Functional Programming in *Haskell*

— Winter 2025 —

Prof. Dr. Karl Stroetmann

February 9, 2025

Danger **Work in Progress!** Danger

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

   by Rebecca Skinner [Ski23].

2. *Programming in Haskell*, second edition

   by Graham Hutton [Hut16].

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

   by Miran Lipovača [Lip11].

   This book is available free online at https://learnyouahaskell.com/.

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 following contrived example:

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

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` 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:

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

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

```python
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 1.1: A *Python program to compute the prime numbers up to n.*

Figure 1.1 on page 4 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 $a \cdot \mathtt{p}$ with $a \geq \mathtt{p}$ have to be removed from the list, since a product of the form $a \cdot \mathtt{p}$ with $a < \mathtt{p}$ has already been removed when removing multiples of a or, if a is not prime, of multiples of whatever prime is contained in a.

```
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 1.2: Computing the prime numbers.

Figure 1.2 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 all primes.

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, and
2. expressions,
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 large 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) Bool represents boolean values. It has two possible values: True and False.

Example Usage:
```haskell
-- A boolean literal
isHaskellFun :: Bool
isHaskellFun = True
```

(g) 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:
```haskell
-- 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:
```haskell
-- 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:

```haskell
[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:

```haskell
-- 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:

```haskell
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 list 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 lists, 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:

```haskell
("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. When declaring the type of a tuple, the size, i.e. the number of elements, is 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:

```haskell
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 `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	
&&	3	right-associative	Boolean **and**.
||	2	right-associative	Boolean **or**
$	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 : [])).
```

This natural right grouping 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:

```haskell
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:

```haskell
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:

```haskell
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:

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

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 as add taking an integer `x` and returning a new function that takes an integer `y`. 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:

```haskell
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 The Use of where and let Clauses

Complex function definitions often benefit from local 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:

```haskell
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.

```haskell
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.

2.4.6 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:

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

This function can be partially applied:

```haskell
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.

2.4.7 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:

```haskell
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.

2.4.8 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:

```haskell
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:

```haskell
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.

2.4.9 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:

```haskell
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.

2.4.10 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.

2.4.11 Function Composition and Pipelines

Function composition is a powerful tool in Haskell, allowing complex functions to be built by composing simpler ones. The composition operator is defined as:

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

You should take note of the syntax of the type declaration: When declaring the type of an operator (in this case the operator is the symbol `.`) we have to enclose the operator in parentheses and write `(.)`.

Using composition, a pipeline of functions can be created without resorting to nested function calls. For example:

```haskell
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.

2.4.12 Point Free Style

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

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

This can be rewritten in point-free style as:

```haskell
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.

2.4.13 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:

```haskell
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:

```haskell
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]`:

```haskell
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.

Bibliography

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

[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.

[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.