use Scrum exclusively for the programming aspect of our project. We split the documentation and main parts of the project trilaterally and worked highly collaboratively,

which proved to be very effective. While the meeting protocols served as the main source of our requirements to implement, we still created the artifacts on GitLab to stay consistent and have a general overview of what needs to be done by whom. This adapted approach helped us to be more efficient and encouraged our progress.

Chapter 10

Conclusion

10.1 Looking Back

Looking back on our efforts in this project, we can say that we are very satisfied with how our project has turned out. Although progressing smoothly for the most part, we encountered some difficulties, particularly in integrating Scrum into this project. The application of Scrum was unorthodox, given the nature of the project. This presented us with some challenges. However, the experience was invaluable in showing us how methods like Scrum can be adapted and used in a wide range of contexts.

Learning Haskell as newcomers to the functional programming world has been quite challenging but also immensly satisfying. It gave us a new perspective of what is possible to do in terms of programming. A trend in functional programming concepts being introduced to mainstream programming languages has been growing as of lately, which means that the concepts learned are applicable to other programming languages as well.

In summary, this project provided a platform for a deeper and practical understanding of Haskell and functional programming, preparing us for future explorations in the field of functional programming.

10.2 Looking Forward

Looking at further applications, we see several potential extensions for this project.

Advanced concepts One way this project could be extended is by introducing more advanced topics of Haskell and functional programming, such as monads, concurrency, lambda calculus, and more.

More algorithms Another way this project could be extended is by implementing more algorithms that origin from the theoretical side of com-

puter science, such as those appearing in the book *Artificial Intelligence: A Modern Approach* [12]. Additionally, performance comparisons between the reference implementations of the book and potential implementations in Haskell could be performed to show the differences in efficiency.

More parsers Haskell is well-suited for developing parsers, which was shown in the unification subproject. This project could be extended by developing more complex parsers, maybe even interpreters/compilers for programming languages, such as Logo or Lisp. Another aspect that could be studied more into depth is the parsing library *Parsec* for Haskell.

Server-side development Server-side web-based development is a field which many software developers work in. Therefore, it could be interesting to see how Haskell can be used to develop server-side software that needs to handle requests from the internet. Another aspect that could be looked into is what kind of web frameworks exist for Haskell and how one can get started with them.

Discovering Elm Elm [14] is a functional programming language that compiles to JavaScript, whose syntax however is strongly based on Haskell. This makes it a suitable programming language for developing interactive web-based user interfaces and an interesting potential topic for further consideration.

Bibliography

[1] Dr. Olivier Biberstein. BTI7544 Functional Programming in Scala. Lecture notes, 2023.

[2] Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley Longman Publishing Co., Inc., USA, 2nd edition, 1999.

[3] Graham Hutton. *Programming in Haskell*. Cambridge University Press, USA, 1st edition, 2007.

[4] Glasgow haskell compiler. URL: `https://www.haskell.org/ghc/`.

[5] Data.String. URL: `https://hackage.haskell.org/package/base-4.18.0.0/docs/Data-String.html`.

[6] Type signatures as good style. URL: `https://wiki.haskell.org/Type_signatures_as_good_style`.

[7] Data.List. URL: `https://hackage.haskell.org/package/base-4.18.0.0/docs/Data-List.html`.

[8] *The Fibonacci sequence*. HaskellWiki. URL: `https://wiki.haskell.org/The_Fibonacci_sequence`.

[9] Michael T. Goodrich and Roberto Tamassia. *Data structures and algorithms in Java*. Wiley Publishing, 3rd edition, 2003.

[10] Lennart Andersson. Parsing with haskell. 2001.

[11] parsec: Monadic parser combinators. URL: `https://hackage.haskell.org/package/parsec`.

[12] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson, 4 edition, 2021.

[13] How to use GitLab for Agile software development. URL: `https://about.gitlab.com/blog/2018/03/05/gitlab-for-agile-software-development/`.

[14] Elm - delightful language for reliable web applications. URL: `https://elm-lang.org/`.