Duale Hochschule
Baden-Württemberg
Mannheim

Functional Programming in *Haskell*

— Winter 2025 —

Prof. Dr. Karl Stroetmann

February 27, 2025

Danger **Work in Progress!** **Danger**

Contents

Abstract

This is a very short introduction to functional programming with *Haskell*. It is not intended to be a Haskell course. My intention merely is for the reader to get a taste of what programming in *Haskell* feels like. If I succeed in convincing the reader that *Haskell* is a programming language that is both usefull and mind extending, then I consider my job done. Therefore, this short paper will just present some of the highlights of the programming language *Haskell* via examples that I hope the reader finds intriguing enough so that she feels inclined to read some of the outstanding books introducing Haskell in more depth. There are three books that I recommend for those readers who want to understand *Haskell* in more depth:

1. *Programming in Haskell*

   by Graham Hutton [Hut16]. This book has some nice YouTube videos.

2. *Haskell: The Craft of Functional Programming*

   by Simon Thompson [Tho11]. This book is available free online at

   https://www.haskellcraft.com.

3. *Learn You a Haskell for Great Good!*

   by Miran Lipovača [Lip11]. This book is available free online at

   https://learnyouahaskell.com.

4. *Effective Haskell: Solving Real-World Problems with Strongly Typed Functional Programming*

   by Rebecca Skinner [Ski23].

Chapter 1

Introduction

In this introduction I will do two things:

1. First, I discuss those features of Haskell that set Haskell apart from other programming languages.
2. Second, I will present a few short example programs that give a first taste of Haskell.

1.1 Why Haskell is Different

Before we present any details of *Haskell*, let us categorize this programming language so that we have an idea about what to expect. *Haskell* has the following properties:

1. *Haskell* is a functional programming language.

   A functional programming language is any programming language that treats functions as first class citiziens:

   (a) A function can be given as an argument to another function.
   (b) A function can be produce a function as its result.

   A well known programming language that supports functional programming is *Python* and there are several books discussing functional programming in *Python*, e.g. [Lot22], [Mer15], and [Rei23].

2. *Haskell* is statically typed.

   Every variable in Haskell has a fixed type, which can not be changed. In this respect, *Haskell* is similar to the programming language *Java*. However, in contrast to *Java*, we do not have to declare the type of every variable and every function because most of the time the type of a variable can be inferred by the type system. Therefore, in Haskell we usually specify only the types of non-trivial functions.

   The benefit of this approach is that many type errors will already be caught by the compiler. This is in contrast to programs written in a dynamically typed language like *Python*, where type errors are only discovered at runtime.

3. *Haskell* is a pure functional programming language.

   Once a variable is assigned a value, this value can not be changed. For example, if we want to sort a list, we are not able to change the list data structure. All we can do is to compute a new list which contains the same elements as the old list and which, furthermore, is sorted.

   The property of being a pure language sets *Haskell* apart from most other programming languages. Even the language *Scala*, which is designed as a modern functional programming language, is not pure.

   What is the big deal about purity? On one hand, it forces the user to program in a declarative style. Although, in general, nobody likes to be forced to do something, there is a huge benefit in pure programming.

   (a) In a pure programming language, functions will always return the same result when they are called with the same arguments. This property is called referential transparency. This makes reasoning about code easier, as you can replace a function call with its result without changing the behavior of the program. Therefore the correctness of functions can be verified mathematically.

   (b) Since pure functions do not depend on or modify external state, their behavior is entirely predictable. The advantage is that testing becomes straightforward because functions can be tested in isolation without worrying about interactions with other parts of the system.

   (c) Compilers for pure languages can make aggressive optimizations, such as caching function results (memoization) or reordering computations, because they know that functions are side-effect-free. The advantage is that programs can often run faster.

   (d) Furthermore, concurrency becomes much easier to manage when functions do not use global variables and do not change their arguments.

4. *Haskell* is a compiled language similar to *Java* and `C`, but additionally offers an interpreter. Having an interpreter is beneficial for rapid prototyping. The property that *Haskell* programs can be compiled ensures that the resulting programs can befaster than, for example, *Python* programs.

5. *Haskell* is a lazy language. The programming language `C` is an eager language. If an expression of the form

   $$f(a_1, \cdots, a_n)$$

   has to be evaluated, first the subexpressions $a_1, \cdots a_n$ are evaluated. Let us assume that a_i is evaluated to to value x_i. The, $f(x_1, \cdots, x_n)$ is computed. This might be very inefficient. Consider the example shown in Figure 1.1 on page 3.

   Let us assume that the expression

   ```
   f(h(0), g(1))
   ```

   needs to be evaluated and that the computation of `g(1)` is very expensive. In a `C`-program, the expressions and `h(0)` and `g(1)` will be both evaluated. If it turns out that `h(0)` is 2, then the evaluation of `g(1)` is not really necessary. Nevertheless, in `C`

```c
int f(int x, int y) {
    if (x == 2) {
        return 42;
    }
    return 2 * y;
}
```

Figure 1.1: A `C` program.

this evaluation takes place because `C` has an eager evaluation strategy. In contrast, an equivalent *Haskell* would not evaluate the expression `h(0)` and hence would be much more efficient.

6. *Haskell* is difficult to learn.

   You might ask yourself why *Haskell* hasn't been adopted more widely. After all, it has all these cool features mentioned above. The reason is that learning *Haskell* is a lot more difficult then learning a language like *Python* or *Java*. There are two reasons for this:

   (a) First, *Haskell* differs a lot from those languages that most people know.

   (b) In order to be very concise, the syntax of *Haskell* is quite different from the syntax of established programming languages.

   (c) *Haskell* requires the programmer to think on a very high level of abstraction. Many students find this difficult.

   (d) Lastly, and most importantly, *Haskell* supports the use of a number of concepts like, e.g. functors and monads from category theory. It takes both time and mathematical maturity to really understand these concepts.

   If you really want to understand the depth of *Haskell*, you have to dive into those topics. That said, while you have to understands both functors and monads, you do not have to understand category theory.

   (e) Fortunately, it is possible to become productive in *Haskell* without understanding functors and monads. Therefore, this lecture will focuss on those parts of *Haskell* that are more easily accessible.

1.2 A First Taste of Haskell

Computing All Prime Numbers

A prime number is a natural number p that is different from 1 and that can not be written as a product of two natural numbers a and b that are both different from 1. If we denote the set of all prime numbers with the symbol $\mathbb{P}$, we therefore have:

$$\mathbb{P} = \bigl\{p \in \mathbb{N} \mid n \neq 1 \wedge \forall a, b \in \mathbb{N} : (a \cdot b = p \implies a = 1 \vee b = 1)\bigr\}$$

An efficient method to compute the prime numbers is the sieve of Eratosthenes. This is an algorithm used to find all primes up to a given number. The method works by iteratively marking the multiples of each prime number starting from 2. The numbers which remain unmarked at the end of the process are the prime numbers. For example, consider the list of integers from 2 to 30:

$$2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30.$$

We will now compute the set of all primes less or equal than 30 using the Sieve of Eratosthenes.

1. The first number in this list is 2 and hence 2 is prime:

   $$\mathbb{P} = \{2, \cdots\}.$$

2. We remove all multiples of 2 (i.e., 2, 4, 6, 8, ..., 30). This leaves us with the list

   $$3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29.$$

   The first remaining number 3 is prime. Hence we have

   $$\mathbb{P} = \{2, 3, \cdots\}$$

3. We remove all multiples of 3 (i.e., 3, 6, 9, 12, ..., 27). Note that some numbers (like 6 or 12, for instance) may already have been removed. This leaves us with the list

   $$5, 7, 11, 13, 17, 19, 23, 25, 29.$$

   The first remaining number 5 is prime. Hence we have

   $$\mathbb{P} = \{2, 3, 5, \cdots\}.$$

4. We remove all multiples of 5 (i.e., 5, 10, 15, 20, ...). This leaves us with the list

   $$7, 11, 13, 17, 19, 23, 29.$$

   The first remaining number 7 is prime. Hence we have

   $$\mathbb{P} = \{2, 3, 5, 7, \cdots\}.$$

5. Since the square of 7 is 49, which is greater that 30, there is no need to remove the multiples of 7, since all multiples $a \cdot 7$ for $a < 7$ have already been removed and all multiples $a \cdot 7$ for $a \geq 7$ are greater than 30. Hence the remaining numbers are all prime and we have found that the prime numbers less or equal than 30 are:

   $$\{2, 3, 5, 7, 11, 13, 17, 19, 23, 29\}.$$

Figure 1.2 on page 5 shows a Python script that implements the Sieve of Eratosthenes to compute all prime numbers up to a limit `n`. This script first initializes the list `is_prime` to track prime status. It then iteratively marks the multiples of each prime number, finally collecting and printing all numbers that remain marked as prime. This script implements one optimization: If `p` is a prime, then only the multiples of `p` that have the form $a \cdot \texttt{p}$ with $a \geq \texttt{p}$ have to be removed from the list, since a product of the form $a \cdot \texttt{p}$ with $a < \texttt{p}$ has already been removed when removing multiples of a or, if a is not prime, of multiples of whatever prime is contained in a.

Figure 1.3 on page 5 shows a *Haskell* program to compute **all** primes. Yes, you have read that correct. It doesn't compute the primes up to a given number, but rather it computes <u>all</u> primes.

```
def sieve_of_eratosthenes(n):
    """Return a list of prime numbers up to n (inclusive)."""
    if n < 2:
        return []
    is_prime = [True] * (n + 1)
    is_prime[0] = is_prime[1] = False  # 0 and 1 are not primes
    p = 2
    while p * p <= n:
        if is_prime[p]:
            for i in range(p * p, n + 1, p):
                is_prime[i] = False
        p += 1
    return [i for i, prime in enumerate(is_prime) if prime]
```

Figure 1.2: A *Python program to compute the prime numbers up to* n.

```
1  primes :: [Integer]
2  primes = sieve [2..]
3
4  sieve :: [Integer] -> [Integer]
5  sieve (p:ns) = p : sieve [n | n <- ns, mod n p /= 0]
```

Figure 1.3: Computing the prime numbers.

The first thing to note is that line 1 and line 4 are type annotations. They have only been added to aid us in understanding the program. If we would drop these lines, the program would still work. Hence, we have an efficient 2-line program to compute the prime numers. Lets discuss this program line by line.

(a) Line 1 states that the function `primes` returns a list of `Integer`s. `Integer` is the type of all arbitrary precision integers. The fact that we have enclosed the type name `Integer` in the square brackets "`[`" and "`]`" denotes that the result has the type *list* of `Integer`.

(b) In line 2, the expression "`[2..]`" denotes the list of all integers starting from 2. Since *Haskell* is lazy, it is able to support infinite data structures. The trick is that these lists are only evaluated as much as they are needed. As long as we do not inspect the complete list, everything works fine.

(c) Line 2 calls the function `sieve` with the argument `[2..]`. *Haskell* uses prefix notation for calling a function. If f is a function an a_1, a_2, and a_3 are arguments of this function, then the invocation of f with these arguments is written as

$$f\ a_1\ a_2\ a_3$$

Note that the expression

$$f(a_1, a_2, a_3)$$

denotes something different: This expression would apply the function f to a single argument, which is the triple (a_1, a_2, a_3).

(d) Line 4 declares the type of the function `sieve`. This function takes one argument, which is a list of `Integer`s and returns a list of `Integer`s.

(e) Line 5 defines the function `sieve` that takes a list of numbers l that has the following properties:

- The list l is a sorted ascendingly.
- If p is the first element of the list l, then p is a prime number.
- The list l does not contain multiples of any number q that is less than p:

$$\forall q \in \mathbb{N} : \bigl(q < p \rightarrow \forall n \in \mathbb{N} : n \cdot q \notin l\bigr).$$

Given a list l with these properties, the expression

```
sieve l
```

returns a list of all prime number that are greater or equal than p, where p is the first element of l:

$$\texttt{sieve}\, l = [\, q \in l \mid q \in \mathbb{P}\,].$$

Hence, when `sieve` is called with the list of all natural number greater or equal than 2 it will return the list of all prime numbers, since 2 is the smallest prime numer.

There is a lot going on in the definition of the function `sieve`. We will discuss this definition in minute detail.

1. The function `sieve` is defined via matching. We will discuss matching in more detail in the next chapter. For now we just mention that the expression

   ```
   (p:ns)
   ```

   matches a list with first element `p`. The remaining elements are collected in the list `ns`. For example, if `l = [2..]`, i.e. if l is the list of all natural numbers greater or equal than 2, then the variable `p` is bound to the number 2, while the variable `ns` is bound to the list of all natural numbers greater or equal than 3, i.e. `ns = [3..]`.
2. The right hand side of the function definition, i.e. the part after the symbol "`=`" defines the value that is computed by the function `sieve`. This value is computed by calling the function "`:`", which is also known as the cons function because it constructs a list. The operator "`:`" takes two arguments. The first argument is a value `u` of some type a and the second argument `us` is list of elements of the same type a. An expression of the form

   ```
   u : us
   ```

   then returns a list where `u` is the first elements and `us` are the remaining elements. For example, we have

   ```
   1 : [2, 3, 4] = [1, 2, 3, 4].
   ```

3. The recursive invocation of the function sieve takes a list comprehension as ist first argument. the expression

   ```
   [n | n <- ns, mod n p /= 0]
   ```

   computes the list of all those number `n` from the list `ns` that have a non-zero remainder when divided by the prime number `p`, i.e. this list contains all those number from the list `ns` that are not multiples of `p`. There are two things to note here concerning the syntax of *Haskell*:

   - Functions are written with prefix notation. For example, we first write the function name `mode` followed by the arguments `m` and `p`. In *Python* the expression

     ```
     mod m p
     ```

     would have been written as `m % p`.
   - The operator "`/=`" expresses inequality, i.e. the *Haskell* expression

     `a /= b`   would be written as   `a != b`

     in the programming language *Python*.

   Putting everything together, the *Haskell* expression

   ```
   [n | n <- ns, mod n p /= 0]
   ```

   is therefore equivalent to the *Python* expression

   ```
   [n for n in ns if n % p != 0].
   ```

Chapter 2

Types, Expressions, and Functions

In this Chapter we will give a more systematic overview of *Haskell*. In particular, we define

1. types,
2. expressions, and
3. functions.

At this point you might wonder why we don't also discuss statements and control structures. The reason is simple: There are no statements in *Haskell*. Everything is an expression. Neither are there control structures.

Haskell is a statically typed, purely functional programming language known for its expressive type system and emphasis on immutability. In *Haskell*, types play a central role in the design and implementation of programs. This section discusses the primitive types available in *Haskell*, their properties, and how they form the building blocks for more complex types in the language.

Primitive types in *Haskell* refer to the basic types that are built into the language and directly supported by the compiler. These types include numbers, characters, booleans, and the unit type. Understanding these types is crucial for both writing correct programs and for taking full advantage of *Haskell*'s type system.

2.1 Primitive Types

Haskell provides several predefined types, each serving different needs with respect to performance, precision, and range. The main numeric types in *Haskell* include:

(a) `Int` is a fixed-precision integer type, which means that its range is limited by the underlying hardware. Typically, `Int` is implemented as a 32-bit or 64-bit integer. Its limited range means that operations on very lar ge numbers may result in overflow.

Example Usage:

```haskell
-- Defining an integer of type Int
smallInt :: Int
smallInt = 42

```

(b) `Integer` represents arbitrary-precision integers. This means that there is no fixed upper bound on the size of an `Integer` value, though operations may become slower as numbers grow larger.

Example Usage:

```haskell
-- Defining an integer of type Integer
bigInt :: Integer
bigInt = 123456789012345678901234567890
```

(c) `Float` is the type of single-precision floating-point numbers. While it may be faster and uses less memory, its precision is limited compared to `Double`.

Example Usage:

```haskell
-- Defining a floating-point number of type Float
singlePrecision :: Float
singlePrecision = 3.14159
```

(d) `Double` is a double-precision floating-point number, offering more precision at the cost of additional memory and potentially slower computation in some contexts.

Example Usage:

```haskell
-- Defining a floating-point number of type Double
doublePrecision :: Double
doublePrecision = 2.718281828459045
```

(e) `Char` is used to represent single Unicode characters. Characters in *Haskell* are enclosed in single quotes.

Example Usage:

```haskell
-- A character literal
letterA :: Char
letterA = 'A'
```

(f) `String` is used to represent Unicode strings. In *Haskell*, strings are enclosed in double quotes.

Example Usage:

```haskell
-- A string literal
letterA :: Char
letterA = "Hello, World!"
```

In *Haskell* the type `String` is an alias for the type `[Char]`, which represents a list of characters.

(g) `Bool` represents boolean values. It has two possible values: `True` and `False`.

Example Usage:

```
-- A boolean literal
isHaskellFun :: Bool
isHaskellFun = True
```

(h) The unit type is denoted by `()`. This type has exactly one value, which is also written as `()`. It is analogous to the concept of *None* in *Python*. It is typically used when a function does not need to return any meaningful value.

Example Usage:

```
-- A function that returns the unit type
printMessage :: String -> ()
printMessage msg = putStrLn msg
```

One of *Haskell*'s powerful features is its ability to perform type inference. For example, numeric literals in *Haskell* are polymorphic, i.e. they do not have a fixed type but rather a type class. This concept will be discussed in a subsequent chapter. For example, a literal such as `5` can be interpreted as an `Int`, an `Integer`, a `Float`, or a `Double`, depending on the context. This is achieved via type classes such as `Num`.

Example:

```
-- The literal 5 is polymorphic and can be any type that is an instance of Num
    .
polymorphicExample :: Num a => a
polymorphicExample = 5
```

2.2 Composite Types: Lists and Tuples

Haskell provides several composite types that allow for the grouping of values. Two of the most commonly used composite types are **lists** and **tuples**. Both types enable the construction of complex data structures by combining simpler types, yet they serve different purposes and have distinct characteristics.

2.2.1 Lists

A **list** in *Haskell* is an homogeneous ordered collection of elements. The fact that a list is homogeneous means that all of elements must be of the same type. Lists are one of the most fundamental data structures in *Haskell* and are used extensively for processing sequences of data. Lists are denoted using square brackets, with elements separated by commas. For example, the list containing the integers 1, 2, and 3 is written as:

```
[1, 2, 3]
```

If `a` is any type, then the type of a list of type `a` is written as:

$$[a]$$

Lists are the workhorse of many functional languages and *Haskell* is no exception. Therefore, *Haskell* provides a rich set of functions for processing lists, such as `map`, `filter`, `foldr` and `foldl`. These functions will be discussed later after we have discussed the syntax of functions.

Lists are implemented as linked lists, which means that operations such as prepending an element (using the `:` operator) are very efficient. For example:

```
-- Prepending 0 to an existing list:
numbers :: [Int]
numbers = 0 : [1, 2, 3]  -- results in [0, 1, 2, 3]
```

However, other operations have a linear complexity. For example, finding the length of a list has a linear complexity because the whole list needs to be traversed. This is in contrast to the programming language *Python*, where lists are implemented as dynamic arrays. Hence, in *Python*, finding the length of a list has complexity $\mathcal{O}(1)$.

Pattern matching on lists is a powerful feature in *Haskell*. One common idiom is to match against the empty list `[]` or a cons cell `(x:xs)`, where `x` is the head of the list and `xs` is the tail. For example:

```
sumList :: Num a => [a] -> a
sumList []     = 0
sumList (x:xs) = x + sumList xs
```

This recursive definition demonstrates how lists lend themselves naturally to inductive processing.

There is another very important difference between lists in *Python* and lists in *Haskell*: In *Haskell*, lists are immutable, i.e. once we have constructed a list, there is no way to change an element in this list. This is similar to *tuples* in *Python*.

2.2.2 Tuples

In contrast to a list, a **tuple** is a composite type that can hold a fixed number of elements, which may be of different types. Tuples are written using parentheses, with elements separated by commas. For instance, the tuple:

```
("Alice", 30, True)
```

contains a `String`, an `Int`, and a `Bool`. The type of this tuple is written as:

`(String, Int, Bool)`

In general, the type of a tuple is denoted as:

$$(\mathtt{t}_1, \mathtt{t}_2, \cdots, \mathtt{t}_n)$$

Here, $\mathtt{t}_i$ is the type of the i^{th} component.

Unlike lists, tuples are heterogeneous, i.e. the elements can be of different types. When declaring the type of a tuple, the size, i.e. the number of elements, is implicitly also defined. Tuples are immutable. In fact, every data structure in *Haskell* is immutable.

Tuples are particularly useful when you need to group a set of values that naturally belong together, such as coordinates or key-value pairs. Tuples support pattern matching, which allows functions to easily deconstruct them. For example, a function that extracts the first element of a pair can be defined as:

```
first :: (a, b) -> a
first (x, _) = x
```

Similarly, functions can be defined to operate on larger tuples by matching each component:

```haskell
describePerson :: (String, Int, Bool) -> String
describePerson (name, age, isEmployed) =
  name ++ " is " ++ show age ++ " years old and " ++
  (if isEmployed then "employed" else "unemployed") ++ "."
```

The previous example is easy to misunderstand because the function `describePerson` receives not three elements but rather one element, which is a tuple of three elements.

2.3 Haskell Expressions and Operators

In *Haskell*, every construct is an expression that evaluates to a value. Unlike imperative languages where statements perform actions, in *Haskell* even control constructs such as conditionals and pattern matching yield results. One of the most powerful features of *Haskell* is its flexible and composable syntax for expressions, which is largely governed by a rich set of infix operators. These operators come with fixed precedences and associativities that determine the order in which parts of an expression are evaluated. In this section we present a detailed discussion of *Haskell*'s operators, their precedence, associativity, and how they interact within expressions. We will also provide a comprehensive table of many common operators, together with examples to illustrate their usage.

Before examining the operators, it is important to recall that *Haskell* function application (i.e., writing `f x` to apply the function `f` to the argument `x`) has the highest precedence of all operations. This means that in an expression like `f x + y`, the application of `f` to `x` is performed first, and then the addition is carried out.

Operator Precedence and Associativity

The precedence of an operator indicates how tightly it binds to its operands. Operators with higher precedence are applied before operators with lower precedence. Associativity, on the other hand, determines how operators of the same precedence are grouped in the absence of explicit parentheses. For example, left-associative operators group from the left. For example,

`a - b - c`

is interpreted as

`(a - b) - c`).

Right-associative operators group from the right. For example

`a ^ b ^ c`

is interpreted as

`a ^ (b ^ c)`).

Non-associative operators cannot be chained without explicit parentheses. The following table lists many of the common operators in *Haskell*, along with their default precedences and associativities. (Note that these declarations can be found in the standard libraries and GHC documentation, and some operators may have additional variants defined by specific libraries.)

Operator	**Precedence**	**Associativity**	**Description and Example**
function application	10	N/A	`f x y` applies `f` to `x` and `y`. Example: `sum [1,2,3]`
`.`	9	right-associative	function composition. Example: `(f . g) x = f (g x)`
`!!`	9	left-associative	list indexing. Example: `[10,20,30] !! 1 = 20`
`^`	8	right-associative	power of natural numbers. Example: `2 ^ 3 = 8`
`^^`	8	right-associative	floating point exponentiation Example: `9 ^^ (-0.5) = 3.0`
`*`, `/`,	7	left-associative	multiplication, floating point division Example: `6 * 7 = 42`
`` `div` ``, `` `mod` ``	7	left-associative	integer division/modulus. Example: `mod 8 3 = 2`
`+`, `-`	6	left-associative	addition and subtraction.
`++`	5	right-associative	List concatenation. Example: `[1,2] ++ [3] = [1,2,3]`
`:`	5	right-associative	Cons operator for lists. Example: `1 : [2,3] = [1,2,3]`
`==`, `/=`	4	non-associative	relational operators.
`<`, `<=`, `>`, `>=`	4	non-associative	relational operators
`<$>`	1	left-associative	map operator for functors
`<*>`	1	left-associative	apply operator for applicatives
`&&`	3	right-associative	Boolean **and**.
`||`	2	right-associative	Boolean **or**
`>>=`	1	left-associative	bind operator for monads
`>>`	1	left-associative	sequencing operator for monads
`$`	0	right-associative	function application operator. Example: `f $ x + y = f (x + y)`

There are many more operators in *Haskell*. Furthermore, we can define our own operators. This is much more powerful than the concept of operator overloading that we have in *Python*. We will discuss the details later after we have discussed the definition of functions. Finally, we can use functions as infix operators if we enclose them in a pair of back-quote symbols "`". For example, `div` is a function performing integer division, but instead of writing

`div 8 3` we can instead write `` 8 `div` 3 ``.

It is important to understand that function application has the highest precedence. For example, in the expression

```
f x + y
```

the function `f` is applied to `x` before adding `y`. If we want the addition to be part of the argument to `f`, we have to use parentheses as follows:

```
f (x + y).
```

Alternatively, we can use the dollar-operator and write

```
f $ x + y.
```

This left-grouping is common for arithmetic operators and ensures consistency with standard arithmetic evaluation.

The operator `$` is particularly useful because of its very low precedence. It allows the programmer to write expressions without a multitude of parentheses. For example, consider:

```
print $ sum $ map (\x -> x * 2) [1,2,3].
```

Without `$`, the same expression would require nested parentheses:

```
print (sum (map (\x -> x * 2) [1,2,3])).
```

By declaring `$` as having a precedence of 0 and being right associative, *Haskell* ensures that all other operators bind more tightly, so the expression to the right of `$` is completely grouped before being passed as an argument.

List operations provide a good demonstration of both precedence and associativity. The cons operator `:` is right-associative, so

```
1 : 2 : 3 : []
```

is interpreted as

```
1 : (2 : (3 : [])).
```

Of course, this just denotes the list $[1, 2, 3]$. Th e right-associativity of the cons operator is essential for constructing lists.

Relational operators, such as `==`, `<`, and `>=`, are declared as non-associative so that expressions like

```
a < b < c
```

are not allowed without parentheses. This design choice prevents ambiguous chaining of comparisons; instead, the programmer must explicitly write

```
(a < b) && (b < c)
```

to test whether `b` lies between `a` and `c`.

Attention: There is a snag when dealing with negative numbers. If `f` is a function that takes one argument of type `Integer` and we want to call it with a negative number, for example with `-42`, then we can not write the following:

```
f -42
```

The reason is that *Haskell* interprets this as an expression where `42` is subtracted from `f`. The correct way to call `f` with an argument of `-42` is therefore to write the following:

```
f (-42)
```

2.4 Defining Functions in *Haskell*

In *Haskell*, functions are first-class citizens and form the backbone of the language. Unlike imperative languages where functions might be seen merely as procedures or routines, in *Haskell* every function is a pure mapping from inputs to outputs. In this context, the word pure is a technical term that means that the function has no side effects, i.e. it cannot change any variables or perform input or output, unless it is specifically declared to be an `IO` function.

This section provides an in-depth exploration of function definitions in *Haskell*. First, we discuss the basic syntax of function definitions and their type signatures. After that we discuss matching, guards, higher-order functions, currying, lambda expressions, recursion, and polymorphism.

2.4.1 Introduction to Function Definitions

At its core, a function in *Haskell* is defined by a name, a set of parameters, and an expression that computes the result. The simplest form of a function definition is:

```
square :: Integer -> Integer
square x = x * x
```

Here, `square` is a function that takes a number `x` of type `Integer` and returns `x * x`.

Every function has a type, and while the compiler is capable of inferring types, it is a good practice to include explicit type signatures. Consider:

```
add :: Integer -> Integer -> Integer
add x y = x + y
```

The type signature of `add` tells us that it takes two integers and returns an integer. Type signatures serve as a form of documentation and help catch errors during compilation.

Haskell's type inference system can often deduce the type without explicit signatures. For instance, writing:

```
multiply x y = x * y
```

allows the compiler to infer that `multiply` has a type compatible with `Num a => a -> a -> a`. Here, `Num` is a so called type class and the type signature

`Num a => a -> a -> a`

tells us that if we have two arguments `x` and `y` of type `a` where the type `a` is an instance of the type class `Num`, then the expression `multiply x y` will again have the type `a`. The notion of a type class is an advanced concept that will be discussed later. Although type inference is possible, explicit type signatures are recommended for readability.

2.4.2 Basic Function Syntax

A function definition in *Haskell* follows the general form:

$$\texttt{functionName arg}_1\ \texttt{arg}_2\ \dots\ \texttt{arg}_N = \texttt{expression}$$

Functions can have multiple parameters, and the absence of parentheses around the parameters emphasizes that *Haskell* functions are curried by default. The concept of currying will be discussed now. Consider the following example:

```haskell
add :: Int -> Int -> Int
add x y = x + y
```

This definition can be interpreted stating that the functions `add` takes an integer `x` and returns a new function that itself takes an integer `y` as its argument and returns an integer. For clarity, we could have written the type signature of `add` as follows:

```
add ::  Int -> (Int -> Int)
```

This notation emphasizes that `add` takes and integer and returns a function of type `Int -> Int`. The operator `->` is right associative and hence the types

`Int -> Int -> Int` and `Int -> (Int -> Int)`

are the same.

2.4.3 Pattern Matching in Function Definitions

Pattern matching is a fundamental mechanism in *Haskell* for deconstructing data. It allows functions to perform different computations based on the structure of their inputs. Consider the definition of the factorial function:

```haskell
factorial :: Integer -> Integer
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

Here, the pattern `0` directly matches the base case. Pattern matching can be used with more complex data types such as lists and tuples.

For example, here is a function that computes the length of a list:

```haskell
listLength :: [a] -> Int
listLength []     = 0
listLength (_:xs) = 1 + listLength xs
```

In this example, the empty list `[]` is matched by the first clause, while the pattern `(_:xs)` matches any non-empty list, ignoring the head element and recursively processing the tail. In this pattern, the underscore _ denotes the so called anonymous variable. This is the same as in *Python*.

2.4.4 Guards and Conditional Function Definitions

Guards offer an alternative way to define functions that behave differently based on Boolean conditions. Instead of writing multiple equations with pattern matching, guards allow for a more readable, condition-based approach. For example, a function to compute the absolute value:

```haskell
absolute :: Integer -> Integer
absolute x
  | x < 0     = -x
  | otherwise = x
```

Each guard (beginning with `|`) is a Boolean expression. The first guard that evaluates to `True` determines which expression is returned. The `otherwise` guard is a catch-all that always evaluates to `True`.

Guards can also be combined with pattern matching. Consider a function that classifies numbers:

```
classify :: Integer -> String
classify 0 = "zero"
classify n
  | n < 0     = "negative"
  | n > 0     = "positive"
```

This function first checks if the number is zero. If not, it uses guards to determine whether the number is negative or positive.

2.4.5 Case Expressions

A **case expression** allows pattern matching against a value within an expression, similar to `switch` statements in languages like `C` or *Java*. However, *Haskell*'s `case` expressions integrate seamlessly with **pattern matching**. Syntactically, a `case` expression follows this general form:

```
case expression of
    pattern1 -> result1
    pattern2 -> result2
    pattern3 -> result3
    ...
```

To evaluate this case expression, *Haskell* proceeds as follows:

- The **expression** is evaluated.
- The first **pattern** that matches is chosen.
- The corresponding **result** is returned.
- If no pattern matches, a **runtime error** occurs.

Below is an implementation of a function that computes the first element of a list:

```
myHead :: [a] -> a
myHead xs = case xs of
    []     -> error "No head for empty lists!"
    (x:_) -> x
```

Since `case` expressions **evaluate to values**, they can be used inline just like any other expression. Furthermore, `case` expressions can be combined with guards:

```
describeNumber :: Int -> String
describeNumber x = case x of
    _ | x < 0     -> "Negative number"
      | x > 0     -> "Positive number"
      | otherwise -> "Zero"
```

2.4.6 The Use of `where` and `let` Clauses

Complex function definitions often benefit from local variable bindings to make the code clearer and more modular. *Haskell* provides two constructs for this purpose: `where` clauses and `let` expressions.

(a) A `where` clause allows the definition of auxiliary functions and variables at the end of a function definition. For example, a function so solve the quadratic equation

$$a \cdot x^2 + b \cdot x + c = 0$$

can be defined as follows:

```
quadratic ::  Double -> Double -> Double -> (Double, Double)
quadratic a b c = (x1, x2)
where
    discriminant = b * b - 4 * a * c
    x1 = (-b + sqrt discriminant) / (2 * a)
    x2 = (-b - sqrt discriminant) / (2 * a)
```

The `where` clause contains definitions that are local to the function `quadratic`, making the main expression easier to read.

(b) A `let` expression provides a way to bind variables in an expression.

```
compute :: Int -> Int
compute x = let y = x * 2
                z = y + 3
            in z * z
```

Here, `y` and `z` are only visible in the expression following the `in` keyword.

A `let` expression can be used on the right hand side of a guarded equation and is then local to this equation, whereas the variable defined in a `where` clause are defined for all equations defining a function.

2.4.7 Currying and Partial Application

A unique feature of *Haskell* is that functions are curried by default. This means that every function taking multiple arguments is actually a series of functions, each taking a single argument. Consider the addition function:

```
add :: Int -> Int -> Int
add x y = x + y
```

This function can be partially applied:

```
increment :: Int -> Int
increment = add 1
```

Here, `increment` is a new function that adds 1 to its argument. Currying promotes code reuse and leads to elegant function composition.

2.4.8 Lambda Expressions

Lambda expressions, or anonymous functions, allow for the definition of functions without explicitly naming them. They are useful for short-lived functions, particularly when passing a function as an argument to higher-order functions. For instance:

```
squares :: [Int] -> [Int]
squares xs = map (\x -> x * x) xs
```

The lambda expression `(\x -> x * x)` takes an argument `x` and returns its square. Lambda expressions are concise and facilitate inline function definitions.

2.4.9 Higher-Order Functions

Functions that take other functions as arguments or return them as results are called higher-order functions. They are central to functional programming. For example, the `map` function applies a function to every element in a list:

```
myMap :: (a -> b) -> [a] -> [b]
myMap f []     = []
myMap f (x:xs) = f x : myMap f xs
```

A custom higher-order function might filter elements in a list based on a predicate:

```
myFilter :: (a -> Bool) -> [a] -> [a]
myFilter \_ [] = []
myFilter p (x:xs)
  | p x       = x : myFilter p xs
  | otherwise = myFilter p xs
```

In this definition, `myFilter` takes a predicate `p` and a list, returning a list of elements for which `p` returns `True`.
Note that *Haskell* comes with the functions `map` and `filter` that are defined exactly as we have defined the functions `myMap` and `myFilter`. We had to rename these function when defining them ourselves because in contrast to *Python*, *Haskell* does not allow the redefinition of predefined functions.

2.4.10 Recursive Function Definitions

Since there are no control structures like `for` loops or `while` loops in *Haskell*, we have to use recursion far more often than in *Python*. In *Haskell*, many functions, particularly those that operate on recursive data structures such as lists, are defined recursively. Consider the definition of the fibonacci function that computes the n^{th} Fibonacci number:

```
fibonacci :: Integer -> Integer
fibonacci 0 = 0
fibonacci 1 = 1
fibonacci n = fibonacci (n - 1) + fibonacci (n - 2)
```

While this implementation is straightforward, it may not be efficient for large `n`. More advanced techniques, such as memoization or tail recursion, can optimize recursive functions.

Tail recursion is a form of recursion where the recursive call is the last operation in the function. Tail-recursive functions can be optimized by the compiler to iterative loops, saving stack space. For example, a tail-recursive factorial function can be written as:

```haskell
factorialTR :: Integer -> Integer
factorialTR n = factHelper n 1
  where
    factHelper 0 acc = acc
    factHelper k acc = factHelper (k - 1) (k * acc)
```

In this version, the accumulator `acc` carries the intermediate results, ensuring that the recursive call to `factHelper` is in tail position.

2.4.11 Polymorphism and Overloaded Functions

Haskell functions are often polymorphic, meaning that they can operate on values of various types. The function `id`, which returns its argument unchanged, is a classic example:

```haskell
id :: a -> a
id x = x
```

Here, `id` is defined for any type `a`. Polymorphism is facilitated by *Haskell*'s type system and its use of type classes, which allow functions to operate on a range of types that share common behavior.

Another example is the `const` function:

```haskell
const :: a -> b -> a
const x _ = x
```

`const` takes two arguments and returns the first, ignoring the second. Its polymorphic type signature reflects the fact that it can be applied to arguments of any types.

2.4.12 The Function Composition Operator (.)

The function composition operator, written as a single dot "`.`", is one of Haskell's most elegant and powerful tools. It allows you to combine two functions into a new function without having to mention the argument explicitly. In mathematical notation, function composition is written as

$$(f \circ g)(x) = f\big(g(x)\big),$$

and in Haskell the `.` operator is defined as follows:

```haskell
(.) :: (b -> c) -> (a -> b) -> a -> c
f . g = \x -> f (g x)
```

The type signature `(b -> c) -> (a -> b) -> a -> c` of the function composition operator "`.`" tells us the following:

- The operator takes two functions as arguments. The first function has type `b -> c` and the second has type `a -> b`.
- The composed function takes an input of type `a` and returns a result of type `c`.

In other words, if you have two functions `f :: b -> c` and `g :: a -> b`, then their composition `(f . g)` is a function of type `a -> c` which, given an argument `x`, computes `g x` first and then applies `f` to that result, yielding `f (g x)`.

Example: Consider the following functions:

```haskell
increment :: Int -> Int
increment x = x + 1

double :: Int -> Int
double x = x * 2
```

Using the composition operator, we can define a new function that doubles a number and then increments it:

```haskell
doubleThenIncrement :: Int -> Int
doubleThenIncrement = increment . double
```

Evaluating `doubleThenIncrement 3` proceeds as follows:

`doubleThenIncrement 3 = increment (double 3) = increment 6 = 7`.

Example: Another common use of function composition is to reduce parentheses in nested function calls. For instance, consider sorting a list and then reversing it:

```haskell
import Data.List (sort)

reverseSorted :: Ord a => [a] -> [a]
reverseSorted = reverse . sort
```

Here, `sort` is applied to the list first, and then `reverse` is applied to the sorted list. Without composition, you would have to write:

`reverseSorted xs = reverse (sort xs)`.

The function composition operator has the following properties:

- **Associativity:** Function composition is right-associative. This means that the expression

  `f . g . h` is parsed as `f . (g . h)`,

  so you can write a chain of composed functions without excessive parentheses.

- By using the composition operator, we can define functions without mentioning their arguments explicitly. For example, instead of writing:

  `f x = negate (abs x)` we can write `f = negate . abs`.

  This style is called point-free function definition and helps to make code more concise and expressive.

The `.` operator has a very high precedence (`infixr 9`), which means it binds more tightly than most other operators. For example:

`negate . abs $ (-3)` is equivalent to `negate (abs (-3))`

Without the proper use of parentheses, the second expression would be parsed incorrectly.

Summary: The `.` operator allows you to build complex functions by composing simpler ones. Its type signature encapsulates the idea that the output of one function becomes the input of another, and its high precedence and associativity properties facilitate concise, point-free definitions. Whether you are chaining arithmetic transformations or building data processing pipelines, mastering function composition is key to writing elegant Haskell code. Using composition, a pipeline of functions can be created without resorting to nested function calls. For example:

```
process :: [Int] -> Int
process = sum . map square . filter even
```

Here, the list is first filtered for even numbers, then each even number is squared, and finally the squares are summed. The composition operator makes the data flow clear and concise.

2.4.13 Point Free Style

In point-free programming, functions are defined without explicitly mentioning their arguments. Consider the function:

```
sumSquares :: [Int] -> Int
sumSquares xs = (sum . (map (\x -> x * x))) xs
```

This can be rewritten in point-free style as:

```
sumSquares :: [Int] -> Int
sumSquares = sum . map (\x -> x * x)
```

Point-free style can lead to more concise definitions, though it is important to balance conciseness with clarity. My own experience is that it takes a while to get used to point-free style, but once you get the hang of it, you will use it often.

2.4.14 List Comprehensions in *Haskell*

List comprehensions provide a concise and expressive way to construct lists by specifying their elements in terms of existing lists. Inspired by mathematical set notation, list comprehensions allow you to generate new lists by transforming and filtering elements from one or more source lists. Their elegant syntax and expressive power make them a favorite tool for many *Haskell* programmers.

At its core, a list comprehension has the following general syntax:

$$[\text{expression} \mid \text{qualifier}_1, \text{qualifier}_2, \ldots, \text{qualifier}_n]$$

In this construct, the *expression* is evaluated for every combination of values generated by the qualifiers. Qualifiers can be either generators or filters. A generator has the form

```
pattern <- list
```

and is used to extract elements from an existing list, while a filter is simply a Boolean expression that restricts which elements are included. We begin with a simple example where we generate a list of squares for the numbers from 1 to 10:

```
squares :: [Int]
squares = [ x * x | x <- [1..10] ]
```

Here, `x <- [1..10]` is a generator that iterates over the numbers 1 through 10, and the expression `x * x` calculates the square of each number.

List comprehensions also allow you to filter elements by adding a Boolean condition. For instance, to generate a list of even squares from 1 to 10 we can write the following:

```
evenSquares :: [Int]
evenSquares = [ x * x | x <- [1..10], even x ]
```

In this example, the qualifier `even x` acts as a filter, ensuring that only even values of `x` are considered. As a result, only the squares of even numbers are produced.

List comprehensions can also combine multiple generators to produce lists based on the Cartesian product of several lists. For example, the following comprehension generates pairs of numbers where the first element is taken from `[1,2,3]` and the second from `[4,5]`:

```
pairs :: [(Int, Int)]
pairs = [ (x, y) | x <- [1,2,3], y <- [4,5] ]
```

This comprehension evaluates the tuple `(x, y)` for each combination of `x` and `y`, resulting in a list of pairs.

2.4.15 Haskell Sections

Haskell sections are a syntactic convenience that allow for the partial application of binary operators. In essence, a section is an expression in which one operand of a binary operator is fixed, yielding a new function that awaits the missing operand. For example, the section `(+1)` represents a function that adds 1 to its argument and is therefore equivalent to the lambda expression $\lambda x \to x + 1$. Likewise, `(1+)` is equivalent to $\lambda x \to 1 + x$. This mechanism enables concise function definitions without the need for explicit lambda notation.

The general form of a section is either:

`(o e)` or `(e o)`

where `o` is a binary operator and `e` is an expression that serves as one of its argument. The missing argument is supplied when the section is applied to an argument. Therefore,

1. `(o e)` is equivalent to `\x -> x o e`, and
2. `(e o)` is equivalent to `\x -> e o x`.

For example,

`(+ 1)` is equivalent to `\x -> x + 1`.

Hence we can define a function that increments its argument as follows:

```
increment :: Num a => a -> a
increment = (+1)
```

Sections work for any infix operator, and their behavior is influenced by the operator's properties. Consider the subtraction operator:

```
subtractFromTen :: Num a => a -> a
subtractFromTen = (10-)
```

Here, `(10-)` is a section that subtracts its argument from 10. Notice that the order matters: while `(10-)` computes $10 - x$, the expression `(-10)` is simply a negative literal representing -10. Let us take a look at a few more examples:

- `(==0)` is a section that checks if its argument is equal to 0:

```haskell
isZero :: (Eq a, Num a) => a -> Bool
isZero = (==0)
```

- `(/2)` divides its argument by 2:

```haskell
half :: Fractional a => a -> a
half = (/2)
```

- `(2^)` computes 2 raised to the power of its argument:

```haskell
powerOfTwo :: (Integral b, Num a) => b -> a
powerOfTwo = (2^)
```

2.5 Algebraic Data Types in Haskell

In Haskell, an **Algebraic Data Type (Adt)** is a way to define new types by combining other types. ADTs allow developers to construct complex data structures in a type-safe and declarative manner. The term "algebraic" comes from the fact that these data types are constructed using algebraic operations such as sums (alternatives) and products (combinations). ADTs are primarily classified into two types:

- **Sum types** (disjoint unions), which represent a choice between multiple alternatives.
- **Product types**, which combine multiple values together.

2.5.1 Defining Algebraic Data Types

ADTs are defined in Haskell using the `data` keyword. The general syntax is:

```haskell
  data TypeName = Constructor1 Type1 Type2 ...
                | Constructor2 Type3 Type4 ...
                ...
                | Constructor7 Type8 Type9 ...
```

Each constructor represents a possible form that a value of this type can take.

2.5.2 Example: Defining a Simple Adt

Consider defining a type to represent a traffic light:

```haskell
data TrafficLight = Red | Yellow | Green
```

This defines `TrafficLight` as a type with three possible values: `Red`, `Yellow`, and `Green`. Each value is represented by a constructor with no associated data.

2.5.3 Sum Types (Disjoint Unions)

A sum type is a type that can take on one of several different values. The `|` symbol is used to define different constructors. For example:

```haskell
data Shape = Circle Float | Rectangle Float Float
```

Here, a `Shape` can either be a `Circle` (with a radius of type `Float`) or a `Rectangle` (with width and height of type `Float`).

2.5.4 Product Types

Product types combine multiple values together in a single constructor. A classic example is a coordinate point:

```
data Point = Point Float Float
```

This defines a `Point` type where each `Point` consists of two `Float` values representing the x and y coordinates.

2.5.5 Pattern Matching with ADTs

Pattern matching allows functions to be defined by destructuring ADT values. Here is an example using the `Shape` type:

```
area :: Shape -> Float
area (Circle r) = pi * r * r
area (Rectangle w h) = w * h
```

The function `area` takes a `Shape` and returns its area by matching on its constructor.

2.5.6 Recursive Data Types

ADTs can also be recursive, allowing the definition of complex data structures such as lists or trees. Consider the definition of a binary tree:

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
```

This defines a tree where each node contains either a single value (`Leaf`) or two subtrees (`Node`).

2.6 Type Classes in Haskell

A **type class** in Haskell is a mechanism for defining a set of functions that can operate on multiple types. It enables *ad-hoc polymorphism*, meaning that a single function can have different implementations depending on the type of its arguments. Type classes are similar to interfaces in object-oriented languages but are more flexible because they are based on Haskell's strong type system.

A type class is defined using the `class` keyword. Here is a simple example:

```
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
```

This declaration defines an `Eq` type class with two methods: `(==)` for equality and `(/=)` for inequality. Any type that is an instance of `Eq` must provide implementations for these functions.

2.6.1 The `Eq` Type Class

The `Eq` type class is used for types that support equality comparison. It is defined in the Haskell Prelude as:

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

The last two lines provide default implementations: if `(==)` is defined, then `(/=)` can be derived automatically and vice versa.

All standard Haskell types, such as `Int`, `Char`, and `Bool`, are instances of `Eq`. User-defined types can also be made instances of `Eq`:

```haskell
data Color = Red | Green | Blue

instance Eq Color where
    Red   == Red   = True
    Green == Green = True
    Blue  == Blue  = True
    _     == _     = False
```

Alternatively, Haskell allows automatic instantiation of the type class `Eq` for algebraic data types via the keyword `deriving`:

```haskell
data Color = Red | Green | Blue deriving Eq
```

2.6.2 The `Ord` Type Class

The `Ord` type class is used for totally ordered data types. It extends `Eq` and defines methods for comparison:

```haskell
class Eq a => Ord a where
    compare :: a -> a -> Ordering
    (<), (<=), (>=), (>) :: a -> a -> Bool
    max, min :: a -> a -> a
```

Here, the function `compare` returns an `Ordering` value, which can be `LT` (less than), `GT` (greater than), or `EQ` (equal). The other comparison functions can then be defined in terms of `compare`:

```haskell
compare x y
    | x == y    = EQ
    | x <= y    = LT
    | otherwise = GT
```

Like `Eq`, `Ord` instances can be derived automatically for algebraic data types:

```haskell
data Color = Red | Green | Blue deriving (Eq, Ord)
```

For derived `Ord` instances, the order of constructors in the data declaration determines the comparison result. For example:

```haskell
Red   < Green == True
Green > Blue  == False
```

2.6.3 Custom Instances of `Ord`

If a type does not have a natural ordering, we can define our own:

```haskell
data Size = Small | Medium | Large

instance Eq Size where
    Small  == Small  = True
    Medium == Medium = True
    Large  == Large  = True
    _      == _      = False

instance Ord Size where
    Small  <= _      = True
    Medium <= Large  = True
    Medium <= Medium = True
    Large  <= Large  = True
    _      <= _      = False
```

With this instance, Haskell can compare `Size` values, allowing operations like sorting.

Chapter 3

Example Programs

In this Chapter we will present a few small example programs that that illustrate the concepts discussed so far.

3.1 Perfect Numbers

The first example shows the computation of the perfect numbers. A natural number n is a perfect number if it is the sum of all its proper divisors, where a natural number t is a proper divisor of n if t divides n evenly, i.e. iff

$$\texttt{mod}\, n\, t = 0.$$

Figure 3.1 on page 28 show a program to compute the list of all perfect numbers. The function `isPerfect` takes a single natural number and checks whether it is perfect, while the function `perfect` computes the list of all perfect numbers.

```haskell
isPerfect :: Integer -> Bool
isPerfect n = sum [t | t <- [1.. n-1], mod n t == 0] == n

perfect :: [Integer]
perfect = [n | n <- [1..], isPerfect n]
```

Figure 3.1: Computing the perfect numbers.

3.2 Computing Word Frequencies

The following example is inspired by an example from the book *Thinking Functionally with Haskell* by Richard Bird [Bir14]. In stylometry one of the tasks is to compute the relative frequencies of different words occurring in a text. Stylometry can be used for authorship attribution, compare for example the book *Authorship Attribution* by Patrick Juola [Juo08]. Figure 3.2 on page 32 shows a program that reads a file, in our case this file contains the text

of the book *Moby Dick* by Herman Melville [Mel51] and then computes the frequencies of the hundred most common words.
Before we can discuss the program from Figure 3.2 we need to discuss some library functions that we will use.

(a) The `break` function in Haskell is a higher-order function from the `Prelude` module that is used to split a list into two parts based on a predicate. The type signature of `break` is:

```
break ::  (a -> Bool) -> [a] -> ([a], [a])
```

Therefore a call of this function has the form

```
break p xs
```

- The first argument `p` has type `(a -> Bool)`. It is a **predicate** that determines where the list should be split.
- The second argument `xs` is a list of elements of some generic type `[a]` that is to be split in two parts.
- The function satisfies the following specification:

```
break p xs = (first, second)
    where
        first = [ x | x <- xs, not (p x) ]
        first + second == xs
```

The return value is a pair. The `first` component of this pair is the list of all elements `x` of the list `xs` that do not satisfy the predicate `p`. The `second` component contains the remaining elements of `xs`.

For example, we have

```
break (> 3) [1, 2, 3, 4, 5] = ([1, 2, 3], [4, 5])
```

The function `break` scans the list from left to right. It collects elements into the first list until the predicate function returns `True`. The second list starts from the first element where the predicate `p` is satisfied. The function `break` can be implemented as follows:

```
break _ [] = ([], [])  -- Base case: Empty list returns two empty lists
break p (x:xs)
    | p x       = ([], x:xs)
    | otherwise = (x:ys, zs)
    where (ys, zs) = break p xs
```

(b) The function `sort` is part of the `Data.List` module in Haskell. It is used to sort a list in ascending order based on the default ordering of its elements. It has the following type signature:

```
sort :: (Ord a) => [a] -> [a]
```

Given a list `xs` the expression `sort xs` returns the list `xs` sorted in ascending order.

The function requires that the elements of the list belong to the `Ord` type class (i.e., they must support comparison operations). The implementation in `Data.List` is using the merge sort algorithm.

(c) The function `words` is part of the `Data.List` module in Haskell. It is used to split a string into a list of words, where words are defined as contiguous sequences of non-whitespace characters. It has the following type signature:

```
words ::  String -> [String]
```

Given a string `s`, the expression `words s` breaks `s` into a list of words, using whitespace as the delimiter. Consecutive whitespace characters are treated as a single separator, meaning that empty strings between spaces are ignored. Leading and trailing whitespace is also ignored. The following examples show the behaviour of the function `words`:

- `words "Hello, world!" = ["Hello,","world!"]`
- `words "Martin Müller-Lüdenscheid" = ["Martin","Müller-Lüdenscheid"]`

Note that punctuation marks like "`,`" or "`!`" are not separated from other letters.

A simple way to implement `words` recursively is as follows:

```haskell
import Data.Char (isSpace)
words :: String -> [String]
words [] = []   -- Base case: an empty string results in an empty list
words s  = let s' = dropWhile isSpace s   -- drop leading whitespace
           in case s' of
               [] -> []
               _  -> let (word, rest) = break isSpace s'
                     in word : words rest
```

(d) The function `isAlpha` has type

```
isAlpha :: Char -> Bool
```

and checks, whether the given character is an alphabetic Unicode character, i.e. for a character `c` the call

```
isAlpha c
```

returns `True` if `c` is either one of the lower case letters `'a'`, $\cdots$, `'z'`, or of the upper case letters `'A'`, $\cdots$, `'Z'`, or a unicode letter from another script, like, e.g. the Greek letter α.

(e) The function `toLower` has the following type signature:

```
toLower ::  Char -> Char
```

For a letter `c`, the expression

```
toLower c
```

returns the lower case version of the letter `c`. For example,

`toLower 'A' = 'a'` and `toLower 'b' = 'b'`.

(f) The function `concatMap` has the type

```
concatMap :: (a -> [b]) -> [a] -> [b]
```

Thus, `concatMap` takes two arguments:

- A function of type `a -> [b]`, which maps each element of type `a` to a list of elements of type `b`.

- A list of type `a`, which serves as the input to be processed.

The result is a single list of type `b` obtained by applying the function to each element of the input list and then concatenating the resulting lists.

The function `concatMap` can be understood as the composition of `map` and `concat`:

- `map` applies the given function to each element of the input list, producing a list of lists: `[a] -> [[b]]`.
- `concat` then flattens this list of lists into a single list: `[[b]] -> [b]`.

Formally, we can express this as:

```
concatMap f xs = concat (map f xs)
```

where:

- `f` is the function of type `a -> [b]`,
- `xs` is the input list of type `[a]`.

The following example shows `concatMap` at work:

```
concatMap (\x -> [x, x+1]) [1, 3, 5]
```

- The function `\x -> [x, x+1]` maps each element `x` to the list `[x, x+1]`.
- Applying `map` yields the list of lists `[[1, 2], [3, 4], [5, 6]]`.
- Applying `concat` flattens this to `[1, 2, 3, 4, 5, 6]`.

Thus, the result is the list

```
[1, 2, 3, 4, 5, 6].
```

We proceed to discuss the program that is shown in Figure 3.2 on page 32.

1. In line 1, we import the functions `sort` and `words` from the module `Data.List`.
2. Similarly, we import `isAlpha` and `toLower` in line 2.
3. The function `countRuns` takes a sorted list of elements of type `a` and returns a list of pairs. Each pair consists of a unique element from the input list and the count of its consecutive occurrences (runs). For example, we have:

   ```
   countRuns ['a','a','a','a','b','c','c'] = [(4,'a'),(1,'b'),(2,'c')]
   ```

   The implementation is recursive. If the given list is non-empty and therefore has the form `x:xs`, we first check how often the element `x` occurs at the beginning of the list `xs` by breaking `xs` into two sublists:

   - `run` is the longest prefix of the list `xs` that only contains the element `x`.
   - `rest` is the remainder of the list `xs` after removing all occurrences of the element `x` from the beginning of the list `xs`.

   Then the element `x` occurs `1 + length run` times in the list `x:xs` because it occurs `length run` times in `xs` and it also is the first element of the list `x:xs`. Furthermore, we have to call `countRuns` recursively on the `rest` list.

```haskell
import Data.List (sort, words)
import Data.Char (isAlpha, toLower)

type Text = [Char]

countRuns :: Eq a => [a] -> [(Int, a)]
countRuns [] = []
countRuns (x:xs) =
    let (run, rest) = break (/= x) xs
    in (1 + length run, x) : countRuns rest

frequency :: Int -> Int -> Double
frequency c n = fromIntegral c / fromIntegral n

divide :: Int -> [(Int, a)] -> [(Double, a)]
divide n ps = [(frequency c n, x) | (c, x) <- ps]

myWords :: String -> [String]
myWords = map (filter isAlpha) . words

showPair :: (Double, String) -> String
showPair (f, w) = w ++ ": " ++ show f ++ "\n"

comnWrds :: Int -> Text -> String
comnWrds n b = concatMap showPair $
               divide nw . take n . reverse . sort . countRuns . sort $
               aw
  where
    aw = (myWords . map toLower) b
    nw = length aw

readBook :: FilePath -> IO String
readBook filename = readFile filename

main :: IO ()
main = do
    contents <- readBook "moby-dick.txt"
    putStrLn $ comnWrds 100 contents
```

Figure 3.2: Finding the most common words, `part I`.

4. The function `frequency` takes two natural numbers as its input.
    - `c` is the number of occurrences of a given word, while

- `n` is the total number of words.

The function computes the fraction `c/n`. It makes use of the function

```
fromIntegral :: Int -> Double
```

that transforms a natural number into a floating point number. This is necessary, because division is only defined for floating point numbers.

5. The function `divide` is called with two arguments:

   ```
   divide n ps
   ```

   Here, `n` is the total number of words, while `ps` is a list of pairs of the form `(c, w)` where `w` is a word and `c` is the number of occurrences of this word in a given text. It converts the number of occurrences into frequencies by dividing the count `c` by the total number of words.

6. When we use the function `words`, the resulting words will contain punctuation symbols. For example, we have

   ```
   words "Hello, world!" == ["Hello," ,"world!"]
   ```

   This is not what we want. The function `myWords` takes a string, extracts the words and finally removes all symbols from these words that are not alphabetical characters.

7. The function `showPair` is only used to format the output. It takes a pair of the form (f, w) where w is a word and f is the frequency of this words. It returns a string of the form "w: f\n", where `\n` is a newline symbol.

8. The function `comnWrds` is the star of the show and does the main work of extraction the word and computing their frequencies. The function receives two arguments

   (a) `n` is the number of words that should be returned.

   (b) `b` is a string representing a book.

   The task of the function is to return the `n` most common word from the book `b`.

   The implementation works by chaining a number of different functions together in a pipeline.

   (a) The variable `aw` (short for all words) is a list containing all words. Note that we first transform all characters in the string `b` to lower case before we extract the list of words. This way, the strings “The” and “the” represent the same word.

   (b) The variable `nw` (short for number of words) stores the number of all different words that have been found.

   (c) Next, the words are sorted using the function `sort`. This way, the same words are grouped together and hence it is easy to count how often each word occurs.

   (d) The function `countRuns` counts the frequency of each words. It transforms the sorted list of words into a list of pairs of the form

   ```
   [(1, "abasement"), (1, "abandonedly"), ...  (981, "of"), (1073,
   "the"), ...]
   ```

In these pairs, the first component is the number of occurrences of the corresponding word. This list is the sorted ascendingly according to the number of occurrences.

(e) Since we want the most frequent words at the beginning, this list is reversed.

(f) Then we `take` the first `n` pairs from this list.

(g) The function `divide` turns a pair of the form

$$(c, w)$$

where w is a word and c is the number of occurrences of this word into a pair of the form

$$(f, w)$$

where f is the frequency of the word. This is done by dividing c by the total number of all words `nw`.

(h) Finally, the function `concatMap` turns the list of pairs into a string with the help of the function `showPair`.

3.3 The Wolf, the Goat, and the Cabbage

Next, we show how to solve logical puzzles with *Haskell*. The puzzle we want to solve is known as the wolf, goat, and cabbage problem:

> *An farmer has to sell a wolf, a goat, and a cabbage on a market place. In order to reach the market place, she has to cross a river. The boat that she can use is so small that it can only accommodate either the goat, the wolf, or the cabbage in addition to the farmer. Now if the farmer leaves the wolf alone with the goat, the wolf will eat the goat. If, instead, the farmer leaves the goat with the cabbage, the goat will eat the cabbage. Is it possible for the farmer to develop a schedule that allows her to cross the river without either the goat or the cabbage being eaten?*

We will solve this problem by formulating it as a search problem. This search problem is then solved via breadth first search.

Definition 1 (Search Problem) A search problem is a 4-tuple of the form

$$\mathcal{P} = \langle Q,\ R,\ \texttt{start},\ \texttt{goal}\rangle$$

where

(a) Q is the set of states, also known as the state space.

(b) R is a binary relation on Q, i.e. we have $R \subseteq Q \times Q$.

If $\langle x, y\rangle \in Q$, then there is a direct connection between the states x and y, that is we can move in one step from the state x to the state y.

(c) `start` is the start state, hence $\texttt{start} \in Q$.

(d) `goal` is the goal state, hence $\texttt{goal} \in Q$.

A path is a list $[s_1, \cdots, s_n]$ such that $\langle s_i, s_{i+1}\rangle \in R$ for all $i \in \{1, \cdots, n-1\}$. The length of this path is defined as the length of this list minus 1, i.e. the path $[s_1, \cdots, s_n]$ has length $n-1$. The reason for defining the length of this path as $n-1$ and not n is that the path consists of $n-1$ edges of the form $\langle s_i, s_{i+1}\rangle$ where $i \in \{1, \cdots, n-1\}$. A path $[s_1, \cdots, s_n]$ is a solution to the search problem $\mathcal{P}$ iff the following conditions are satisfied:

1. $s_1 = \texttt{start}$, i.e. the first element of the path is the start state.
2. $s_n = \texttt{goal}$, i.e. the last element of the path is the goal state. ◇

We will model the wolf, goat, and cabbage problem as a search problem. To this end we define the set

$$\texttt{all} := \{\texttt{"farmer"}, \texttt{"wolf"}, \texttt{"goat"}, \texttt{"cabbage"}\}.$$

Every state will be represented as a subset `s` of the set `all`. The idea is that the set `s` specifies those objects that are on the left side of the river. We assume that initially the farmer is on the left side of the river. Therefore, the set of all possible states can be defined as the set

$$Q := \bigl\{s \mid s \in 2^{\texttt{all}} \wedge \neg\texttt{problem}(s) \wedge \neg\texttt{problem}(\texttt{all}\backslash s)\bigr\}$$

Here, we have used the procedure `problem` to check whether a given state `s` has a problem, meaning that either the wolf can eat the goat or the goat can eat the cabbage. Note that since `s` is the set of objects on the left side, the expression $\texttt{all}\backslash s$ computes the set of objects on the right side of the river.

We implement a function `problem` that takes a set `s` of items as input and checks whether there is a problem. A state `s` of objects has a problem if both of the following conditions are satisfied:

1. The farmer is not an element of the set `s` and
2. either `s` contains both the goat and the cabbage or `s` contains both the wolf and the goat.

Therefore, we can implement the function `problem` in *Haskell* as follows:

```haskell
problem :: Set String -> Bool
problem s = "farmer" ∉ s &&
            (("goat" ∈ s && "cabbage" ∈ s) ||  -- goat eats cabbage
             ("wolf" ∈ s && "goat"    ∈ s))    -- wolf eats goat
```

We proceed to compute the relation R that contains all possible transitions between different states. We will compute R using the formula:

$$R := R_1 \cup R_2$$

Here R_1 describes the transitions that result from the farmer crossing the river from left to right, while R_2 describes the transitions that result from the farmer crossing the river from right to left. Mathematically, we can define the relation R_1 as follows:

$$R_1 := \bigl\{\langle s, s\backslash b\rangle \mid s \in Q \wedge b \in 2^s \wedge s\backslash b \in Q \wedge \texttt{"farmer"} \in b \wedge \texttt{card}(b) \leq 2\bigr\}$$

Let us explain this definition in detail:

1. Initially, s is the set of objects on the left side of the river. Hence, s is an element of the set of all states that we have defined as Q.

2. b is the set of objects that are put into the boat and that do cross the river. Of course, for an object to go into the boat is has to be on the left side of the river to begin with. Therefore, b is a subset of s and hence an element of the power set of s.

3. Then $s\backslash b$ is the set of objects that are left on the left side of the river after the boat has crossedthe river. Of course, the new state $s\backslash b$ has to be a state that does not have a problem. Therefore, we check that $s\backslash b$ is an element of the set of states Q.

4. Furthermore, the farmer has to be in the boat. This explains the condition

   $\texttt{"farmer"} \in b.$

5. Finally, the boat can only have two passengers. Therefore, we have added the condition

   `card(b) <= 2.`

   Here, the function `card` computes the *cardinality* of the set b, i.e. it computes the number of elements of b.

Next, we have to define the relation R_2. However, as crossing the river from right to left is just the reverse of crossing the river from left to right, R_2 is just the inverse of R_1. Hence we define:

$$R_2 := \big\{ \langle y, x\rangle \mid \langle x, y\rangle \in R_1 \big\}$$

Finally, the start state has all objects on the left side. Therefore, we have

$$\texttt{start} := \texttt{all}$$

In the end, all objects have to be on the right side of the river. That means that nothing is left on the left side. Therefore, we define

$$\texttt{goal} := \{\}$$

Figure 3.3 on page 37 shows a *Haskell* program that formulates the wolf, goat, and cabbage problem as a search problem. We discuss this program line by line:

1. In the first line we activate the Unicode syntax language extension. This language extension enables us to use Unicode characters as operator symbols.

   Normally, every text that is enclosed in the strings "`{-`" and "`-}`" is considered a comment. However, there is one exception: If a multiline comment starts as "`{-#` " and ends with "`#-}`, then the comment is considered a pragma. A pragma can be used to activate a language extension.

2. In line 2 we import a number of items from the module `Data.Set`.

   (a) `Set` is the type of sets.

   (b) The operator `(\\)` is used to build the difference of two sets, i.e. if a and b are sets, then

   $$a \setminus\!\setminus b := \{\, x \in a \mid x \notin b \,\}.$$

```haskell
{-# LANGUAGE UnicodeSyntax #-}
import Data.Set (Set, (\\), fromList, toList, filter)
import Bfs (search)
import SetUtils (power, (∈), (∉))

problem :: Set String → Bool
problem s = "farmer" ∉ s &&
            (("goat" ∈ s && "cabbage" ∈ s) ||  -- goat eats cabbage
             ("wolf" ∈ s && "goat"    ∈ s))    -- wolf eats goat

allItems :: Set String
allItems = fromList ["farmer", "wolf", "goat", "cabbage"]

noProblem :: Set String → Bool
noProblem s = not (problem s) && not (problem $ allItems \\ s)

states :: Set (Set String)
states = Data.Set.filter noProblem (power allItems)

r1 :: [(Set String, Set String)]
r1 = [ (s, s \\ b) | s ← toList states,
                     b ← toList $ power s, s \\ b ∈ states,
                     "farmer" ∈ b, length b <= 2
     ]

r2 :: [(Set String, Set String)]
r2 = [ (s1, s2) | (s2, s1) ← r1 ]

r :: [(Set String, Set String)]
r = r1 ++ r2

start :: Set String
start = allItems

goal :: Set String
goal = fromList []

main :: IO ()
main = mapM_ (putStrLn . show) $ head $ search r start goal
```

Figure 3.3: The wolf, the goat, and the cabbage.

(c) The function `fromList` has type signature `fromList :: Ord a => [a] -> Set a.`

The expression `fromList l` takes a list `l` of elements that can be compared and returns a set containing the elements of `l`. We have to require that the type a belongs to the type class `Ord` because in Haskell sets are implemented as ordered binary search trees and hence the elements of a set need to be comparable.

(d) The function `toList` has type signature

`toList :: Set a -> [a].`

The expression `toList s` takes a set `s` of elements and returns a list containing the elements of `s`.

(e) The function `Data.Set.filter` has type signature

`Data.Set.filter :: (a -> Bool) -> Set a -> Set a.`

The expression `Data.Set.filter p s` takes a predicate `p` and a set `s`. It returns a new set containing those elements `x` of `s` such that `p x` is `True`:

`Data.Set.filter p s =` $\{$ `x` $\in$ `s | p x` $\}$

Note that we have to qualify the function name `filter` as `Data.Set.filter` in order to distinguish it from the predefined function `filter` that has the type signature

`Data.Set.filter :: (a -> Bool) -> [a] -> [a].`

3. Line 3 imports the module `Bfs` which implements breadth-first search. We import the function `search` from this module, which has the following type signature:

```
search :: Ord a => [(a, a)] -> a -> a -> [Bfs.Path a]
```

The function `search` is called as follows:

```
search r s g
```

Here `r` is a binary relation on some set of states `q`, i.e. we have `r` $\in$ `q`$\times$`q`. The arguments `s` and `g` are elements of `q` and the function returns a list of all shortest paths leading from the state `s` (start) to the state `g` (goal).

The module `Bfs` is shown in Figure 3.4 on page 40.

4. In line 4 we import the module `SetUtils`. This module provides a number of functions and operators that are useful when working with sets.

(a) The function `power` has the type signature

`power ::  Ord a => Set a -> Set (Set a)`

Given a set `s` the expression `power s` computes the set of all subsets of `s`, i.e. we have

`power s =` $\{$ `a | a` $\subseteq$ `s` $\}$.

For exampe, we have:

`power` $\{1, 2\}$ `=` $\{\{\}, \{1\}, \{1,2\}, \{2\}\}$

(b) The binary operator "$\in$" has the type signature

```
(∈) ::  Ord a => a -> Set a -> Bool
```

The expression $x \in s$ yields `True` if and only if x is an element from the set s.

(c) The binary operator "$\notin$" is the negation of the operator "$\in$".

The module `SetUtils` is shown in Figure 3.5 on page 41.

5. Line 6–9 define the function `problem`. This function takes a state, i.e. a set of the items on a shore and returns `True` if there would be a problem on that shore.

6. Line 12 defines the set `allItems` as

   {`"farmer"`, `"wolf"`, `"goat"`, `"cabbage"`}.

7. The function `noProblem` takes a set `s` of items as its argument. This set represents the items on the left shore. It checks that there is neither a problem on the left shore nor on the right shore.

   Note that if `s` is the set of items on the left shore, then `allItems \\ s` is the set of items on the right shore.

8. `states` is the set of all states of the search problem. These are those subsets of the set `allItems` such that there is no problem on either shore.

9. `r1` is the relation that describes all those transitions from the left shore to the right shore that do not lead to a problematic state. Rather than representing this relation as a set of pairs it is represented as a list of pairs because this makes the definition a little bit more convenient.

   (a) `s` is a state.

   (b) `b` is the set of items that cross the river on the boat.

   - First of all, `b` has to be a subset of `s` because only item on the left shore cross the river from the left to the right.
   - The new state is `s \\ b` must not be problematic. Hence it has to be a memeber of `states`.
   - The `"farmer"` has to be on the boat and there may be at most two items on the boat.

   (c) After the crossing of the river the new state is the set of those items that are left on the left shore `s \\ b`.

10. `r2` is the inverse of the relation `r1`.

11. `r` is the union of the relations `r1` and `r2`.

12. The `start` state is the state where all items are on the left shore.

13. The `goal` state is the state where all items are on the right shore. Hence nothing is left on the left shore.

14. The expression `search r start goal` computes the list of all minimal paths from `start` to `goal` and the function `head` extracts the first such path.

    This path is a list. The elements of this list are then converted to strings via `show` and printed on separate lines via `putStrLn`.

15. The function `mapM_` is a variant of the function `map` that works with functions performing `IO`.

```haskell
{-# LANGUAGE UnicodeSyntax #-}
module Bfs (search) where

import Data.Set (Set)
import qualified Data.Set as Set

(∪) :: Ord a ⇒ Set a → Set a → Set a
(∪) = Set.union

(∈) :: (Ord a) ⇒ a → Set a → Bool
(∈ ) = Set.member

(∉) :: (Ord a) ⇒ a → Set a → Bool
x ∉ s = not (x ∈ s)

type Path a = [a]

search :: forall a. Ord a ⇒ [(a, a)] → a → a → [Path a]
search relation start goal = go [[start]] (Set.singleton start)
  where
    go :: [Path a] → Set a → [Path a]
    go [] _ = []
    go paths visited
      | null goalPaths = go newPaths newVisited
      | otherwise = goalPaths
      where
        newPaths = [ path ++ [z] | path ← paths, (y, z) ← relation,
                                   last path == y, z ∉ visited       ]
        newVisited = visited ∪ Set.fromList [last path | path ← newPaths]
        goalPaths = filter (\p → last p == goal) newPaths
```

Figure 3.4: Breadth First Search.

Figure 3.6 on page 42 displays the relation R graphically.

Figure 3.7 shows a shedule for solving the puzzle.

```haskell
{-# LANGUAGE UnicodeSyntax #-}
module SetUtils (power, (∪), (∩), (∈), (∉), (∅)) where

import Data.Set (Set, (\\), fromList, toList, findMin, insert)
import qualified Data.Set as Set

-- the empty set
(∅) :: Ord a ⇒ Set a
(∅) = fromList []

-- union of sets
(∪) :: Ord a ⇒ Set a → Set a → Set a
(∪) = Set.union

-- intersection of sets
(∩) :: Ord a ⇒ Set a → Set a → Set a
(∩) = Set.intersection

-- is x an element of s
(∈) :: (Ord a) ⇒ a → Set a → Bool
x ∈ s = x `Set.member` s

-- is x not an element of s
(∉) :: (Ord a) ⇒ a → Set a → Bool
x ∉ s = not (x ∈ s)

-- compute the power set of the set s
power :: Ord a ⇒ Set a → Set (Set a)
power s
  | Set.null s = fromList [ (∅) ]
  | otherwise  = p ∪ Set.map (\x → insert m x) p
    where m = findMin s
          p = power (s \\ fromList [m])
```

Figure 3.5: The set utilities.

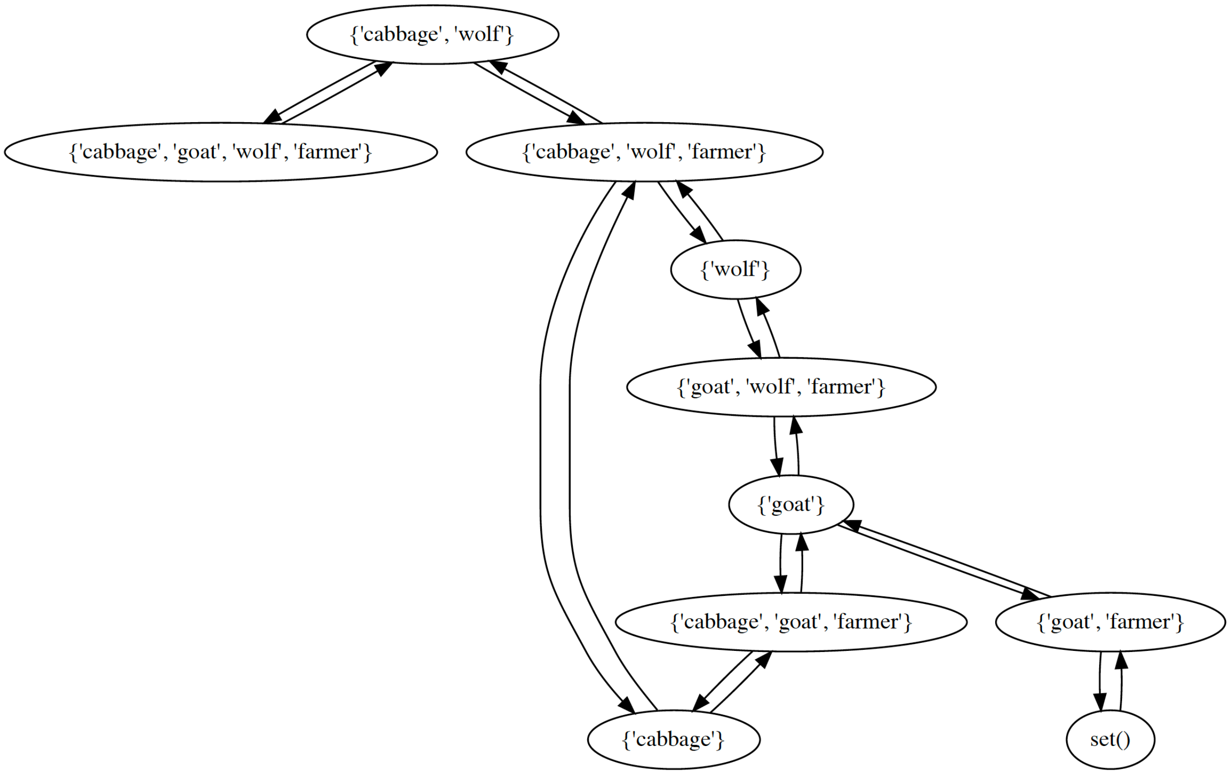

Figure 3.6: The relation R shown as a directed graph.

```
{'cabbage', 'farmer', 'goat', 'wolf'}                                      {}
                          >>>> {'farmer', 'goat'} >>>>
{'cabbage', 'wolf'}                                         {'farmer', 'goat'}
                          <<<< {'farmer'} <<<<
{'cabbage', 'farmer', 'wolf'}                                        {'goat'}
                          >>>> {'farmer', 'wolf'} >>>>
{'cabbage'}                                         {'farmer', 'goat', 'wolf'}
                          <<<< {'farmer', 'goat'} <<<<
{'cabbage', 'farmer', 'goat'}                                        {'wolf'}
                          >>>> {'cabbage', 'farmer'} >>>>
{'goat'}                                         {'cabbage', 'farmer', 'wolf'}
                          <<<< {'farmer'} <<<<
{'farmer', 'goat'}                                         {'cabbage', 'wolf'}
                          >>>> {'farmer', 'goat'} >>>>
{}                                     {'cabbage', 'farmer', 'goat', 'wolf'}
```

Figure 3.7: A schedule for the farmer.

3.4 Symbolic Differentiation

Figure 3.8 onm page 44 shows a package that implements a crude form of symbolic differentiation. For example, when given the expression x^x this program is able to show that

$$\frac{\mathrm{d}}{\mathrm{d}\,x}x^x = x^x \cdot \big(\ln(x) + 1\big).$$

We discuss this program line by line.

1. We define `Expr` as a recursive algebraic data type. This data type is meant to represent arithmetic expressions.
   (a) Given two arithmetic expressions e_1 and e_2, an expression of the form `Add` e_1 e_2 is interpreted as the sum $e_1 + e_2$.
   (b) Similarly, `Sub` e_1 e_2 is interpreted as $e_1 - e_2$.
   (c) `Mul` e_1 e_2 is interpreted as $e_1 \cdot e_2$.
   (d) `Div` e_1 e_2 is interpreted as e_1/e_2.
   (e) `Pow` e_1 e_2 is interpreted as $e_1^{e_2}$.
   (f) Given an arithmetic expression e, `Exp` e is interpreted as $\exp(x)$, where exp is the exponential function.
   (g) Similarly, `Ln` e is interpreted as $\ln(x)$, where ln is the natural logarithm.
   (h) Given a character c, `Var` c is interpreted as the variable with name c.
   (i) Given a real number r, `Num` r is interpreted as the number r.
2. In order to be able to convert an arithmetic expression into a string, we turn `Expr` into an instance of the type class `show` in line 12 and define the function `show` via matching for all the different cases of the given arithmetic expression.

   In order for the implementation to be concise, the result of calling `show` s has more parentheses than necessary.
3. Given an arithmetic expression e and the name of a variable c, the function call

   `diff` e c

   computes the derivative of e with respect to c. For example, line 29 implements the product rule.

```haskell
{-# LANGUAGE UnicodeSyntax #-}
data Expr = Add Expr Expr
          | Sub Expr Expr
          | Mul Expr Expr
          | Div Expr Expr
          | Pow Expr Expr
          | Exp Expr
          | Ln  Expr
          | Var Char
          | Num Double

instance Show Expr where
  show :: Expr → String
  show (Add x y) = "(" ++ (show x) ++ "+" ++ (show y) ++ ")"
  show (Sub x y) = "(" ++ (show x) ++ "-" ++ (show y) ++ ")"
  show (Mul x y) = "(" ++ (show x) ++ "*" ++ (show y) ++ ")"
  show (Div x y) = "(" ++ (show x) ++ "/" ++ (show y) ++ ")"
  show (Pow x y) = "(" ++ (show x) ++ "^" ++ (show y) ++ ")"
  show (Exp x)   = "exp(" ++ show(x) ++ ")"
  show (Ln  x)   = "ln("  ++ show(x) ++ ")"
  show (Var c)   = [c]
  show (Num x)   = show x

diff :: Expr → Char → Expr
diff (Add f g) x = Add fs gs
  where fs = diff f x; gs = diff g x
diff (Sub f g) x = Sub fs gs
  where fs = diff f x; gs = diff g x
diff (Mul f g) x = Add (Mul fs g) (Mul f gs)
  where fs = diff f x; gs = diff g x
diff (Div f g) x = Div (Sub (Mul fs g) (Mul f gs)) (Mul g g)
  where fs = diff f x; gs = diff g x
diff (Pow f g) x = diff (Exp (Mul g (Ln f))) x
diff (Exp f) x = Mul (Exp f) fs
  where fs = diff f x
diff (Ln f) x = (Div fs f)
  where fs = diff f x
diff (Var v) x   = if v == x then Num 1 else Num 0
diff (Num _) _   = Num 0
```

Figure 3.8: Symbolic differentiation.

Bibliography

[Bir14] Richard Bird. *Thinking Functionally with Haskell*. Cambridge University Press, 2014.

[Hut16] Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2nd edition, 2016.

[Juo08] Patrick Juola. *Authorship Attribution*. Now Publishers, 2008.

[Lip11] Miran Lipovača. *Learn You a Haskell for Great Good!* No Starch Press, San Francisco, CA, 2011.

[Lot22] Steven F. Lott. *Functional Python Programming - Third Edition*. Packt Publishing, 3rd edition, 2022.

[Mel51] Herman Melville. *Moby-Dick; or, The Whale*. Harper & Brothers, New York, 1851.

[Mer15] David Mertz. *Functional Programming in Python*. Packt Publishing, 2015.

[Rei23] James L. Reid. *Python Functional Programming: A Hands-on Guide To Write Clean & Powerful Applications*. Independently Published, 2023.

[Ski23] Rebecca Skinner. *Effective Haskell: Solving Real-World Problems with Strongly Typed Functional Programming*. Pragmatic Bookshelf, 2023.

[Tho11] Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley, 3rd edition, 2011.