Haskell Mini Reference 2023

A Quick Guide to the Haskell Functional Programming Language for Busy Coders

Harry Yoon

Version 1.0.3, 2023-05-14

Copyright

Haskell Mini Reference:

A Quick Guide to the Haskell 2010 Programming Language

Published: February 2023

Harry Yoon
San Diego, California

ISBN: 1111

Preface

If we measure "market shares" of all programming languages in some way and plot the data as a pie chart, the functional programming languages, all of them combined, would not even show up as a wedge-shaped slice on the pie chart.

Despite their general importance, and practical usefulness, functional programming is still considered a niche in the software industry. There can be many reasons for this, but one of the main reasons is lack of good educational materials. There are also a lot of misinformation out there regarding functional programming. Many software developers consider functional programming "difficult", which can be done only by the "elitist" programmers. That cannot be further from the truth.

Functional programming is *different* from imperative programming. But, not necessarily more difficult. *Unfamiliarity breeds prejudice*.

Haskell is one of the most widely used functional programming languages. Haskell has been around for over 30 years, and it has influenced the language designs of numerous (modern) programming languages, including many popular imperative languages such as Python, JavaScript, C#, Julia, and Rust to name a few.

Haskell is a pure functional programming language. This means that we primarily, and almost exclusively, use the mathematical principle of function applications and function compositions as the primary means of computation. This also means that more traditional imperative programming styles using side effects cannot be generally used while programming in Haskell (with a few important exceptions).

When programmers with the imperative programming background start learning functional programming languages like Haskell, they generally face two main challenges. First, they will need to learn pure functional programming, which requires a rather different mindset.

This can be the hard part for some people who have been trained in imperative programming for many years. Second, languages like Haskell use somewhat different syntax from most of the main stream languages. In fact, functional programming languages all tend to use more terse syntax, for example, and this trips over many beginning Haskell programmers. However, this is the *easy* part.

Books like this can help you learn Haskell language syntax so that you can focus more on learning high-level functional programming styles. As a matter of fact, this *mini reference* will not teach you how best to program in a functional style, but rather it will only teach you the essentials of the Haskell programming language. If you are looking for a tutorial on functional programming, this book may not be the right one for you. If you are a complete beginner, then you will not find this book very useful.

This book is specifically written for the people

- Who have *some* exposure to functional programming and would like to learn Haskell,
- Who are *learning* functional programming in Haskell and making a rather slow progress due to its somewhat unfamiliar syntax, or
- Who are *experienced* in procedural programming and want to get a quick taste of the Haskell language.

For this intended target audience, this mini reference will provide an excellent overview of the Haskell functional programming language.

> This book is largely based on the official "Haskell 2010 Language Report", but it is not an authoritative language reference. We recommend the readers refer to the original *Report* for more precise and more detailed information whenever there is any ambiguity in the descriptions in the book.

Dear Readers:

Please read b4 you purchase, or start investing your time on, this book.

A programming language is like a set of standard lego blocks. There are small ones and there are big ones. Some blocks are straight and some are L-shaped. You use these lego blocks to build spaceships or submarines or amusement parks. Likewise, you build programs by assembling these building blocks of a given programming language.

This book is a *language reference*, written in an informal style. It goes through each of these lego blocks, if you will. This book, however, does not teach you how to build a space shuttle or a sail boat. If this distinction is not clear to you, it's unlikely that you will benefit much from this book. This kind of language reference books that go through the syntax and semantics of the programming language broadly, but not necessarily in gory details, can be rather useful to programmers with a wide range of background and across different skill levels.

This book is not for complete beginners, however. When you start learning a foreign language, for instance, you do not start from the grammar. Likewise, this book will not be very useful to people who have little experience in real programming. On the other hand, if you have some experience programming in other languages, and if you want to quickly learn the essential elements of this particular language, then this book can suit your needs rather well.

Ultimately, only you can decide whether this book will be useful for you. But, as stated, this book is written for a wide audience, from beginner to intermediate. Even experienced programmers can benefit, e.g., by quickly going through books like this once in a while. We all tend to forget things, and a quick regular refresher is always a good idea. You will learn, or re-learn, something "new" every time.

Good luck!

Table of Contents

Chapter 1. Introduction

The Haskell programming language is based on lambda calculus at its core. In fact, all syntactic structures in Haskell are formally defined through translations of those structures into the lambda calculus-based core part, known as the Haskell kernel.

However, you do not need to be familiar with lambda calculus to use Haskell. Haskell is a *high-level* general purpose programming language that supports, and encourages, pure functional style programming. If you are new to functional programming, then Haskell is *the* best language to learn functional programming with.

Despite some common misconceptions, functional programming styles are widely used in modern programming. For example, many developers are now used to programming styles using higher order functions like map, filter, and reduce. Pattern matching has been adopted by virtually all modern languages. Immutability is considered a holy grail even in imperative programming nowadays, especially in the multi-core concurrent programming environments.

It may still require some time and practice to transition to pure functional programming, but as indicated in the Preface, we do not believe that is the main reason why functional programming languages like Haskell are not as much widely used.

It is most likely the unfamiliar syntax that is what keeps many programmers from trying out functional programming languages. Therefore, we hope that books like this one that focuses on the language grammar can help developers get into functional programming more easily and more willingly.

Other than that, the case for (pure) functional programming is overwhelming, and we will not make any effort to *convert* you in this book.

1.1. Example Haskell Program

Merge sort is one of the *most functional* algorithms. Here's a simple implementation of merge sort in Haskell.

MergeSort.hs

```
module MergeSort(sort) where                ①

divide :: Ord a => [a] -> ([a], [a])        ②
divide xs = splitAt ((length xs + 1) `div` 2) xs

merge :: Ord a => [a] -> [a] -> [a]         ③
merge [] s2 = s2                            ④
merge s1 [] = s1
merge s1@(x:xs) s2@(y:ys)                   ⑤
  | x > y = y : merge s1 ys
  | otherwise = x : merge xs s2

sort :: Ord a => [a] -> [a]                 ⑥
sort [] = []
sort [x] = [x]
sort list =
  let (fstHf, sndHf) = divide list          ⑦
   in merge (sort fstHf) (sort sndHf)
```

① This line declares a module `MergeSort` and exports a function `sort`. Module imports and exports are explained in the Modules chapter.

② This line denotes a type signature for the function `divide`, whose implementation follows in the next line. Notice the general syntax, *name :: type*, separated by double colons (`::`). `splitAt` is a "built-in" function, included in the Haskell Standard Prelude. Types and functions are two of the most important concepts in Haskell programming, and they are explained throughout this book.

③ Likewise, a type declaration of the function, `merge`.

④ The `merge`, and `sort`, functions are implemented using pattern matching, which is described in detail in the later part of the book. Pattern matching was first introduced by Haskell, and it is now becoming a core part of virtually every modern programming language, thanks to its intuitive syntax and expressive power.

⑤ Recursion is at the heart of functional programming. One of the unique features of Haskell is that, lexically, Haskell programs can be written in layout-sensitive or layout-insensitive formats. For instance, the expression written in three lines in this example can be written in one line as well. The layout rule is described in the Lexical Structure chapter, in the very beginning of the book.

⑥ The `sort` function also uses pattern matching and recursion. Notice the common pattern in the way that functions are defined over multiple patterns (and, over multiple lines) in Haskell.

⑦ Unlike some popular beliefs, even pure functional programming uses "variables" (albeit *immutable*). The `let in` expression is explained in the main part of the book, in particular, in the Let and Where chapter. Note that this `let in` expression captures the essence of the merge sort algorithm.

Here's a sample program using this `sort` function:

Main.hs

```
module Main where                        ①

import MergeSort (sort)                  ②

main :: IO ()                            ③
main = do                                ④
  print $ sort [7, 5, 8, 6, 4, 9]        ⑤
```

① Every Haskell program needs a `Main` module, which includes a value named `main`. This is similar to the way C-style languages work, in which the "main" function is the entry point to a program.

② Importing the `sort` function from the `MergeSort` module. Notice the lack of semicolons throughout this code example. Again, this is explained in the context of layout rules.

③ The type of `main` is `IO`, which is an instance of the `Monad` class. Types and classes (or, typeclasses) are explained throughout this reference. The *infamous* `Monad` class is briefly described in the Monads chapter, primarily for completeness. Note that, only through monads, we can include (non-pure) actions in pure functional programs.

④ In the monadic context, the `do` expression can be used for "sequential programming". Do expressions often include multiple statements, e.g., expressions and declarations. The expression in this particular line will output *[4,5,6,7,8,9]* to standard output, or the terminal.

If you do not fully understand this program at this point, then read on. This book will teach you how to read Haskell programs, at least in terms of all essential syntax.

1.2. Functional Programming

Pure functional programming is about computing (desired) values through applications of functions. (In this book, and in functional programming in general, a function means a pure function, that is, a mathematical function.) You get an input, a value, and you produce an output, another value, through pure computations. There are no imperative statements involved like "do this and do that".

Although there is no general consensus as to what exactly is functional programming (FP), FP is often characterized by a few tenets, if you will:

- In FP, functions are the main building blocks of programs.
- FP only deals with values, and values are by definition immutable.
- FP does not cause side effects (that is, unless explicitly intended).

In addition, Haskell has a few important characteristics that are not necessarily considered an intrinsic part of FP. For example,

- Haskell has a strong static type system, with support for parametric polymorphism.
- Haskell supports universal type inference, and hence type declarations are (almost) optional.
- Haskell supports lazy evaluation of expressions by default, which can lead to code optimization.
- Haskell supports user-defined operators, which is much more powerful than the "predefined operator overloading" mechanism found in other programming languages.
- Haskell supports powerful pattern matching, which plays an essential role in virtually every aspect of Haskell programming.
- In Haskell, all functions take one value and return one value, through what is known as currying.
- In Haskell, every function is a value. And, every value is a function.
- Haskell isolates pure functions and non-pure actions using Monads (which originated from category theory).
- As a high-level programming language, Haskell runtimes support automatic memory management, e.g., garbage collection.
- Haskell programs can be either dynamically interpreted, or they can be compiled to executables.

Haskell has such a strong static type system that the Haskell compiler removes all type information when building an executable. That is, there is no need for runtime type information for Haskell programs after they have been verified by the static type checker. Furthermore, despite the pervasive misconceptions that FP languages are "slow", the leading Haskell compiler can produce highly optimized code which are comparable to those generated by other "fast" imperative languages.

1.3. Book Organization

We start the book with a quick introduction to the Haskell software development process, in particular, using the Cabal - GHC toolchain. This is included primarily for completeness, especially for absolute beginners, and it can be skipped if you have some experience with Haskell programming.

In fact, this book assumes that the reader has some exposure to Haskell, or other similar functional programming languages.

In the next chapter, we briefly go through the lexical structure of Haskell programs, again for completeness. This book, by its very nature, emphasizes breadth more than depth. This chapter can also be skipped, maybe except for the layout rules section, unless you are completely new to Haskell. The rest of the book is organized more or less in a top down fashion.

A Haskell program comprises one or more modules. Modules are generally used to manage namespaces and organize large programs. Names can be shared among different modules through Haskell's import-export mechanism. All Haskell programs include a special module `Main`, which includes a value named `main` with the type `IO`. This is the entry point to any Haskell program.

A Haskell module consists of a collection of declarations for entities like ordinary values, datatypes, and type classes, and for fixity information. Some declarations can only be used at the module-, or top-, level, and they are described in the top-level declarations chapter. Some other kinds of declarations, on the other hand, can be included both at the top-level and at some nested context. They are described in the nested declarations chapter.

We also go through some basics of Haskell's type system in these two chapters, including `data` types, `newtype` types, and `type` synonyms.

As with other programming languages, Haskell includes a number of primitive, or "builtin", types. We go through some of them, such as booleans, numbers, characters, and strings in this chapter, and we further discuss user-defined data types and type classes later in the book.

Haskell is a pure functional programming language, and hence it does not have constructs comparable to the "statements" in other imperative programming languages, whose main purpose is to generate side effects. At the level below the modules and declarations are expressions, as described in the next several chapters. An expression denotes a value and has a static type. Expressions are the bread and butter of Haskell functional programming.

It may seem somewhat ironic, because many developers consider functional programming languages like Haskell "complex", but the Haskell's language grammar is much simpler than those of other widely used programming languages. In fact, the Haskell language itself includes only a few different kinds of expressions (again, no side effect causing statements), and the rest of the language constructs (e.g., operators) are included in the standard library. Some of them are part of "the Standard Prelude", and they are no different from the "built-in" language syntax for all intents and purposes.

As is the case with virtually all functional programming languages, functions are the most important construct in Haskell. In the Functions chapter, we review how to define a function, how to invoke a function, and how to compose two or more functions in Haskell. We also introduce lambda functions in this chapter.

Other than the primitive types like `Bool` and `Char`, and numbers, lists are the most important types in Haskell, as in many functional programming languages. Functional programming often involves manipulating lists. Tuples are also important compound data types that deserve a careful study if you are new to Haskell. Tuples provide a light-weight syntax for user-defined data types, which are discussed later in the book.

All expressions in Haskell have static types. Haskell can deduce the broadest possible type for *any* expression, which is called its "principal type". Otherwise, that is, if Haskell cannot deduce the principal type, then it is not a valid expression, i.e., not a valid Haskell code. The expression type signature syntax can be used to specify a type narrower than the principal type. Or, it can sometimes be used to make an otherwise-invalid expression valid by explicitly specifying the type.

As with any high-level programing language, Haskell supports conditional expressions, with the familiar `if` - `then` - `else` syntax. Unlike in many other languages, however, both `then` and `else` clauses are required in Haskell.

Functional programming languages also use "variables". But they have different meanings, and they play different roles, in functional programming languages like Haskell. In particular, variables in Haskell do not imply "storage locations" in memory as in imperative programming languages. (Pure) functional programming languages only deal with "values". Variables are just names for values. In the next chapter, Let and Where, we go through the basic syntax of the `let` expressions. We also discuss the `where` syntax in this chapter, which itself is not an expression but can be used in a somewhat similar fashion to `let`, e.g., to define variables.

If we have to pick one particular feature that is the most important in Haskell, it would be the pattern matching. In Haskell, it is almost the foundation of all other expressions. Virtually everything is built on top of pattern matching. The `case` expressions play the fundamental role in this regards. Other pattern matching syntax is ultimately translated to `case` expressions. A `case` expression can include one or more alternative patterns, and each pattern can include zero or more Boolean guards.

In the following chapter, Patterns, we go through each of the pattern types supported by Haskell. This is a somewhat artificial classification, and in practice, we mostly use some combinations of these patterns.

In Haskell, there is little distinction between functions and operators. Operators are just a special kind of functions (e.g., which take two arguments). In the chapter, Core Functions, we describe a few of the "built-in" functions and operators from the Prelude.

In the next chapter, List Functions, we go through other "built-in" functions that are used to manipulate lists. There are quite a few, and they are *all* important, to varying degrees. We only briefly cover each of these functions, but it is essential to understand and "internalize" all these functions in order to be able to use Haskell effectively. One thing to note is that Haskell comes with other standard libraries beyond the Prelude, but we do not cover those in this book.

Haskell supports a rather powerful polymorphic type system. After having gone through all important expressions, we now go back to a few important kinds of declarations, namely, the `data` type and `class` declarations.

Needless to say, types are important in modern programing. This is especially so in languages like Haskell which provide strong type-safety checks at build time. It is pretty much impossible to have type-related errors at run time. It does not mean that *if you can build it, it runs without errors*, but it is pretty close. Haskell makes it rather easy to create and use custom types through the `data` type declaration syntax. A data type is defined by declaring one or more constructors, with positional fields. Haskell also supports the record syntax for data constructors, e.g., using labeled fields. The record syntax is now widely adopted by many other programming languages.

Haskell's polymorphic type system is based on type classes. We briefly discuss the `class` declarations, `instance` declarations, and the `deriving` syntax in the following chapter. The Standard Prelude includes a few predefined classes, such as `Eq`, `Ord`, `Enum`, `Bounded`, `Read`, `Show`, and other numeric classes like `Num`. We briefly go through some of these classes in the next chapter, Standard Classes.

Whether justifiable or not, Functors and (especially) Monads are generally considered the most difficult topics in Haskell. This (short) book will not be able to convince you otherwise if you are in that camp. But, nonetheless, we briefly cover each of these builtin classes. Learning is about recognizing patterns. If you have some experience in programming, then you will realize that Functors and Monads are just simple abstractions over some familiar programming patterns. If not, no worries. You do not have to understand precisely what these terms mean to be able to program in Haskell.

In the monadic context, one can use sort of "imperative-style" programming, which a majority of programmers are more used to, even in Haskell. This is briefly explained in the next chapter, `do` Expressions.

The most important beneficiary of the `Monad` class is the I/O related actions. In fact, Haskell, as a pure functional programming language, did not initially have support for I/O for many years. Now, through `Monad`, Input/Output can be easily incorporated into Haskell programs. The `IO` type is one of the most important instances of `Monad`.

In the next, and final, chapter, IO Functions we go through some of the I/O related functions defined in the Prelude. These are core functions to be able to do basic IO in any Haskell programs.

It should be noted that, as indicated earlier, we do not cover any of the Haskell Standard libraries in this book, in the interest of space and the reader’s time. This book is a *mini language reference*.

Chapter 2. Haskell Software Development

The Haskell programming language was originally created over 35 years ago. But there have been only two official releases in terms of the language specifications. The Haskell language definition was first publicly released in 1998, which is known as *Haskell 98*. The second and currently most up-to-date spec was released in 2010, which is officially called the *Haskell 2010 Language Report*.

At this point, there does not appear to be an ownership of the language by any particular organizations. That does not mean Haskell is dead or abandoned. Some day, there might be formed another Haskell Committee, and they will produce the next version of the language, if necessary. Meanwhile, the GHC team (originally, of the University of Glasgow) has the de-facto stewardship of Haskell. They create and distribute the most widely-used Haskell compiler and interpreter, called *ghc* and *ghci*, respectively. And, their build tools support an extensive set of "language extensions", which are essentially additions to the language beyond the Haskell 2010 Report.

Although this book's main focus is the Haskell language itself, we will briefly discuss in this chapter the particular toolings provided by the GHC team, to the benefit of the people who are new to Haskell software development.

2.1. Development Tools

The most important tool in programming is clearly the compiler (or, the interpreter). But, the modern software development is aided by various tools. Haskell is no exception. We briefly go through some of the GHC-related development tools in this section, without attempting to be complete or exhaustive.

2.1.1. *GHCup*

GHCup is an optional tool that allows easy management of other Haskell build and package management tools. You can download it from the GHCup Installation page [https://www.haskell.org/ghcup/install/]. Although it is not required, it is often the best and easiest way to manage Haskell tools such as *GHC*, *Cabal*, *Stack*, and *HLS*.

For example, you can easily manage these tools using the *tui* command:

```
$ ghcup tui
```

(If you have used *RustUp* for Rust development, for instance, these two tools are comparable to each other. In fact, there are many similar tools across different programming languages.)

2.1.2. *Cabal*

Cabal is one of the most essential tools for professional Haskell software development. It is a project and package management tool, and it is also a high-level build tool (which uses the *ghc* compiler underneath). You can scaffold a simple Haskell project using the *init* command. For instance,

```
$ cabal init -i
```

You can build a Cabal project using *cabal build*, or you can build and run using *cabal run* during development. For example,

```
$ cabal run --verbose=0                    ①
```

① The *verbose* flag can be used to change the verbosity of the build output messages.

You can also install any Haskell packages (available on Hackage) using *cabal install*. *cabal --help* will print out some common usages of the *cabal* command.

2.1.3. *Stack*

Stack is a (newer) alternative to *Cabal*. That is, you can manage and build a Haskell project using *Stack* instead of *Cabal*. Some people prefer one tool over the other, but it is really a matter of preference.

It should be noted that *Stack* is also integrated into the Haskell Cabal infrastructure. The relationship between *Stack* and *Cabal* is comparable to that of *Gradle* vs *Maven* in Java, for instance.

2.1.4. *HLS*

HLS, or "Haskell Language Server", is used to add Haskell language support to IDEs or other programs that understand the language server protocols. VS Code, along with the third-party provided extensions, provides good dev support for a wide range of programming languages (e.g., syntax highlighting, intellisense, static code analysis during development, etc.). If you install *HLS*, then you can use VS Code, for example, for Haskell development,

2.1.5. *GHC*

GHC stands for "Glasgow Haskell Compiler". As stated, it is the de-facto standard compiler for Haskell. If you develop production-quality software in Haskell, you will most likely have to use *GHC*, either directly or indirectly.

In practice, the *ghc* command is rarely used directly. Most developers use the aforementioned high-level (project-oriented) build tools like *Cabal* or *Stack*.

2.1.6. *GHCI*

If you are new to Haskell programming, or to functional programming in general, REPL is one of the most important tools during software development. It is rather hard to theorize precisely why REPL plays a lot more important roles in functional programming than in imperative programming, but it is not uncommon to see Haskell programmers always keep the REPL terminal open during development.

The *ghci* command, the REPL that comes with the GHC toolchain, does not compile the Haskell program like *ghc*. Rather it interprets the given expressions, one at a time, in the interactive mode. (The *runghc* command also interprets a given Haskell program, but in the non-interactive mode.) You can start a Haskell REPL by simply invoking the command, *ghci*:

```
$ ghci
GHCi, version 9.4.4: https://www.haskell.org/ghc/  :? for help
ghci>                                             ①
```

① The default GHCI prompt, waiting for the next command.

You can see a list of all available commands using the *:h* command. For example, *:info*, or *:i*, displays information about the provided names, and *:type*, or *:t*, shows the type of a given expression.

```
ghci> :i map
map :: (a -> b) -> [a] -> [b]   -- Defined in ‘GHC.Base’
ghci> :t "Hello World"
"Hello World" :: String
ghci> :t 42
42 :: Num a => a                                  ①
```

① Numeric literals are polymorphic in Haskell. We explain what this notation means later in the book.

2.1.7. Haskell source code

As with most programming languages, Haskell programs are generally written in files as text. Haskell programs can be coded in two different forms, a normal program style and a "literate" style. The Haskell source code file written in the regular style is generally saved in a file with the *.hs* extension. This represents a normal code, as is commonly done in any other programming languages.

On the other hand, in the literate programming style, Haskell code should be prefixed with >. (Or, alternatively, code blocks can be enclosed within LaTex style tags.) All other text is considered a comment in the literate style code. Literate source code is generally saved in the files with the extension *.lhs*.

2.2. Language Extensions

As mentioned, the GHC toolchain provides an extensive set of language extensions. You can selectively turn on or off each of these extensions, e.g., using the *ghc* command line options or using the compiler `LANGUAGE` pragmas (which we do not discuss in this book). Note that, in Glasgow Haskell, the baseline for the language definition is Haskell 98, and not Haskell 2010. That is, you will need to enable all necessary language extensions (or, "features") if you plan to use Haskell 2010.

Luckily, GHC also provides a small number of meta-extension options which include other options. For example, there are currently three predefined values, e.g., as of GHC 9.4, *Haskell98* (e.g., no extensions enabled), *Haskell2010*, and *GHC2021*.

We will always be using *Haskell2010* with *ghc* in this book unless otherwise specifically noted. When there is any uncertainty or conflict, the Haskell 2010 Language Report should be the authoritative reference. As for what language extensions are available and how to use them, we recommend the readers refer to the GHC User's Guide.

Chapter 3. Lexical Structure

Haskell uses the Unicode character set. A Haskell program can only include graphic characters and whitespaces. A comment is lexically considered a whitespace.

3.1. Comments

3.1.1. Line comments

An ordinary line comment begins with a sequence of two consecutive dashes (e.g. `--`) and extends to the end of the line, including the newline. (Note that, in Haskell, the double dashes can also be part of lexically legal operator symbols, e.g., `-->`.)

```
-- This is comment.
--- This is also comment.
```

3.1.2. Nested comments

A nested multiline comment begins with `{-` and ends with `-}`. Nested comments may be nested to any depth. Any occurrence of the character sequence `{-` within the nested comment starts a new nested comment, terminated by `-}`. Within a nested comment, each `{-` is matched by a corresponding occurrence of `-}`.

```
{--
{-
I am a comment inside another comment.
-}
--}
```

3.2. Identifiers

An identifier consists of a letter followed by zero or more letters (including underscores `_`), digits, and single quotes (`'`). One or more single quotes are often used at the end of an identifier to denote alternative versions of the given entity with the same identifier but without the single quote suffix. Identifiers are case sensitive.

Haskell identifiers are lexically distinguished into two namespaces:

- Variable identifiers - The identifiers that begin with a lowercase letter, which denote variables or functions, and
- Constructor identifiers - The identifiers that begin with an uppercase letter, which denote types or constructors.

Underscore `_` is treated as a lowercase letter, and it can occur wherever a lowercase letter is syntactically allowed. The identifier, `_`, by itself is a reserved identifier, which is used as the wildcard in patterns. Haskell generally offers warnings for declared but unused identifiers. However, these warnings are suppressed against the identifiers that start with underscores, by convention. This, for example, allows programmers to use names like `_foo` or `_bar` as a placeholder (that they expect to be unused).

3.3. Reserved Words

The following 20 identifiers are reserved in Haskell:

```
case        class       data        deriving
do          else        if          import
in          infix       infixl      infixr
instance    let         of          module
newtype     then        type        where
```

3.4. Operators

Operator symbols consist of one or more symbol characters, and they are classified into two distinct namespaces.

- An operator symbol with two or more characters starting with a colon `:` is a constructor.
- An operator symbol starting with any other character is an ordinary identifier.

All operators are infix by default, and they can be used in a section.

3.4.1. Reserved operator symbols

```
..      :       ::      =       \       |
<-      ->      @       ~       =>
```

3.5. Layout Rules

Haskell uses curly braces and semicolons for the purposes of grouping, etc., just like many other programming languages. Haskell, however, also supports layout-based style of coding without requiring braces and semicolons in many places. These layout-sensitive and layout-insensitive styles of coding can be freely mixed within one program. Although the layout rules include many details, it is based on rather straightforward indentation rules, and in practice, curly braces and semicolons are rarely used in Haskell programs.

3.5.1. Braces and semicolons

Statements written in the layout-based style can be converted to layout-insensitive style by adding braces and semicolons in places determined by the layout rules.

In general, semicolons demarcate the end of an expression, and curly braces represent scope. For example,

```
cube x = c where { c = x * x * x; }
```

Note that an explicit open brace must be matched by an explicit close brace. Within these explicit braces, no layout processing, as described next, is performed.

3.5.2. Layout processing

The braces and semicolons are inserted as follows.

- When an open brace is omitted after the keyword `where`, `let`, `do`, or `of`, a new layout starts:
  - First, the omitted open brace is inserted at the indentation of the next token, and then
  - For each subsequent line,
    - If it contains only whitespace or is indented more, then the previous item is continued.
    - If it is indented by the same amount, then a semicolon is inserted and a new item begins, and
    - If it is indented less, then a close brace is inserted and the current layout list ends.
- When the indentation of the next token after a `where`, `let`, `do`, or `of` is less than or equal to the current indentation level, then
  - Instead of starting a layout, an empty item `{}` is inserted, and
  - Layout processing occurs for the current level.

(Note: If you are a beginner, you do not have to memorize these rules. Haskell's layout rules are rather flexible, and it will all come naturally.)

Chapter 4. Modules

A module defines a collection of entities such as values, datatypes, and classes, in an environment created by a set of imports. A module, in turn, can make some of these entities available to other modules by exporting them. Modules are used for namespace control, and they are not first class values.

A Haskell program comprises one `Main` module and possibly zero or more other modules. The `Main` module exports a value named `main`, which must be an expression of type `IO T` for some type `T`. The value of the whole program is the value of `main`.

4.1. Module Names

A module name is a sequence of one or more identifiers, separated by dots (`.`). Each identifier must begin with a capital letter.

Although it is not part of the language definition, module names can be thought of as being arranged in a hierarchy in which appending a new component (with a dot `.`) creates a child of the original module name.

Modules in standard libraries and other widely used modules tend to use a standardized set of "top-level" module names such as `System`, `Data`, and `Network`, etc. and other related modules are organized "under" this top-level module names such as `System.IO`, `Data.List`, `Data.Char`, etc. It should be emphasized, however, that it is purely a naming convention, and Haskell does not support "submodules" or other relationships among the modules.

4.2. Module Structure

Generally speaking, a module and a source code file in Haskell has a one-to-one correspondence. A Haskell module consists of two parts.

- A module begins with a header:
  - The keyword `module`,
  - The module name,
  - A list of entities to be exported (enclosed in parentheses), and
  - The keyword `where`, and the header is followed by
- A module body:
  - A possibly-empty list of import declarations that specify the modules to be imported into the current module, and
  - A possibly-empty list of top-level declarations.

In case of the `Main` module, the module declaration header can be omitted. In such a case, the header is assumed to be `module Main(main) where`.

4.3. Export Lists

An export list identifies the entities to be exported by a module declaration such as functions, types, and constructors.

If an export list is not provided, then all values, types, and classes defined in the module are automatically exported, and they will be available to anyone importing the module. Note that the entities imported from other modules are not exported in this case.

```
module MyModule where
```

Limiting the names exported is done by adding a parenthesized list of names after the module name:

```
module MyModule (MyType1, MyClassA, myFuncX) where
```

Note that all instance declarations are automatically exported with associated datatypes, and they cannot be explicitly specified in the export list.

If a module imports another module, it can also export that module, using the `module` prefix:

```
module MyModule (module Data.Set, module Data.Char) where

import Data.Set
import Data.Char
```

4.4. Import Declarations

An import declaration brings into scope the entities exported by another module. The import declaration specifies the name of a module, and it may optionally include the specific entities to be imported from that module. Imported names serve as top level declarations in the current module.

For each entity imported, both the *qualified* and *unqualified* names of the entity is brought into scope. If the import declaration uses the `qualified` keyword, however, only the qualified names of the entities are brought into scope.

An `as` clause may be used with both qualified and unqualified import statements to provide local aliases.

4.4.1. Importing all

If no specific entities are specified after the imported module name, then all the entities exported by that module are imported, including functions, data types and constructors, classes, and other re-exported modules. For instance, using the following module `M` as an example,

```
module M(X(..), y) where
data X = X
y = 1
```

The following import declaration imports both `X` and `y`.

```
import M
```

These names can be used either as qualified, e.g., `M.X` and `M.y`, or unqualified, e.g., `X` and `y`.

For a qualified import, however,

```
import qualified M
```

Only the qualified names are available in the importing module, e.g., `M.X` and `M.y` in this example. Or, we can use an `as` alias,

```
import M as M2
```

Or,

```
import qualified M as M2
```

In these two cases, the names `M2.X` and `M2.y` are brought into scope, in addition to `X` and `y` in the case of unqualified import.

Note that it is legal for more than one module in scope to use the same alias provided that all names can still be resolved unambiguously. For example,

```
module Main where
  import qualified M as M2
  import qualified N as M2
```

This is valid as long as the module N does not export names X and y.

4.4.2. Importing some or none

The imported entities can be specified explicitly by listing them in parentheses. The list may be empty, in which case only the instances are imported, if any. When the (..) form of import is used for a type or class, the (..) refers to all of the constructors, methods, or field names exported from the module.

Using the same example module M,

```
import M(X(..))                           ①
import M as M2(y)                         ②
import qualified M(X(..))                 ③
import qualified M as M2(X(..), y)        ④
```

① The names M.X and X are imported.

② The names M2.y and y are imported.

③ The name M.X is imported.

④ The names M2.X and M2.y are imported.

The following import declaration, on the other hand, imports no names from the module M.

```
import M()
```

4.4.3. Importing all but some

As a variation of the method for importing all exported names, one can explicitly exclude some names by using the form `import moduleM hiding(import1, …, importn)`. This import declaration specifies that all entities exported by the named module should be imported except for those specifically named in the list.

For example,

```
import M hiding ()                        ①
import M hiding (X)                       ②
import qualified M hiding ()              ③
import qualified M hiding (y)             ④
import qualified M as M2 hiding(X)        ⑤
```

① This brings the names `X`, `y`, `M.X`, and `M.y` into scope.

② This imports the names `y` and `M.y`.

③ This imports the names `M.X` and `M.y`.

④ This imports the name `M.X`.

⑤ This imports the name `M2.y`.

Chapter 5. Top-Level Declarations

A Haskell module can include

- Zero, one, or more top-level declarations,
  - `type` synonym declarations,
  - `newtype` declarations,
  - `data` type declarations,
  - `class` declarations,
  - `instance` declarations,
  - `default` declarations, and
- Other declarations that can be included in both top-level and nested scopes (e.g., within a `let` expression), which comprise
  - Type signatures,
  - Fixity declarations,
  - Function declarations, and
  - Pattern bindings.

These declarations can also be classified into three groups:

- User-defined data types, e.g., `type`, `newtype`, and `data` declarations,
- Type classes and overloading, e.g., `class`, `instance`, and `default` declarations, and
- The rest nested declarations, e.g., type signatures, fixities, and value bindings for both functions and patterns.

Haskell's builtin types, such as integers and floating-point numbers, and other primitive types are described in the Basic Types chapter.

5.1. Types and Classes

Haskell uses a polymorphic type system augmented with type classes. Idiomatic haskell programming styles are often based on manipulating parametrized types (aka, generic types).

5.2. Haskell Type System

Haskell's type system attributes a type to each expression during compilation. The type of an expression depends on an environment that determines the types of the variables in the expression. It also depends on a class environment if types are instances of classes. In general, a type is defined over a context for a set of type variables, typically denoted by (one letter) lowercase alphabets. For example,

```
Eq a => a -> a
```

This denotes a function which takes a value of type `a` and returns a value of the same type `a` (`a -> a`). The type constraint `Eq a` states that this function type can only be defined on the types which are instances of type class `Eq`. The most general type that can be assigned to a particular expression (e.g., in a given environment) is called its *principal type*. The Haskell type system can infer the principal types of all valid expressions. Therefore, explicit type signatures for expressions are usually not necessary.

5.3. Typeclasses

A `class` declaration introduces a new type class and a set of overloaded operations, called *class methods*. An `instance` type of that class must support those operations. An `instance` declaration declares a new type of a given type class, and it (generally) includes the implementations of the class methods.

5.4. Contexts and Class Assertions

A *context* consists of zero or more class assertions, with a general form `( C1 u1, …, Cn un )`, where `Ci ui` is a class assertion. `Ci` represents a type class identifier, and `ui` can be either a type variable, or the application of type variable to one or more types. (e.g., `Eq a` in the above example.) When there is only one type assertion, the outer parentheses can be omitted. A class identifier begins with an uppercase letter whereas a type variable begins with a lowercase letter.

In general, we write `cx => t` to indicate the constraint that the type `t` is restricted by the context `cx`. When the context is empty, we just write `t` without `=>`.

5.5. Type Syntax

Type values are built from type constructors. The names of type constructors start with uppercase letters just like data constructors. But, unlike data constructors, infix type constructors are not allowed, other than `(->)`. Type expressions have the following four main forms:

Type Variables

Type variables (or, "generic type parameters", as they are called in some other programming languages) are written as identifiers beginning with a lowercase letter, as just indicated.

Type Constructors

Here are some examples of type constructors. (Note that they are generally called "generic types" in other languages.)

- The built-in `Char`, `Int`, `Integer`, `Float`, and `Bool` are type constants. (That is, they are not "generic".)
- `Maybe` and `IO` are unary type constructors.
- `Either` is a binary type constructor.

- The declarations `data T …` or `newtype T …` introduce the type constructor `T`.

Haskell provides special syntax for certain built-in type constructors:

- The unit type constant is written as `()`, and it has one value `()`.
- The binary function type constructor is written as `(->)` (as a prefix). A function type `(->) t1 t2` can also be written, using the infix notation, as `t1 -> t2`. Function type arrows are right-associative just like in expressions. For instance, `Int -> Char -> Bool` is equivalent to `Int -> (Char -> Bool)`.
- The list type constructor is written as `[]`. A list type `[] t` can also be written as `[t]`. It denotes the type of lists with the element type `t`.
- The tuple type constructors (with two or more components) are written as `(,)`, `(,,)`, and so on. A tuple type `(, …,) t1 … tk` can also use the special syntax `(t1, …, tk)`. It denotes the type of k-tuples with its component types `t1` through `tk`.

Type Applications

A type application `t1 t2` is a type expression of types `t1` and `t2`.

Parenthesized Types

A parenthesized type of a form `(t)` is identical to the type `t`.

Notice that Haskell supports consistent syntax for expressions and their corresponding types. For example, if `t1` and `t2` are the types of expressions `e1` and `e2`, respectively, then a function `e1 -> e2`, a tuple `(e1, e2)`, and a list `[e1]` have the function type `t1 -> t2`, the tuple type `(t1, t2)`, and the list type `[t1]`, respectively.

5.6. User-Defined Types

There are three primary constructs in Haskell through which a new type or type alias can be introduced:

- The `data` declaration for creating a new algebraic datatype,
- The `newtype` declaration for creating a new type based on an existing type, and
- The `type` declaration for creating a type synonym for another type.

5.6.1. The `data` declarations

A new algebraic datatype can be declared with the `data` keyword. Datatypes, along with the record syntax, are described later in the book.

Here's a simple example:

```
data Cat = Cat Int Bool                    ①
```

① The `Cat` on the left hand side is a type constructor (with no type variables), whereas the `Cat` on the right hand side is a data constructor. A `data` type can be defined with one or more constructors. When a data type has only one constructor, it is conventional to use the same name for the type itself and its (only) data constructor. The `Cat` constructor, in this example, includes two fields of `Int` and `Bool` types.

5.6.2. The `newtype` declarations

A new type can be introduced whose representation is the same as an existing type using the `newtype` keyword:

```
newtype cx => T u1 ... uk = N t
```

This declaration creates a new type `T u1 … uk` based on, but distinct from, the type `N t`. `newtype` does not change the underlying representation of an object.

For example,

```
newtype Age = Age Int
newtype Weight = Weight Float
```

A `newtype` declaration may use the record syntax with *one field*. For example,

```
newtype Age = Age { unAge :: Int }
```

The declaration brings into scope both a constructor and a de-constructor:

```
Age   :: Int -> Age
unAge :: Age -> Int
```

5.6.3. The `type` declarations

A `type` synonym declaration introduces a new type that is *equivalent* to an old type.

```
type T u1 ... uk = t
```

This `type` declaration introduces a new type constructor, `T`. For example,

```
type LastName = String
type Perhaps = Maybe Int
type Both a = Either a a
```

Chapter 6. Nested Declarations

Nested declarations may be used in any declaration list, e.g., either at the top-level of a module or within a `where` or `let` construct.

6.1. Type Signatures

A type signature declaration specifies types for variables, e.g., patterns and functions. A type signature has the following general form, for one or more variables `v1` … `vn`:

```
v1, ..., vn :: cx => t
```

`cx` refers to a context and `t` represents a type variable or type application. This is equivalent to

```
v1 :: cx => t
...
vn :: cx => t
```

Although Haskell can deduce the *principal type* of any variable, it is conventional to include the type signature declarations for top-level variables, especially functions, in a program. In many cases, the type you want to use for a variable may not be the broadest principal type (which is generally polymorphic in Haskell).

Note that, although it is syntactically not required, the type signature declaration of a variable (almost always) immediately precedes the binding declaration of the variable.

A variable cannot be declared with more than one type signature even if the signatures are identical.

6.2. Fixity Declarations

A fixity declaration gives the fixity (or, "associativity") and binding precedence of one or more operators. A fixity declaration may appear anywhere that a type signature appears and, like a type signature, it declares a property of a particular target operator.

Also like a type signature, a fixity declaration can only occur in the same sequence of declarations as the declaration of the operator itself, and no more than one fixity declaration may be given for any operator. There are three kinds of fixity:

- Non-associativity - `infix`,
- Left-associativity - `infixl`, and
- Right-associativity - `infixr`.

There are ten precedence levels, `0` to `9`, from binding least tightly to binding most tightly. If the level is omitted, `9` is assumed. Any operator without an explicit fixity declaration is assumed to be `infixl 9`. E.g.,

```
infixl 6 `plus`
a `plus` b = a + b
```

6.3. Function Bindings

A function binding binds a variable to a function value. A function binding declaration for variable `f` has the following general form with `n` clauses, `n >= 1`:

```
f p11 ... p1k  match1
...
f pn1 ... pnk  matchn
```

where each `pij` is a pattern each `matchi` is of the general form:

```
| gsi1 = ei1
...
| gsimi = eimi
  where { declsi }
```

The expressions, `gsi1` through `gsimi`, are called the guards, and they are evaluated to the Boolean values. Pattern matching is further discussed throughout this book, especially in the `case` expressions and patterns chapters.

In case when `matchi` has a single guard that is merely `True`, it can be simply written as follows:

```
= ei where { declsi }
```

Note that

- All clauses defining a function must be contiguous, and
- The number of patterns in each clause must be the same.

For example,

```
fun :: Int -> Int -> String                ①
fun 0 0 = "Origin"                         ②
fun x 0                                    ③
  | x > 0 = "Positive x-axis"
  | x < 0 = "Negative x-axis"
fun 0 y                                    ④
  | y > 0 = "Positive y-axis"
  | y < 0 = "Negative y-axis"
fun _ _ = "Not so special"                 ⑤
```

① The general type signature declaration syntax is discussed earlier in this chapter. We further discuss what this particular signature means for functions later in the book. As indicated, it is a universal convention that the type signature for a top-level function binding is placed immediately before the biding declaration.

② This clause is equivalent to `fun 0 0 | True = "Origin"`.

③ This clause includes a pattern and a match with two guards.

④ Ditto. After the function name, `0 y` is a pattern, and the rest is a match.

⑤ The underscore symbol `_` is a wildcard pattern. The two juxtaposed patterns, in a function binding declaration as in this example, effectively represent a tuple pattern (e.g., for the two function arguments), as we further discuss later, in the context of `case` expressions.

6.4. Pattern Bindings

A pattern binding declaration binds variables to values. The general form of a pattern binding is `p match`, where a `match` is the same structure as for function bindings.

```
p | gs1 = e1
  | gs2 = e2
  ...
  | gsm = em
  where { decls }
```

The pattern `p` is matched "lazily" as an irrefutable pattern, as if there were an implicit `~` in front of it.

In case when the guard is simply `True`, the pattern binding has the simple form:

```
p = e
```

For example,

```
x :: Int                              ①
x = 3                                 ②

a, b :: Int
(a, b) | x > 0 = (3, 4)               ③
       | x < 0 = (-3, -4)
       | otherwise = (0, 0)
```

① A type signature declaration for the following pattern binding. Note that `Int` is the type of the value of the expression `3` in this example. We discuss what is an "expression" in Haskell throughout the book.

② A simple pattern binding. Note that, in other more traditional programming languages this kind of syntax may be called a variable declaration and/or variable assignment, etc. In Haskell, the expression on the left-hand side is a pattern (which is clearly more general and more flexible than just using "names" in other languages). This particular pattern binding declaration is equivalent to `x | True = 3`.

③ A slightly more general pattern binding example. The value `otherwise` is a synonym for `True`.

Chapter 7. Basic Types

The Haskell Prelude contains predefined classes, types, and functions that are implicitly imported into every Haskell program.

The following types are defined in the Prelude:

- The boolean type, `Bool`,
- Numeric types, `Int`, `Integer`, `Float`, and `Double`, etc.,
- `Char` and `String`,
- `Lists`,
- `Tuples`,
- `Maybe`, `Either`, `Ordering`, and
- `IO` and `IOError` Types.

In addition, Haskell defines the unit `()` datatype, which represents a void value, and an implicit type "Bottom" `_|_`, which is included in every type.

7.1. Booleans

The boolean type `Bool` is an enumeration.

```
data Bool = False | True
  deriving (Read, Show, Eq, Ord, Enum, Bounded)
```

7.1.1. Boolean functions

The basic boolean functions are `&&` (and), `||` (or), and `not`. The name `otherwise` is defined as `True` to make guarded expressions more readable.

```
(&&) :: Bool -> Bool -> Bool
(||) :: Bool -> Bool -> Bool
not :: Bool -> Bool
otherwise :: Bool
```

For example,

```
ghci> [True && True, True && False, False && True, False &&
False]
[True,False,False,False]
ghci> [True || True, True || False, False || True, False ||
False]
[True,True,True,False]
ghci> [not True, not False]
[False,True]
ghci> otherwise
True
```

7.2. Characters

Haskell's character type `Char` is an enumeration whose values represent Unicode characters. Character literals, e.g., `a`, `Z`, and `#`, are nullary constructors in the datatype `Char`.

Type `Char` is an instance of the classes `Read`, `Show`, `Eq`, `Ord`, `Enum`, and `Bounded`. The `toEnum` and `fromEnum` functions, from the `Enum` class, map characters to and from the `Int` type, respectively. For example,

```
ghci> toEnum 65 :: Char
'A'
ghci> fromEnum 'a' :: Int
97
```

7.3. Strings

`String` in Haskell is an alias for a list of chars. That is,

```
type String = [Char]
```

For example,

```
ghci> h = "hello world"
ghci> import Data.Char
ghci> map toUpper h
"HELLO WORLD"
```

A string literal may include a "gap", that is, a pair of backslashes enclosing one or more whitespace characters, including newlines. Gaps are ignored, which allows writing "multi line" strings in Haskell. For example,

```
ghci> :{                                        ①
ghci| truth = "It's not\      \enough\          ②
ghci|          \ to speak, \
ghci|          \but to speak true."
ghci| :}
ghci> putStrLn truth                            ③
```

① GHCI accepts multi-line commands with this syntax, using a pair of opening and closing symbols, *:{* and *:}*.

② Note that there are three backslash characters. The first two match and form a gap. The third one pairs with the one at the beginning of the next line.

③ This will output *It's notenough to speak, but to speak true.*

7.4. Numbers

The Prelude defines a few basic numeric types:

- Fixed sized integers (`Int`),
- Arbitrary precision integers (`Integer`),
- Single precision floating (`Float`), and
- Double precision floating (`Double`).

Other numeric types such as rationals and complex numbers are defined in libraries. The class `Num` of numeric types is a subclass of `Eq`, since all numbers may be compared for equality. Its subclass `Real` library is also a subclass of `Ord`, since the order comparison operations apply to all but complex numbers.

7.4.1. Numeric operators

The following operators for arithmetic computations are defined in the Prelude:

```
(^)  :: (Num a, Integral b) => a -> b -> a
(^^) :: (Fractional a, Integral b) => a -> b -> a
(**) :: Floating a => a -> a -> a

(*)  :: Num a => a -> a -> a
(/)  :: Fractional a => a -> a -> a
quot :: Integral a => a -> a -> a
rem  :: Integral a => a -> a -> a
div  :: Integral a => a -> a -> a
mod  :: Integral a => a -> a -> a

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

`(^)`, `(^^)`, and `(**)` are exponent operators. Note that `` `quot` ``, `` `rem` ``, `` `div` ``, and `` `mod` `` are usually used as infix operators.

7.4.2. Numeric functions

In addition, the following functions are also defined in the Prelude for numeric types:

```
subtract     :: (Num a) => a -> a -> a
even, odd    :: (Integral a) => a -> Bool
gcd          :: (Integral a) => a -> a -> a
lcm          :: (Integral a) => a -> a -> a
fromIntegral :: (Integral a, Num b) => a -> b
realToFrac   :: (Real a, Fractional b) => a -> b
```

What these functions do should be rather self-evident even if you haven't used Haskell before. `gcd` and `lcm` stand for greatest common divisor and least common multiple, respectively. Note that the distinction between operators and functions is rather subtle in Haskell. This is discussed later in the Expressions chapter.

7.5. The Unit Datatype

The unit type `()` is an enumeration with one nullary constructor `()`. Type `()` is an instance of `Read`, `Show`, `Eq`, `Ord`, `Bounded`, and `Enum`.

```
ghci> [() == (), () /= ()]
[True,False]
ghci> [minBound :: (), maxBound :: ()]
[(),()]
ghci> fromEnum () :: Int
0
ghci> toEnum 0 :: ()
()
```

7.6. Maybe

The `Maybe` datatype, defined in the Prelude, consists of two constructors `Nothing` and `Just a`.

```
data Maybe a = Nothing | Just a
  deriving (Eq, Ord, Read, Show)
```

The `Maybe` type derives from `Eq`, `Ord`, `Read`, and `Show`. In addition, `Maybe` is an instance of classes `Functor`, `Monad`, and `MonadPlus`.

The Prelude also includes `maybe` function, which takes a value `n`, a function `f`, and a value of `Maybe` type and returns the first value `n` if the `Maybe` value is `Nothing` or `f x` if the `Maybe` value is `Just x`.

```
maybe :: b -> (a -> b) -> Maybe a -> b
```

For example,

```
ghci> maybe 0 (+ 10) Nothing
0
ghci> maybe 0 (+ 10) (Just 2)
12
```

7.7. Either

The `Either` datatype consists of two constructors `Left` and `Right`, and it derives from `Eq`, `Ord`, `Read`, and `Show`.

```
data Either a b = Left a | Right b
  deriving (Eq, Ord, Read, Show)
```

The `either` function takes two functions and a value of `Either`, and it invokes the first function or the second function depending on whether the given value is the `Left` or `Right` variant, respectively.

```
either :: (a -> c) -> (b -> c) -> Either a b -> c
either f g (Left x)  =  f x
either f g (Right y) =  g y
```

For example,

```
ghci> either (* 2) (+ 10) (Left 3)
6
ghci> either (* 2) (+ 10) (Right 5)
15
```

7.8. Ordering

```
data Ordering = LT | EQ | GT
  deriving (Eq, Ord, Enum, Bounded, Read, Show);
```

The `Ordering` datatype is used to represent "greater than", "less than", and "equal to" relationships. For example,

```
ghci> :{
ghci| cmp :: Int -> Int -> Ordering
ghci| cmp x y
ghci|    | x > y = GT
ghci|    | x < y = LT
ghci|    | otherwise = EQ
ghci| :}
ghci> [cmp 1 3, cmp 3 1, cmp 3 3]
[LT,GT,EQ]
```

7.9. Bottom

The pseudo-type "Bottom" `_|_` is a subtype of all types in Haskell. It is an empty type. That is, it does not have a value of its own kind. The bottom refers to a computation which does not return a value in Haskell, e.g., due to some kind of errors, or because the computation never terminates (and, hence does not return a value). The `undefined` value can be used in situations where a value of bottom is needed.

7.10. The `IO` Type

The `IO` type serves as a tag for operations (actions) that interact with the outside world. `IO` is a unary type constructor, and it is an abstract type. No data constructors are visible to the user. `IO` is an instance of the `Functor` and `Monad` classes. We discuss the basic I/O and I/O-related functions at the end of the book.

7.11. The `IOError` Type

`IOError` is also an abstract type, representing errors raised by I/O operations. It is an instance of the Show and Eq classes. Values of this type are constructed by various I/O functions, including the `userError` function defined in the Prelude.

Chapter 8. Expressions

Haskell is based on lambda calculus. But, as a high-level programming language, it provides syntax for expressions and what not. In the following few chapters, we describe the syntax and informal semantics of Haskell expressions.

8.1. Variables

Haskell, as a pure functional programming language, has no concept of "updating". That is, a value does not contain any mutable state. Variables are bound to values via the pattern binding declarations. The same variable can be bound to different values, even within the same scope. The new binding "shadows" the earlier bindings.

8.2. Literals

In Haskell, numeric literals are polymorphic.

- An integer literal is a syntactic shorthand for applying `fromInteger` to the given value of type `Integer`.
- A floating point literal is a shorthand notation of an application of `fromRational` to the given value of type `Rational`.

8.3. Operators

Haskell provides special syntax for "operators". An operator is a function that can be applied using infix notation, or partially applied using a section. An operator is either an operator symbol, e.g., `++`, or is an ordinary identifier in back quotes, e.g., `` `op` ``. That is, `` x `op` y `` is semantically equivalent to `op x y`. In reverse, an operator symbol can be converted to an ordinary identifier by enclosing it in parentheses.

Haskell's "builtin" operators (e.g., from the Prelude) have the following fixity declarations (operator precedence and associativity):

```
infixr 9  .                                 ①
infixr 8  ^, ^^, **
infixl 7  *, /, `quot`, `rem`, `div`, `mod`
infixl 6  +, -
infix  4  ==, /=, <, <=, >=, >
-- infixr 5  :                              ②
infixr 3  &&
infixr 2  ||
infixl 1  >>, >>=
infixr 1  =<<
infixr 0  $, $!, `seq`
```

① This is a function composition operator. In Haskell, the function application syntax (which is not an operator) has the highest precedence (it's literally off the chart ☺), and it is left-associative. The next in line is the function composition, which is right-associative (as indicated by infix`r`).

② The cons operator `:` is also a builtin syntax, and not a declared operator. But, if a fixity declaration were given, it would be `infixr 5 :`. The fixity declaration syntax (e.g., for user-defined operators) is explained later in the book.

> A lot of beginning Haskell programmers find Haskell difficult. They generally attribute this difficulty to FP. That is, however, most likely not the case. The initial difficulty that beginners face is the syntax, not the *functional programming*. For instance, these fixity rules are, although trivial in a sense, one of the most difficult to learn, or to get used to. In imperative programming, this is not that significant, in which we rarely use long expressions. In functional programming, on the other hand, we deal with (only) expressions. Sometimes, long

> expressions. Despite this, or possibly because of this, the use of parentheses are generally discouraged in Haskell when they are not necessary. Therefore, you will have to know these fixity rules by heart to be able to read (and, write) Haskell code.

8.4. Errors

Errors during expression evaluation, denoted by `_|_` ("bottom"), are indistinguishable by a Haskell program from non-termination. Since Haskell is a non-strict language, all Haskell types include `_|_`. That is, a value of any type may be bound to a computation that, when demanded, results in an error. When evaluated, errors cause immediate program termination and cannot be caught by the user.

8.5. The `error` and `undefined` Functions

8.5.1. The `error` function

`error` stops execution and displays an error message.

```
error :: String -> a
```

8.5.2. The `undefined` value

When `undefined` is used, the error message is created by the compiler.

```
undefined :: a
undefined = error "Prelude.undefined"
```

Chapter 9. Functions

A function is an abstract type, and they do not have constructors. A function value is created by declaring its name, zero or more parameters, and an equal sign `=`, followed by an expression, which is the definition of the function. All function names must start with a lowercase letter or `_`. For example,

```
incrBy1 :: Int -> Int                   ①
incrBy1 x = x + 1                       ②
```

① A type signature, immediately preceding the function binding.

② This notation suggests that if you apply the function `incrBy1` to `x`, its value will be `x + 1`.

9.1. Function Applications

Function application is written as, `e1 e2`. Application associates left. That is, `x y z` is equivalent to `(x y) z`, for instance. This syntax is somewhat unusual in that, in mathematics, and in fact in the vast majority of programming languages, function application uses the parentheses notation. However, the Haskell syntax, based on lambda calculus, is *the* most efficient notation for function application, which is at the heart of everything else in Haskell.

For example,

```
f1 :: Int -> Int -> Int -> Bool
f1 x y z = (x > y) && (y > z)           ①

main = do                               ②
  print $ f1 5 4 3                      ③
  print $ f1 3 3 1                      ④
```

① As described in the Nested Declarations chapter, a function binding uses patterns. In this example, the triple `x y z`, after the function name `f1`, is an (implicit) tuple pattern comprising three variable patterns. This is an irrefutable pattern, meaning that any valid application will match this clause, and we do not need, and cannot have, any other clauses below this line.

② As indicated earlier, the value `main` has a polymorphic type `IO a`. In all examples in this book, which is also generally the case in practice, the type of `main` is almost always `IO ()`. Hence, we will generally omit the type signature for `main` in this book. The `do` notation is explained near the end of the book, in the context of the I/O. But, in effect, `do` allows us to use "imperative style" programming. In this example, the `do` expression includes two `print` expressions, which are processed sequentially one after the other. Note that we almost always use the layout-sensitive coding style. That is, the curly braces enclosing these two `print` expressions in this example are omitted by using the indentation rules.

③ An example of function application, `f1 5 4 3`. Note the similarity between the function binding pattern and the function application syntax. This application evaluates to `True`, in this example. `print` is one of the builtin I/O functions that we use throughout this book without first defining them. It prints the given value to the terminal. The lazy infix application operator `$` is explained later in the Core Functions chapter. Since function applications are left-associative, `print f1 5 4 3` would have had a different meaning (and, in fact, syntactically invalid). We could have done `print (f1 5 4 3)`, but the syntax with fewer parentheses is generally preferred in Haskell.

④ `f1 3 3 1` evaluates to `False`. As we will discuss shortly, the `f1` function can be either viewed as taking three arguments (and returning a value), or it can be viewed as taking one argument (and returning a function). `f1 3 3 1`, `(f1 3) 3 1`, and `((f1 3) 3) 1` are all syntactically equivalent, and they are also semantically equivalent through currying.

9.2. Operator Applications

Application of a binary operator `op` on `e1` and `e2`, e.g., `(op) e1 e2` can be written as infix application, `e1 op e2`. Likewise, application of a binary function, e.g., `f e1 e2`, can be also written with an infix form, ``e1 `f` e2``. Note that, lexically, operators belong to two categories, operator symbols and ordinary identifiers.

Here are a couple of example functions to demonstrate the infix-based function application syntax:

```
(+*+) :: Int -> Int -> Int
x +*+ y = x + 2 * y                       ①
mold :: Int -> Int -> Int
x `mold` y = x * (y + 2)                  ②
```

① Alternatively, `(+*+) x y = x + 2 * y`. Note that Haskell allows defining any arbitrary operators, in particular, using operator symbols. But, as the saying goes, *with the great power comes the great responsibility*.

② Or, `mold x y = x * (y + 2)`.

Then, we can use them as follows, for instance:

```
main = do                                 ①
  print $ 5 +*+ 10                        ②
  print $ (+*+) 10 5
  print $ mold 1 2
  print $ 2 `mold` 1
```

① As indicated, the `main` function signature `main :: IO ()` is always omitted in this book.

② These four `print` function applications will output `25`, `20`, `4`, and `6`, to the terminal.

9.3. Lambda Abstractions

Functions can also be declared anonymously. For example, an expression, `\x -> x * x`, defines a function which takes one argument and returns its squared value. Anonymous functions, also called lambdas or lambda expressions, are useful for simple functions that need not be separately declared first.

As with (regular) functions, a lambda is just a value in Haskell, which has a function type. For example,

```
squareAll :: [Int] -> [Int]
squareAll = map (\a -> a * a)                ①
biggerThan :: [Int] -> ([Int] -> [Int])      ②
biggerThan n = \xs -> filter (> n) xs        ③
```

① The `map` function takes two arguments. In this example, only one (e.g., a lambda function) is given. This is called the partial application. It is useful for currying and sections, for example.

② Note that the part in the parentheses in this type signature is the type of the lambda function on the right hand side of the function binding in the next line. The parentheses in this example is redundant, as we discuss next.

③ This is for illustration only. Lambdas are typically declared at the point of use, and they are rarely given names. Note that, by convention, the variables that end with `s` are lists. E.g., `zss` could refer to a list of lists (because it ends with two `s`'s). The builtin `filter` function is discussed later. `(> n)` is a section.

A general lambda abstraction can be written as

```
\ p1 ... pn -> e
```

where the `pi` are patterns. Note that the backslash character in the lambda syntax is supposed to represent the Greek Lambda character. An example lambda function with two arguments:

```
main = do
  let lamb = \x y -> 2 * x + y          ①
  print $ lamb 5 10                      ②
```

① Again, for illustration only. This `let` binding with a lambda function is the same as a function binding, `lamb x y = 2 * x + y`. Note that, unlike in the case of regular functions, a lambda function cannot have more than one pattern clause.

② This will print out `20` to the terminal.

9.4. Curried Applications

As indicated, function applications are left-associative in Haskell, and a function that takes `n` arguments, e.g., `f e1 e2 … en`, is equivalent to a function that takes `n-1` arguments, e.g., `g e2 … en`, if `f e1 == g`. The expression `f e1` is called *partial application*.

Hence, `f` can be viewed as a function that takes one argument (`e1`) and returns a function (`g`) that takes `n-1` arguments (`e2 … en`). Likewise, function application of `g` that takes `n-1` arguments, `g e2 … en`, is equivalent to `h e3 … en` if `g e2 == h`. Therefore, again the function `g` can be viewed as a function that takes one argument (`e2`) and returns another function (`h`) that takes `n-2` arguments (`e3 … en`). We can continue this process down to the level where the last function takes one argument and returns a simple value (e.g., a function takes zero arguments).

(In pure functional programming languages like Haskell, a function that takes zero arguments must return a constant value. There are *no* other options, unlike in other impure languages, as you can easily convince yourself. Hence, there is a one-to-one correspondence between a simple value and a nullary function that returns that value. In fact, they are equivalent in Haskell.)

Converting a function that takes `n` arguments, `n >= 2`, to a functional form that takes one argument and returns one value, i.e., a function, is called "currying". In Haskell, there is little difference between these two forms, and in fact we do not need "conversion". Syntactically, Haskell does not really distinguish these two interpretations. Therefore, we consider all functions in Haskell take one argument and return one value. This is manifested, for example, in the function type notations. All functions in Haskell are curried functions.

9.4.1. An informal illustration

As an example, let's consider the following three functions, `f1`, `f2`, and `f3`, which have different *arities*, e.g., 1, 2, and 3, respectively.

```
f1 :: Int -> Int
f1 c = 5 + 3 * c

f2 :: Int -> Int -> Int
f2 b c = 1 + 2 * b + 3 * c

f3 :: Int -> Int -> Int -> Int
f3 a b c = a + 2 * b + 3 * c
```

As indicated, the function application is left-associative, whereas the arrows in the function type signatures associate right in Haskell. (This illustration will show you why that is chosen to be the case.)

The type signature of `f3` is, therefore, equivalent to `f3 :: Int -> (Int -> Int -> Int)`. In this (curried) interpretation, the `f3` function takes one argument of type `Int` and returns one value of type `Int -> Int -> Int`, which happens to be the type of the function `f2`.

The type of `f2` is `f2 :: Int -> (Int -> Int)`, which indicates that `f2` takes one value of `Int` and returns one value of `Int -> Int`, which happens to be the type signature of `f1`. The `f1` function also takes one value (of type `Int`) and returns one value (of type `Int`).

We have deliberately chosen the implementations of these three functions. Now, let's trace back. The `f3` function takes three arguments and returns one value, in the conventional (non-curried) view:

```
f3 :: Int -> Int -> Int -> Int          ①
f3 a b c = a + 2 * b + 3 * c
```

① Haskell could have chosen different notations for multi-argument functions (e.g., something like `(Int, Int, Int) -> Int`), but they didn't. The illustration in this section will convince you why that was not necessary.

This is, however, equivalent to

```
f3 :: Int -> (Int -> Int -> Int)
(f3 a) b c = a + 2 * b + 3 * c
```

That is, the partial application `f3 a` is a function that takes two `Int` arguments and returns an `Int` value. `f3 a` happens to be the same as `f2` when `a` happens to be `1`. Likewise,

```
f2 :: Int -> (Int -> Int)
(f2 b) c = 1 + 2 * b + 3 * c
```

The partial application `f2 b` is a function that takes an `Int` value and returns an `Int` value. `f2 b` happens to be the same as `f1` when `b == 2` (again, in this deliberately constructed example). That is,

```
f1 :: Int -> Int
f1 c = 5 + 3 * c
```

Hence, there is no difference between `f2`, which takes two arguments `b` and `c` and returns one `Int` value and `(f2 b)`, which returns a function that takes one argument `c` and in turn returns an `Int` value. Likewise, there is no difference between `f3`, which takes three arguments `a`, `b`, and `c` and returns one `Int` value and `(f3 a)`, which returns a function that takes two arguments `b` and `c` and returns an `Int` value.

Note also that the left-associativity of function applications and the right-associativity of arrows in the function types dovetail well with each other. (Notice the respective positions of the (optional) parentheses we've added in these examples.)

9.5. Sections

Sections are a syntactic shorthand for partial application of binary operators. For example, using the multiplication `*` operator,

```
triple = (*) 3                 ①

main = do
  print $ triple 10            ②
```

① `triple` is a function that takes one argument since the other argument (of the binary `(*)`) has been partially applied with a value `3`. Note that this pattern binding is essentially equivalent to a function binding with one clause, `triple x = ((*) 3) x` (which is in turn equivalent to `triple x = 3 * x`). Its type signature is `triple :: Int -> Int`. As one can easily see, the syntactic difference between pattern bindings and function bindings are somewhat superficial.

② This will print 30.

Now, using triple instead of (*) 3 has some syntactic convenience since we do not have to use many parentheses, e.g., triple 10 vs ((*) 3) 10. Section provides this syntactic convenience without having to create a new binding. For example, this triple function can be written as (3 *). (Note the order.)

Another, possibly more important, advantage of sections is that we can supply the argument either on the left or right hand side, unlike in the case of general partial applications, in which arguments are consumed from left to right. That is, (op e1) and (e1 op) are generally two different sections. (Multiplication happens to be commutative, and hence (e *) and (* e) are effectively the same function.)

Formally, given a binary operator op and an expression e, a right section is written as

```
e op                                          ①
```

① This is equivalent to the normal partial application form, (op) e. (Note the difference between the infix and prefix notations.)

Likewise, a left section for op and e is written as

```
op e                                          ①
```

① This form has no corresponding partial application form.

The right section (e op) is syntactically valid if and only if (e op x) parses in the same way as ((e) op x). Likewise, the left section (op e) is syntactically valid if and only if (x op e) parses in the same way as (x op (e)).

9.6. Function Composition

Function composition (`.`) plays as an essential role as function application in Haskell. The builtin function composition operator `.` composes two given functions.

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
```

Composing a function `h` (`a -> b`) with `g` (`b -> c`), i.e., `g . h`, yields a function from `a` to `c` (`a -> c`). (Note the order.)

`(g . h) x` is defined to be `g (h x)`. That is, if we set `f = g . h`, then `f x = g (h x)`. Note that a function (partial) application `g h` would have associated left. That is, for a given argument `x`, the function application would have been `(g h) x`, or `g h x` (which is syntactically invalid in this example). On the other hand, `g . h` applied to `x` would yield a different value, `g (h x)`. For example,

```
fnOne :: Int -> Int
fnOne x = x + 1
fnTwo :: Int -> Int
fnTwo x = 2 * x
fnCombo :: Int -> Int
fnCombo = fnTwo . fnOne

main = do
  print $ (fnTwo . fnOne) 3              ①
  print $ fnCombo 3                      ②
```

① This will print *8*.

② The same. Note that `fnCombo x = 2 * (x + 1)`. The power of function composition often comes from the fact that we can manipulate, and compute, functions without applying them first to any specific values.

Chapter 10. Lists

The list literal, `[e1, …, ek]`, represents a list of `k` expressions, `e1`, `e2`, … through `ek`. The empty list is denoted `[]`.

In Haskell, the list data constructor is a special operator `:` (or, "cons"). Lists are an `instance` of classes, `Read`, `Show`, `Eq`, `Ord`, `Functor`, `Monad`, and `MonadPlus`. Standard operations on lists defined in the Prelude are included later in the book.

10.1. List Constructors

Lists are an algebraic datatype with two constructors, albeit with special syntax. The first constructor is the null list, written `[]` ("nil"), and the second is `:` ("cons"). For example,

```
main = do
  let a = [] :: [Int]                    ①
  print a
  let b = 1 : ([] :: [Int])              ②
  print b
  let c = 'a' : ['d', 'e', 'f']          ③
  print c
  let d = 'g' : c                        ④
  print d
```

① A nil constructor for `[Int]` list. `a` is an empty list of type `[Int]`. The `print` function in the next line will print `[]`.

② A cons constructor with two arguments, `1` and an empty `[Int]` list. `b` is `[1]`.

③ A cons constructor with `'a'` and a `[Char]` list, `['d', 'e', 'f']`. `c` is `"adef"`, or `['a', 'd', 'e', 'f']`.

④ Another cons constructor example. `d` is `"gadef"`.

Note that, for example, `[1, 2, 3, 4]` is the same as `1 : 2 : 3 : 4 : []`, which is the same as `1 : (2 : (3 : (4 : [])))`. (The builtin cons operator is right-associative.) In general, a list literal is a shorthand for the constructor expressions with each element subsequently added to the head.

```
main = do
  print $ 'L' : 'i' : 's' : 't' : []     ①
```

① This will print *"List"*.

10.2. Enumerations

Haskell supports a special syntax for creating a list with enumerable elements. This is called the "arithmetic sequences" (or, "ranges" or "enumerations", etc.). Syntactically, it can take one of the following four forms:

```
[ exp1 .. ]
[ exp1, exp2 .. ]
[ exp1 .. exp3 ]
[ exp1, exp2 .. exp3 ]
```

That is, `exp2` and `exp3` are optional, while it requires `[ exp1 .. ]`. The expressions, `exp1`, `exp2`, and `exp3`, should be of type `t`, which is an instance of class `Enum`. Any of these arithmetic sequences denotes a list of type `[t]`. They are defined as follows:

- `[exp1 ..]` == `enumFrom exp1`
- `[exp1, exp2 ..]` == `enumFromThen exp1 exp2`
- `[exp1 .. exp3]` == `enumFromTo exp1 exp3`
- `[exp1, exp2 .. exp3]` == `enumFromThenTo exp1 exp2 exp3`

When `exp3` is omitted, it is assumed to be the biggest element for the given `Enum` type `t`. Otherwise, the semantics of arithmetic sequences are entirely dependent on the type `t`. In cases of numeric types, `exp1` is the first element, and `exp2 - exp1` represents the "step". For example,

```
main = do
  print [5 .. 10]                          ①
  print [2, 4 .. 11]                       ②
  print $ take 5 [1 .. ]                   ③
  print $ take 5 [2.0, 5.0 .. ]            ④
```

① This will print `[5,6,7,8,9,10]`. Note that the last element (`exp3`) is inclusive.

② This will print `[2,4,6,8,10]`.

③ This will print `[1,2,3,4,5]`. Note that `[1 .. ]` is an infinite list, with the `Integer` element type.

④ This will print `[2.0,5.0,8.0,11.0,14.0]`.

Another example, using `Char` elements,

```
main = do
  print ['d' .. 'h' ]                      ①
  print ['d', 'f' .. 'k' ]                 ②
  print $ take 10 ['w' .. ]                ③
  print $ take 10 ['t', 'v' .. ]           ④
```

① This will print `"defgh"`.

② This will print `"dfhj"`.

③ This will print `"wxyz{|}~\DEL\128"`. Note that `Char` type is bounded. That is, `['w' .. ]` is not an infinite list.

④ This will print `"tvxz|~\128\130\132\134"`.

10.3. List Comprehensions

List comprehensions are now widely supported by many different programming languages, including Scala and Python.

A list comprehension in Haskell has the following general syntax:

```
[ exp | q1, ..., qi, ..., qn ]
```

Here `n` is equal to, or bigger than, `1`, and each qualifier `qi` can be one of the following three forms:

- Generators of the form `pat <- exp`, where `pat` and `exp` are patterns and expressions of types `t` and `[t]`, respectively,
- Boolean expressions known as *guards*, to filter preceding generators, and
- Local `let` bindings that are to be used in the generated expression `exp` or subsequent boolean guards and generators.

A list comprehension evaluates the target expression `exp` in the successive environments, from left to right, which are created by evaluating the generators in the qualifier list.

Note that, in the list comprehension, pattern matching in a generator is simply used for filtering. That is, if a match fails then that element of the list is just excluded from the resulting list.

Here are some examples:

```
main = do
  let c1 = [x * x | x <- [1 ..]]          ①
  print (take 5 c1 :: [Integer])          ②
```

① An infinite list of squared integer values. This list comprehension includes one generator, `x <- [1 ..]`, which uses the enumeration syntax.

② This will output *[1,4,9,16,25]* to the terminal.

```
divisors :: Int -> [Int]
divisors n = [d | d <- [1 .. n],
              n `mod` d == 0]          ①

main = do
  print (divisors 10, divisors 12)     ②
```

① A Boolean guard. This guard is used as a filter for the "divisors" of the given `Int` argument.

② This line will print *([1,2,5,10],[1,2,3,4,6,12])*.

```
main = do
  let c2 =
        [ (a, b)
        | a <- [1 .. 5] :: [Int]
        , b <- [1 .. 5] :: [Int]
        , let s = a + b                ①
        , s >= 3                       ②
        , s <= 4                       ③
        ]
  print c2                             ④
```

① A local `let` binding, whose value is used in the subsequent guards.

② A Boolean guard.

③ Another guard. These two guards could have been combined as one guard `s >= 3 && s <= 4` in this example.

④ The output: *[(1,2),(1,3),(2,1),(2,2),(3,1)]*.

Chapter 11. Tuples

Tuples are algebraic data types with special syntax, `(e1, …, ek)`. A tuple size must be equal to, or greater than, `2`, but there is no preset upper bound, other than practical limitations. A compliant Haskell implementation is required to support tuples up to size `15`.

All tuples are instances of `Eq`, `Ord`, `Bounded`, `Read`, and `Show`, that is, as long as all their component types are.

For example,

```
apply :: (t -> a, t -> b) -> t -> (a, b)
apply (f1, f2) list = (f1 list, f2 list)

main :: IO ()
main = do
  print (True, 'A', "Haskell")          ①
  print $ apply (head, tail) [1, 2, 3]  ②
```

① `(True, 'A', "Haskell")` is a 3-element tuple of a `Bool`, a `Char`, and a `String`.

② `(head, tail)` is a 2-element tuple of functions. Note the definition of `apply` which takes a pair of functions as its first argument.

11.1. Tuple Constructors

The constructor for an n-tuple is written as `(, … ,)` with `n-1` commas, e.g., by omitting the expressions surrounding the commas in an n-tuple. Hence, for instance, `(,,) a b c` constructs a tuple `(a, b, c)`.

Likewise, the tuple type constructor has a similar syntax, as described earlier in the book. For instance, `(,,) Bool Char Int` denotes the same type as `(Bool, Char, Int)`.

As an example,

```
main = do
  let x = (,,) 'a' True 'z'           ①
  print x                             ②
```

① Variable `x` has a type `(Char, Bool, Char)`, or `(,,) Char Bool Char`.

② This will print *('a',True,'z')*.

11.2. Tuple Functions

The following functions are defined in the Prelude for pairs (2-tuples):

```
fst     :: (a,b) -> a
snd     :: (a,b) -> b
curry   :: ((a, b) -> c) -> a -> b -> c
uncurry :: (a -> b -> c) -> (a, b) -> c
```

11.2.1. The `fst` and `snd` functions

- The `fst` function takes a pair and it returns its first element, e.g., `fst (x,y)` returns `x`.
- The `snd` function takes a pair and it returns its second element, e.g., `fst (x,y)` returns `y`.

For example,

```
main = do
  let pair = ("Hello", 42 :: Int)
  print $ fst pair                    ①
  print $ snd pair                    ②
```

① The output: *"Hello"*

② The output: *42*

11.2.2. The `uncurry` and `curry` functions

- The `uncurry` function takes a (curried) function that accepts two arguments and converts it to a function which takes a single argument of a pair type. That is, `uncurry f pair` is defined to be `f (fst pair) (snd pair)`. (Note that, since function applications are left-associative, `uncurry f pair` is the same as `(uncurry f) pair`.)
- The `curry` function converts an uncurried function that takes a pair into a (regular) curried function. That is, `curry ucf x y`, or `(curry ucf) x y`, is defined to be `ucf (x, y)`.

For instance,

```
addFn :: Int -> Int -> Int                    ①
addFn a b = a + 2 * b

uncurriedAddFn :: (Int, Int) -> Int           ②
uncurriedAddFn = uncurry addFn

pairFn :: (Int, Int) -> Int                   ③
pairFn (a, b) = 2 * a - b

curriedPairFn :: Int -> Int -> Int            ④
curriedPairFn = curry pairFn
```

① A "regular" function.

② An uncurried version of `addFn`.

③ A function that takes a pair.

④ A curried version of `pairFn`.

Here are a few simple examples of using these functions:

```
main = do
  print $ addFn 1 2                    ①
  print $ uncurriedAddFn (1, 2)        ②
  print $ pairFn (1, 2)                ③
  print $ curriedPairFn 1 2            ④
```

① The output: *5*

② The same output: *5*

③ The output: *0*

④ The same output: *0*

11.3. The Unit and Parenthesized Expressions

The unit expression `()` has type `()`, whose only member is `()` (other than the bottom `_|_`). `()` can be thought of as the "nullary tuple" (with zero elements). (That is, the unit notation using the tuple-like syntax is not a coincidence.)

Haskell does not support one-element tuple types unlike in some other programming languages. The form `( exp )` is simply a parenthesized expression, and it is equivalent to `exp`. From the viewpoint of algebraic data types, a single element tuple type is no different from the element type itself. That is, a (hypothetical) type `(t)` must be the same type as `t`, and a single element tuple cannot be a distinct type in Haskell.

Chapter 12. Expression Type Signatures

Expression type signatures have the following two forms:

```
exp :: t
exp :: cx => t
```

where `exp` is an expression and `t` is a type. The context `cx` is optional, as in normal type signature declarations.

Expression type signatures may be used

- To explicitly type an expression, or
- To resolve ambiguous typings due to overloading.

As with normal type signatures,

- The declared type may be more specific than the principal type derivable from `exp`, but
- It is illegal to give a type that is more general than, or not comparable to, the principal type.

For example,

```
addTwoNums :: Num a => a -> a -> a        ①
addTwoNums x y = x + y

main = do
  print $ addTwoNums (1 :: Int) 2         ②
  print $ addTwoNums (1.0 :: Float) 2.0   ③
  -- addTwoNums (1 :: Int)(2 :: Integer)  ④
```

① A type signature for the `addTwoNums` function. Note that it uses the most general type which supports addition `+` for the operands and return values. This is also the function's principal type.

② The integer `1` is polymorphic, but we explicitly declare it as `Int` using the expression type signature syntax. Note that, in this example, `2` is also of the `Int` type (without requiring another explicit expression type signature).

③ Likewise, `1.0` and `2.0` are both of the `Float` type.

④ This will cause a compile error since the type annotation is not consistent with the type signature of the `addTwoNums` function.

Here's another example, in which ambiguity arises as to what type Haskell is supposed to use for an expression whose type is not explicitly specified in the type signature declaration. This is the so-called "show . read" problem.

```
readAndShow :: String -> String
-- readAndShow x = show (read x)        ①
readAndShow x = show (read x :: Int)    ②

main = do
  print $ readAndShow "300"             ③
  print $ readAndShow "abc"             ④
```

① Haskell cannot compile this function because it does not know the type of `read x`. We must limit the type through an annotation.

② We use an explicit expression type signature to indicate that the type of `read x` is `Int`. Note that because of the precedence rules, `read x :: Int` is the same as `(read x) :: Int`. The function application binds most tightly in Haskell.

③ This will print *"300"*.

④ This will return an error, *type: Prelude.read: no parse*.

Chapter 13. Let and Where

13.1. The `let - in` Expression

A `let` expression introduces a nested and possibly mutually-recursive list of declarations, with the following general form:

```
let { d1 ; ... ; dn } in exp
```

Here, `exp` is an expression. The value of `exp` is the value of the overall `let` expression.

Each declaration `di` is translated into an equation of the form `pi = ei`, where `pi` and `ei` are patterns and expressions, respectively. The `let` declarations are lexically-scoped.

For example, in its simplest form,

```
multiples :: Int -> [Int]
multiples x =
  let mult n = n * x                    ①
   in map mult [1 .. 10]                ②

main = do
  print $ multiples 10                  ③
```

① This `let` expression binds `mult n` to an expression `n * x`.

② This "local function" `mult` is used in the expression of the `in` part. The `map` function is a list function defined in the Prelude, and it is described later in the list functions chapter.

③ This will print *[10,20,30,40,50,60,70,80,90,100]*.

13.1.1. Deconstruction

As another example, a pattern on the left hand side of a declaration in a `let` expression can be used to destructure the expression on the right hand side of the declaration.

For instance, the following function would extract the first two characters from a string whose length is at least 2:

```
firstTwoChars :: String -> [Char]
firstTwoChars str =
  let (a:b:_) = str                         ①
   in "First two chars: " ++ [a, ',', b]

main = do
  print $ firstTwoChars "hello world"       ②
```

① `str` needs to have at least 2 characters for this pattern to work.

② This will print *"First two chars: h,e"*.

13.2. Where Clauses

Similar to `let`, `where` can be used to declare bindings in function declarations and `case` expressions. For example,

```
summation :: Int -> Int
summation m = aux m 0
  where                                     ①
    aux n acc                               ②
      | n <= 0 = acc
      | otherwise = aux (n - 1) (n + acc)

main = do
  print $ summation 10                      ③
```

① A "local function" aux is declared in the where clause. Note that the aux function is "tail recursive".

② acc is an *accumulator*.

③ This will print *55*.

Unlike let bindings, the scope of the where bindings can extend over several guarded equations. For instance,

```
piecewise :: Float -> Float -> Float
piecewise x y
  | y > z = z                          ①
  | y < z = -z                         ②
  | otherwise = 0
  where                                ③
    z = x * x

main = do
  print $ piecewise 3 16               ④
  print $ piecewise 4 16               ⑤
  print $ piecewise 5 16               ⑥
```

① z is defined in the where clause below.

② The same z is used in a different guarded equation. Note that this cannot be done with a let expression, which only scopes over the expression which it encloses.

③ Note that where is part of the syntax of function declarations and case expressions, and they do not for separate expressions like let expressions.

④ This will print *9.0*.

⑤ This will print *0.0*.

⑥ This will print *-25.0*.

Chapter 14. Conditional Expressions

A conditional expression has the form `if e1 then e2 else e3`. It first evaluates the Boolean expression `e1`, and if its value is `True` or `False`, then it returns the value of `e2` or `e3`, respectively. Otherwise, it returns `_|_`. Note that the type of `e2` and `e3` must be the same, which is also the type of the overall `if` expression.

For example,

```
summation :: Int -> Int
summation n =
  if n <= 0                                    ①
    then 0
    else n + summation (n - 1)

main = do
  print $ summation 10                         ②
```

① An `if - then - else` expression. Notice the layout. The `then` and `else` clauses have the same indentations.

② This will print *55*.

This `summation` function is equivalent to the following definition, using the Boolean guards:

```
summation' :: Int -> Int
summation' n
  | n <= 0 = 0
  | otherwise = n + summation' (n - 1)
```

Chapter 15. Case Expressions

A `case` expression has the following general form:

```
case e of { p1 match1 ; ... ; pn matchn }
```

Each alternative `pi matchi` consists of a pattern `pi` and its match, `matchi`. Each match in turn consists of a sequence of pairs of guards `gsij` and bodies `eij` (expressions), followed by optional `where` bindings, `declsi`.

```
| gsi1 -> ei1
...
| gsimi -> eimi
  where declsi
```

When there is only one guard that always evaluates to `True`, e.g., `pat | True -> exp`, then it can be omitted for an alternative short hand form, `pat -> exp`.

A `case` expression must have at least one alternative, and all bodies must have the same principal type, which is the type of the whole `case` expression.

A `case` expression is evaluated by pattern matching the expression `e` against the individual alternatives, from top to bottom. If `e` matches the pattern of an alternative, then the guarded expressions for that alternative are tried sequentially from top to bottom. If the guard succeeds, then the corresponding body is evaluated. If all guards fail, then this guarded expression fails and the next guarded expression is tried. If none of the guarded expressions for a given alternative succeed, then matching continues with the next alternative. If no alternative succeeds, then the value of the `case` expression is `_|_`.

The conditional expression, `if e1 then e2 else e3`, for example, can be written as follows, using the `case` expression:

```
case e1 of
  True  -> e2
  False -> e3
```

The function declaration using patterns is a shorthand syntax for using a `case` expression. That is, for instance,

```
f p11 ... p1k = e1
...
f pn1 ... pnk = en
```

This function definition for `f` is equivalent to the following:

```
f x1 x2 ... xk =
  case (x1, x2, ..., xk) of                ①
    (p11, ..., p1k) -> e1                  ②
    ...
    (pn1, ..., pnk) -> en
```

① The matching expression is a tuple when `k >= 2`, consisting of the function arguments, in the given order. Otherwise it's a single value.

② The pattern on the left-hand side is a tuple pattern.

Here are a couple of examples:

```
not' :: Bool -> Bool                       ①
not' x = case x of
  True -> False                            ②
  False -> True                            ③
```

① The `not` function is defined in the Prelude, and hence we use a different name `not'`, for illustration.

② If a given argument evaluates to `True`, the `not'` function returns `False`. The value `True` in this example is called a literal pattern.

③ Otherwise, that is, when `x == False`, it returns `True`.

This `not'` function is equivalent to the following:

```
not'' :: Bool -> Bool
not'' True = False
not'' False = True
```

The above two definitions of the `not` function are semantically equivalent. Likewise, the following two definitions of the `isZero` function are equivalent to each other.

```
isZero :: Int -> Bool
isZero :: Int -> Bool
isZero x = case x of
  0 -> True                                   ①
  _ -> False                                  ②
```

① If the value of `x` is `0`, then the `isZero` functions returns `True`.

② Otherwise, it returns `False`. The underscore `_` is a wildcard pattern, and it matches any `Int` value in this example.

```
isZero' :: Int -> Bool
isZero' 0 = True
isZero' _ = False
```

Pattern matching is described in more detail in the next chapter.

Chapter 16. Patterns

The `case` expressions are used with patterns, as described in the previous chapter. Patterns can also appear in lambda abstractions, function definitions, pattern bindings, list comprehensions, and `do` expressions, which are all ultimately translated into `case` expressions.

16.1. Pattern Matching

Patterns are matched against values. Attempting to match a pattern can result in one of the following three results:

- It may succeed, returning a binding for each variable in the pattern,
- It may fail, or
- It may diverge (i.e. return `_|_`).

Pattern matching proceeds from left to right, and outside to inside. We describe each of the valid patterns in Haskell in the following sections.

16.2. Wildcard Patterns

The wildcard pattern `_` is an irrefutable pattern, and it matches any value. It is similar to a variable pattern, but there is no binding. Hence, the `_` patterns are useful when some part of a pattern is not referenced on the right-hand-side. For example,

```
wildcardPatterns :: String -> Char
wildcardPatterns x =
  case x of
    "" -> ' '
    _ -> '!'                               ①
```

① The wildcard pattern `_` matches any non-null string in this example.

16.3. Literal Patterns

A numeric, `Char`, or `String` literal pattern `p` matches against a value `v` if `v == p`. In case of numeric literals,

- An integer literal pattern can only be matched against a value in the class `Num`, and
- A floating literal pattern can only be matched against a value in the class `Fractional`.

For example,

```
literalPatterns :: Int -> Int
literalPatterns x =
  case x of
    33 -> 30                                   ①
    -44 -> -50                                 ②
    _ -> 0
```

① `x = 33` matches this literal pattern. The value of the `case` expression is `30`.

② `x = -44` matches this negative number literal pattern. The `case` expression returns `-50` in this case.

16.4. Constructor Patterns

Haskell supports a few different forms of constructor patterns. The "record pattern" is described in the next section. A constructor pattern is a nested pattern, and the arity of a constructor must match the number of sub-patterns associated with it.

The pattern `F {}` matches any value built with constructor `F`, whether or not `F` was declared with record syntax.

When the constructor is defined by `data`, matching the pattern `con pat1 … patn` depends on the value:

- If the value is of the form `con v1 … vn`, sub-patterns are matched from left to right against the components of the data value.
  - If all matches succeed, the overall match succeeds.
  - Otherwise, the first to fail or diverge causes the overall match to fail or diverge, respectively.
- If the value is of the form `con' v1 … vm` with `con` and `con'` two different constructors, then the match fails.
- If the value is `_|_`, then the match diverges.

For example,

```
data Boring = Empty | Vacant

nullaryPatterns :: Boring -> Bool
nullaryPatterns x =
  case x of
    Empty -> True                        ①
    _ -> False                           ②

main = do
  print $ nullaryPatterns Empty          ③
  print $ nullaryPatterns Vacant         ④
```

① A nullary constructor pattern.

② The wildcard pattern. In this particular example, it only matches the other nullary constructor `Vacant` of the `Boring` type. Hence, `_ -> False` is equivalent to `Vacant -> False`.

③ This will print `True` to the terminal.

④ This will print `False`.

```
consPatterns :: Either Int String -> Int
consPatterns x =
  case x of
    Left 1 -> 100                        ①
    Right "Five" -> 500                  ②
    _ -> 0

main = do
  print $ consPatterns $ Left 1          ③
  print $ consPatterns $ Right "Ten"     ④
```

① A constructor pattern. The `Either` type is defined with two data constructors, `Left` and `Right`.

② Another constructor pattern.

③ This will print `100`.

④ The argument `Right "Ten"` matches neither constructor pattern in this example, and hence it matches the wildcard pattern and the function returns `0`.

When the constructor is defined by `newtype`, the pattern `con pat` matches against a value as follows:

- If the value is of the form `con v`, then `pat` is matched against `v`.
- If the value is `_|_`, then `pat` is matched against `_|_`.

For example,

```
newtype Truth = Truth Bool               ①

newtypePatterns :: Truth -> Int
newtypePatterns x =
  case x of
    Truth True -> 1000000                ②
    _ -> 0
```

```
main = do
  print $ newtypePatterns $ Truth True    ③
  print $ newtypePatterns $ Truth False   ④
```

① A newtype `Truth` is created with `Bool`. Note that the `Bool` type has two nullary constructors, `True` and `False`.

② A constructor pattern.

③ This will print `1000000`.

④ This will print `0`.

Binary data constructors can also use the infix syntax. For instance,

```
data Sum = Sum Int Int                    ①

infixPatterns :: Sum -> Int
infixPatterns x =
  case x of
    1 `Sum` 2 -> 3                        ②
    _ -> 0

main = do
  print $ infixPatterns (Sum 1 2)         ③
  print $ infixPatterns $ 1 `Sum` 2       ④
  print $ infixPatterns (Sum 2 2)         ⑤
```

① The type `Sum` has a single data constructor `Sum`, which takes two `Int` arguments.

② An infix constructor pattern. This pattern ``1 `Sum` 2`` is equivalent to the normal constructor pattern `Sum 1 2`.

③ This will print `3`.

④ Same as above. This will output `3` to the terminal.

⑤ This will print `0`.

16.5. Labeled Patterns

In the ordinary constructor patterns, pattern matching occurs based on the position of arguments in the value being matched. When matching against a constructor using labeled fields, the fields are matched based on their names, and in the order they are listed in the pattern. Otherwise, these two constructor patterns work more or less the same way. Fields not named by the pattern are ignored. That is, they are matched against `_`.

```
data Color =
  Color { red, gray, blue :: Int }        ①

labeledPatterns :: Color -> String
labeledPatterns x =
  case x of
    Color {red = 0} -> "Not so red"        ②
    Color {blue = 255} -> "Full of blue"  ③
    _ -> ""

main = do
  print $ labeledPatterns
   Color {red = 0, gray = 0, blue = 255} ④
  print $ labeledPatterns
   Color {red = 1, gray = 0, blue = 255} ⑤
  print $ labeledPatterns
   Color {red = 1, gray = 1, blue = 254} ⑥
```

① A constructor with labeled fields. The record syntax is explained later in the book.

② A labeled field constructor pattern. This pattern matches as long as the value of the field `"red"` is `0` regardless of values of other fields.

③ Another labeled pattern. This pattern matches as long as the value `"blue"` is `255`.

④ Since patterns are tested from top to bottom, this will match the first pattern `Color {red = 0}` in the `case` expression.

⑤ This will match the second labeled pattern, `Color {blue = 255}`.

⑥ This will match the wildcard pattern, which is an irrefutable pattern.

16.6. Variable Patterns

Pattern matching also allows values to be assigned to variables. For example, matching a pattern `var` against a value `v` always succeeds and binds `var` to `v`. This is called the variable pattern. It is similar to the wildcard pattern in that both are irrefutable patterns, that is, they will match any value.

For example,

```
variablePatterns :: Char -> String
variablePatterns x =
  case x of
    '\0' -> "None found"            ①
    c -> "Found: " ++ [c]           ②

main = do
  print $ variablePatterns 'a'      ③
  print $ variablePatterns 'z'      ④
```

① A character literal pattern.

② A variable pattern. This pattern will match any `x` other than the null character, `\0`, in this example.

③ This will print `"Found: a"`.

④ This will print `"Found: z"`.

16.7. As-Patterns

Patterns of the form `var@apat` are called as-patterns, and allow one to use `var` as a name for the value being matched by `apat`. That is, matching an as-pattern `var@apat` against a value `v` is the result of matching `apat` against `v` and, if the match is successful, binding `var` to `v`. If the match of `apat` against `v` fails or diverges, then so does the overall match of the as-pattern. For example,

```
asPatterns :: String -> (Char, Int)
asPatterns x =
  case x of
    "" -> ('\0', 0)
    w@(c:_) -> (c, length w)          ①

main = do
  print $ asPatterns "Hello, world"      ②
  print $ asPatterns "Bonjour le monde"  ③
```

① An as-pattern. The pattern `(c:_)` matches a string with at least one character, in this example, and it binds a variable `c` to the first character of the matched string. The string itself is bound to a variable `w` through this as-pattern.

② This will print *('H',12)*. `c` and `w` are bound to `'H'` and `"Hello, world"`, respectively.

③ This will print *('B',16)*.

16.8. Tuple Patterns

A tuple pattern provides a convenient syntax over what is essentially a constructor patten. Wildcard patterns are often used to ignore certain elements in pattern matching. As nested patterns, other (sub-)patterns are also commonly used in the element positions of the tuple patterns. For example,

```
tuplePatterns :: (Int, Char, Bool) -> Int
tuplePatterns x =
  case x of
    (1, _, _) -> 1
    (_, c, True) -> fromEnum c
    _ -> 0

main = do
  print $ tuplePatterns (1, 'a', False)  ①
  print $ tuplePatterns (2, 'A', True)   ②
  print $ tuplePatterns (3, 'a', False)  ③
```

① The value `(1, 'a', False)` will match the first pattern, and this expression will print *1* to the terminal through IO action.

② This will print *65*. The ASCII code of the English uppercase letter `'A'` happens to be 65. `fromEnum` is a method of the `Enum` class.

③ This will print *0*.

16.9. List Patterns

Haskell also provides some convenient pattern syntax for matching lists, which essentially amounts to some variations of the constructor patterns, similar to how the tuple patterns work.

In particular, you can match with the nil constructor, or an empty list, `[]`, or you can match with the cons `:` constructor, `(x:xs)`, where `x` represents a single element, or the "head", and `xs` refers to the rest of the list, or the "tail" list, which can be empty. The patterns like `(x:xs)` or `(_:xs)`, etc. can only match lists with at least one element. (Note that parentheses `()` are not part of the list patterns.) Alternatively, one can also use the complete cons pattern by repeatedly applying the cons operator on each of the elements in a list, e.g., `(x:y:z:[])`, or its syntactically sugared version, `[x,y,z]`, which will match a list with three elements.

Or, one can even use syntax somewhere between the two. For example. a pattern (x:y:zs) will match a list with at least two elements, with zs matching a list with zero or more elements after removing the first two elements in a value. For example,

```
listPatterns :: [Int] -> String
listPatterns x =
  case x of
    [] -> "Empty"                               ①
    [c] -> "Uno: " ++ [toEnum c :: Char]        ②
    [c, d] -> "Dos: " ++ show (c, d)            ③
    (c:_) -> "Mas: " ++ show c ++ " etc"        ④

main = do
  print $ listPatterns []                       ⑤
  print $ listPatterns [100]                    ⑥
  print $ listPatterns [200, 250]               ⑦
  print $ listPatterns [40, 50, 60, 70]         ⑧
```

① The empty list pattern [] matches an empty list.

② The list pattern [c] will match any single element list. Note that the sub-pattern c included in this list pattern is a variable pattern, which is irrefutable.

③ The list pattern [c, d] will match any two-element list.

④ The pattern (c:_) will match any list with at least 1 element. In this example, however, it will match a list with 3 or more elements since the previous patterns match all lists with fewer than 3 elements.

⑤ This will print *"Empty"*.

⑥ This will print *"Uno: d"*. The ASCII code of 'd' happens to be 100. toEnum is also a method of the Enum class.

⑦ This will print *"Dos: (200,250)"*.

⑧ This will print *"Mas: 40 etc"*.

Here are a few more examples of pattern matching in declaring some commonly used functions in Haskell programming, As stated, lists are one of the most important data structures in Haskell programming, and likewise, the list patterns are one of the most widely used patterns. Note that all three functions are defined recursively.

```
elem' :: (Eq a) => a -> [a] -> Bool      ①
elem' _ [] = False
elem' e (x:xs) = (e == x) || elem' e xs  ②
```

① The `elem'` function takes two values of type `a` and a list type `[a]`, and it returns `True` if the first value is an element of the second value/list. Otherwise, it returns `False`. Note that the context specifies that `a` must be an instance of the `Eq` Class.

② Although we mostly use `case` expressions to demonstrate various patterns in this chapter, Haskell allows a special syntax for function declaration with pattern matching, as indicated earlier. This kind of function pattern binding syntax is more widely used, especially for simple functions. This particular function declaration is, for instance, equivalent to the following:

```
elem'' :: (Eq a) => a -> [a] -> Bool
elem'' ex list =
  case (ex, list) of
    (_, []) -> False
    (e, x:xs) -> (e == x) || elem'' e xs ①
```

① Note that, in the first example, the list pattern `x:xs` is enclosed in parentheses. This is because function application has a higher precedence than the cons constructor `:` in the pattern. In general, list patterns are often combined with parenthesis patterns because the cons operator has a generally rather low fixity. In this particular example, however, the parentheses are not needed since it is an element of a tuple pattern.

The following function, `dedupe`, uses the `elem` function (e.g., from the Prelude), and removes all duplicates in a given list.

```
dedupe :: (Eq a) => [a] -> [a]          ①
dedupe [] = []                          ②
dedupe (x:xs)                           ③
  | x `elem` xs = dedupe xs
  | otherwise = x : dedupe xs
```

① The `elem` function requires the type `a` to be an instance of the `Eq` class. Hence, `dedupe` has the same requirement.

② We handle an empty list here.

③ Then, we can assume that all lists have at least one element at this point. This pattern includes two guards. The implementation is straightforward.

The following function, `isAsc`, takes a list of elements of an `Ord` type and it returns `True` if all elements in the given list is sorted in the ascending order. Otherwise, it returns `False`.

```
isAsc :: (Ord a) => [a] -> Bool
isAsc [] = True                         ①
isAsc [_] = True                        ②
isAsc (x:y:xs) =                        ③
  (x <= y) && isAsc (y : xs)            ④
```

① When a list includes no elements, should it be considered sorted?

② What about a list with one element?

③ At this point, we can assume that the list we are matching has at least two elements, and hence `x:y:xs` is a valid pattern. (Note that `xs` can still be an empty list.)

④ The implementation is straightforward.

16.10. Parenthesized Patterns

Pattern matching can extend to nested values, e.g., as we have seen some examples so far, and as we will discuss further at the end of this section. Parenthesized patterns are used for grouping purposes. For example,

```
data Me = Me (Maybe Int) (Maybe String)  ①

parenPatterns :: Me -> Int
parenPatterns x =
  case x of
    Me (Just n) (Just s) -> n + length s ②
    Me (Just n) _ -> n                   ③
    _ -> 0

main = do
  print $ parenPatterns $
    Me (Just 1) (Just "hi")              ④
  print $ parenPatterns $
    Me Nothing (Just "hi")
  print $ parenPatterns $
    Me (Just 1) Nothing
  print $ parenPatterns $
    Me Nothing Nothing
```

① A data type with one constructor, which consists of two fields.

② Constructor patterns can be nested. In this case, the overall pattern is a constructor pattern. Both of its arguments are parenthesized patterns, each of which contains a constructor pattern.

③ Similarly, a constructor pattern with two sub-patterns, a parenthesized pattern over another constructor pattern and the wildcard pattern.

④ These four print expressions will output *3*, *0*, *1*, and *0* to the terminal.

16.11. Nested Patterns

Patterns can be nested. In particular, constructor patterns, list patterns, and tuple patterns, along with parenthesized patterns, can include other sub-patterns, some of which can in turn include other sub-patterns, and so on. Here are some more examples of nested patterns.

```
addTuples :: (Num a, Num b) =>
  (a, b) -> (a, b) -> (a, b)               ①
addTuples (x1, y1) (x2, y2) =              ②
  (x1 + x2, y1 + y2)
```

① The `addTuples` function take two pairs and return their sum.

② As indicated, this pattern is the same as a tuple of two tuples, with each tuple containing two variable patterns.

```
main = do
  print $ addTuples (1.0, 3) (2.0, 0)     ①
```

① This will print *(3.0,3)*.

```
data Point a b = Origin | Point a b        ①
  deriving(Show)

addPoints :: (Num a, Num b) =>
  Point a b -> Point a b -> Point a b
addPoints Origin Origin = Origin           ②
addPoints Origin (Point x2 y2) =
  Point x2 y2
addPoints (Point x1 y1) Origin =
  Point x1 y1
addPoints (Point x1 y1) (Point x2 y2) =
  Point (x1 + x2) (y1 + y2)
```

① A datatype with two constructors.

② All patterns in this function binding are tuples of two constructors, one of which comprises two variable patterns.

```
main = do
  print $ addPoints Origin Origin          ①
  print $ addPoints Origin (Point 3 4.0) ②
```

① This will print *Origin*.

② This will print *Point 3 4.0*.

```
addLists :: (Num a, Num b) =>
  [(a, b)] -> [(a, b)] -> [(a, b)]         ①
addLists [][] = []
addLists [] ((x2, y2):ws) =                ②
  (x2, y2):ws
addLists ((x1, y1):zs) [] =
  (x1, y1):zs
addLists ((x1, y1):zs) ((x2, y2):ws) =
  (x1 + x2, y1 + y2) : addLists zs ws
```

① This `addLists` function takes two lists of pairs and returns a list of pairs by adding their corresponding elements.

② All four of these patterns are implicitly top-level tuple patterns (when converted to a `case` expression). In this particular case, the second element pattern is a list pattern enclosed in parentheses. The inner parentheses are part of the tuple pattern.

```
main = do
  print $ addLists [(1, 2)] [(2, 4), (6, 8)]   ①
```

① This will print *[(3,6),(6,8)]* to the terminal.

16.12. Irrefutable Patterns

The following patterns are irrefutable:

- A variable pattern,
- A wildcard pattern,
- A lazy pattern, in the form of `~apat`, where `apat` ia another pattern, which is described further at the end of the section,
- An as-pattern of the form `var@apat` where `apat` is irrefutable, and
- `N apat` where `N` is a constructor defined by `newtype` and `apat` is irrefutable.

All other patterns are refutable. Matching an irrefutable pattern is non-strict. That is, the pattern matches even if the value to be matched is `_|_`. Matching a refutable pattern is, on the other hand, strict. That is, if the value to be matched is `_|_`, then the match diverges.

16.13. Lazy Patterns

A lazy pattern has the form `~apat`, where `apat` ia another pattern, which may or may not be irrefutable.

Matching the pattern `~apat` against a value `v` always succeeds. But, no actual matching evaluation is done on a `~apat` pattern until one of the variables in `apat` is used. At that point the entire pattern is matched against the value, and the free variables in `apat` are bound to the appropriate values if matching `apat` against `v` would otherwise succeed. If the match fails or diverges, so does the overall computation.

Chapter 17. Core Functions

The Haskell Standard Prelude includes a number of "builtin" functions.

17.1. The `id` Function

```
id :: a -> a
```

The builtin identity function `id` for a given value `x` returns the same value `x`.

```
main = do
  print $ id "Hello, Haskell!"          ①
```

① This will print *"Hello, Haskell!"*.

17.2. The `const` Function

```
const :: a -> b -> a
```

The builtin constant function `const` takes two arguments, and it returns the value of the first argument, ignoring the second argument.

```
main = do
  print $ const 'a' 'b'                 ①
  print $ const 42 "Irrelevant"         ②
```

① This will print *'a'*.

② This will print *42*.

17.3. The flip Function

```
flip :: (a -> b -> c) -> b -> a -> c
```

The builtin `flip` function takes a function of two arguments as an argument, and it return another function which works like the given function, but taking the two arguments in the reverse order. That is, `flip f x y = f y x`.

```
fnPower :: Int -> Int -> Int
fnPower a b = a ^ b
fnPowerFlipped :: Int -> Int -> Int
fnPowerFlipped = flip fnPower

main = do
  print $ fnPower 2 3                ①
  print $ fnPowerFlipped 2 3         ②
```

① This will print *8*.

② This will print *9*.

17.4. The seq Function

```
seq :: a -> b -> b
```

The builtin `seq` function takes two arguments, and it makes both arguments to be evaluated. Its return value is the value of the second argument unless the first argument is `_|_`, in such a case it returns `_|_`.

```
_|_ `seq` b = _|_
a `seq` b = b
```

17.5. The Lazy Infix Application Operator ($)

```
($) :: (a -> b) -> a -> b
```

The lazy infix application operator `$` takes a function and returns the same function. That is, `($) f == f`, or `f $ x == f x`. The `$` operator is right-associative, and it is primarily used in continuation-passing style. For example, the following two print expressions are the same:

```
main = do
  print (sum (map (* 2) [1, 2, 3]))      ①
  print $ sum $ map (* 2) [1, 2, 3]      ②
```

① This will print *12*.

② The same *12*. These two expressions are semantically equivalent.

17.6. The Eager Infix Application Operator ($!)

The eager infix application operator `$!` takes a function and returns a `seq` function with the same function as its second argument. That is, `($!) f == seq _ f`, or `f $! x == x `seq` f x`. The `$!` operator is right-associative, like `$`. Using the same example above,

```
main = do
  print $! sum $! map (* 2) [1, 2, 3]    ①
```

① This will print *12*. The only difference between `$` and `$!` is their strictness. That is, `$` preserves the default laziness whereas `$!` uses the `seq` function to force eager evaluation of arguments.

17.7. The `until` Function

`until p f` yields the result of applying `f` until `p` holds.

```
until :: (a -> Bool) -> (a -> a) -> a -> a
```

For example,

```
main = do
  print $ until (> 10) (* 2) 1              ①
```

① This will print *16*.

17.8. The `asTypeOf` Function

`asTypeOf` is a type-restricted version of `const`. Its typing forces its first argument to have the same type as the second.

```
asTypeOf :: a -> a -> a
```

For example,

```
main = do
  print $ asTypeOf 3 (5 :: Int)             ①
```

① The type of the literal `3` is `Int`.

Chapter 18. List Functions

The Prelude defines the following list-related functions:

```
null, !!, length, ++, concat, reverse
head, tail, last, init
take, drop, splitAt, takeWhile, dropWhile, span, break
map, concatMap, filter, any, all
foldl, foldl1, scanl, scanl1, foldr, foldr1, scanr, scanr1
iterate, repeat, replicate, cycle
zip, zip3, zipWith, zipWith3, unzip, unzip3
lines, words, unlines, unwords
and, or, elem, notElem, lookup, maximum, minimum, sum, product
```

18.1. Basic List Functions

This section describes `null`, `!!`, `length`, `++`, `concat`, and `reverse`.

18.1.1. The `null` function

```
null :: [a] -> Bool
```

The list `null` function returns `True` if a given list is empty. Otherwise, it returns `False`. For example,

```
main = do
  print $ null ([] :: [Char])          ①
  print $ null ['a', 'b', 'c']         ②
```

① It prints *True*.

② It prints *False*.

18.1.2. The index !! operator

```
(!!) :: [a] -> Int -> a
```

The index operator `!!` takes a list and a non-negative index of type `Int` and it returns the value at the given index. When the index is outside the valid index range for the given list, it throws an error. For example,

```
main = do
  print $ ['a', 'b', 'c'] !! 1          ①
  -- print $ ['a', 'b', 'c'] !! (-1)    ②
  -- print $ ['a', 'b', 'c'] !! 4       ③
```

① This outputs *'b'*. List indexes are `0`-based.

② It raises an error. *Prelude.!!: negative index*.

③ It raises an error. *Prelude.!!: index too large*.

18.1.3. The `length` function

```
length :: [a] -> Int
```

The list `length` function returns the length of a given list as an `Int`. This function does not terminate when the given list is not finite.

```
main = do
  print $ length []                     ①
  print $ length ['a' .. 'z']
  -- print $ length [1 .. ]             ②
```

① It prints *0*.

② This function call does not return.

18.1.4. The append ++ operator

```
(++) :: [a] -> [a] -> [a]
```

The list append operator ++ concatenates two given lists.

```
main = do
  print $ ([] :: [Char]) ++ ['a', 'b']   ①
  print $ ['a', 'b'] ++ ['e', 'f', 'g']  ②
```

① The resulting list is the same as `['a', 'b']`.

② The resulting list is the same as `['a', 'b', 'e', 'f', 'g']`.

18.1.5. The concat function

```
concat :: [[a]] -> [a]
```

The list `concat` function takes a list of lists, and it returns the concatenation of all elements of the list.

```
main = do
  print (concat [[1],
    [5, 6, 7],
    [11]] :: [Int])                      ①
  print $ concat ["Hello",
    ", ", "Dr. Haskell",
    " and ", "Mr. Highly Functional!"]   ②
```

① This prints out *[1,5,6,7,11]*.

② Since `String` is `[Char]`, this prints out *"Hello, Dr. Haskell and Mr. Highly Functional!"*.

18.1.6. The reverse function

```
reverse :: [a] -> [a]
```

The list reverse function returns the elements of a given list in reverse order. The argument list should be finite.

```
main = do
  print (reverse [1, 2, 3] :: [Int])     ①
  -- print $ reverse [1 ..]               ②
```

① This prints out *[3,2,1]*.

② This will not terminate.

18.2. Head and Tail Functions

This section describes the head, tail, last, and init functions.

18.2.1. The head function

```
head :: [a] -> a
```

The list head function takes a non-empty list and returns the first element of the list.

```
main = do
  print $ head ['a', 'b', 'c']           ①
  -- print $ head ([] :: [Char])          ②
```

① It prints *'a'*.

② It raises an error, *Prelude.head: empty list*.

18.2.2. The `tail` function

```
tail :: [a] -> [a]
```

The list `tail` function takes a non-empty list and returns a list of the remaining elements of the given list after the first element, which can be an empty list.

```
main = do
  print $ tail ([1, 2, 3] :: [Int])        ①
  -- print $ tail ([] :: [Int])            ②
```

① It prints *[2,3]*.

② It raises an error, *Prelude.tail: empty list*.

18.2.3. The `last` function

```
last :: [a] -> a
```

The list `last` function takes a non-empty and finite list and returns the last element of the list.

```
main = do
  print $ last ['a', 'b', 'c']             ①
  -- print $ last ([] :: [Char])           ②
```

① It prints *'c'*.

② It raises an error *Prelude.last: empty list*.

18.2.4. The `init` function

```
init :: [a] -> [a]
```

The list `init` function takes a non-empty and finite list and returns a list of the remaining elements of the given list before the last element.

```
main = do
  print $ init ([1, 2, 3] :: [Int])      ①
  -- print $ init ([] :: [Int])          ②
```

① It prints *[1,2]*.

② It raises an error, *Prelude.init: empty list*.

18.3. Take and Drop Functions

This section describes the `take`, `drop`, `splitAt`, `takeWhile`, `dropWhile`, `span`, and `break` functions.

18.3.1. The `take` function

```
take :: Int -> [a] -> [a]
```

The list `take` function takes an Int `n` and a list `xs`, and it returns the prefix of `xs` of length `n`. It return `xs` itself if `n > length xs`.

```
main = do
  print (take 2 [1, 2, 3, 4] :: [Int])   ①
  print (take 5 [1, 2, 3] :: [Int])      ②
```

① It prints *[1,2]*.

② It prints *[1,2,3]*.

18.3.2. The `drop` function

```
drop :: Int -> [a] -> [a]
```

The list `drop` function takes an int `n` and a list `xs`, and it returns the suffix of `xs` after the first `n` elements. Or, it return an empty list `[]` if `n >= length xs`.

```
main = do
  print (drop 2 [1, 2, 3, 4] :: [Int])   ①
  print (drop 5 [1, 2, 3] :: [Int])      ②
```

① It prints *[3,4]*.

② It prints *[]*.

18.3.3. The `splitAt` function

```
splitAt :: Int -> [a] -> ([a],[a])
```

The `splitAt n xs` function is defined as `(take n xs, drop n xs)`.

```
main = do
  print $ splitAt 0 ([1, 2, 3] :: [Int]) ①
  print $ splitAt 2 ([1, 2, 3] :: [Int]) ②
  print $ splitAt 4 ([1, 2, 3] :: [Int]) ③
```

① It prints *([],[1,2,3])*.

② It prints *([1,2],[3])*.

③ It prints *([1,2,3],[])*.

18.3.4. The `takeWhile` function

```
takeWhile :: (a -> Bool) -> [a] -> [a]
```

The `takeWhile` function, applied to a predicate `p` and a list `xs`, returns the longest (possibly empty) prefix of `xs` of elements that satisfy `p`.

18.3.5. The `dropWhile` function

```
dropWhile :: (a -> Bool) -> [a] -> [a]
```

The list `dropWhile` function, applied to a predicate `p` and a list `xs`, returns the remaining suffix after the longest (possibly empty) prefix of `xs` of elements that satisfy `p`.

18.3.6. The `span` function

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

The `span p xs` function is equivalent to `(takeWhile p xs, dropWhile p xs)`. For example,

```
main = do
  print $ takeWhile (<= 2) [1, 2, 3, 1]  ①
  print $ dropWhile (<= 2) [1, 2, 3, 1]  ②
  print $ span (<= 2) [1, 2, 3, 1]       ③
```

① It prints *[1,2]*.

② It prints *[3,1]*.

③ It prints *([1,2],[3,1])*.

18.3.7. The `break` function

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

The `break p` function is the same as `span (not . p)`.

```
main = do
  print $ break (>= 2) [1, 2, 3, 1]        ①
```

① It prints *([1],[2,3,1])*.

18.4. Map and Filter Functions

This section describes the `map`, `concatMap`, `filter`, `any`, and `all` functions.

18.4.1. The `map` function

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

The list `map` function takes a function `f` and a list `xs`, and it returns a list obtained by applying `f` to each element of `xs`. That is, `map f [x1, x2, …, xn]` evaluates to `[f x1, f x2, …, f xn]`.

For example,

```
mapDouble :: [Int] -> [Int]
mapDouble = map (* 2)                    ①
```

① A partial application of `map` to section `(* 2)`. The `mapDouble` function takes a list of `Int` and it returns another list by doubling all elements in the given list.

```
main = do
  print $ mapDouble []                 ①
  print $ mapDouble [1, 2, 3]          ②
```

① It prints *[]*.

② It prints *[2,4,6]*.

18.4.2. The `concatMap` function

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

The list `concatMap` function is defined to be a composition of `map` and `concat` functions, e.g., `concat . map`. That is, `concatMap` first applies `map` to a function of type `a -> [b]` and a list of type `[a]`, and then it `concat`s (or, flattens) the resulting list of type `[[b]]` to get the final list of type `[b]`. For example,

```
initial :: [String] -> [Char]
initial = concatMap (take 1)           ①
```

① The `initial` function takes an argument of a list of list of `Char`, and it returns a list comprising the first `Char` of each element list.

```
main = do
  print $ initial ["John", "F", "Kennedy"]      ①
  print $ initial ["Martin", "Luther", "King"] ②
```

① It prints *"JFK"*.

② It prints *"MLK"*.

18.4.3. The `filter` function

```
filter :: (a -> Bool) -> [a] -> [a]
```

The list `filter` function takes a predicate and a list, and it returns a list including only elements that satisfy the predicate. That is, `filter p xs` is the same as `[ x | x <- xs, p x ]`, using a list comprehension. For example,

```
evenInts :: [Int] -> [Int]
evenInts = filter even

main = do
  print $ evenInts [1, 2, 12, 13, 14]    ①
```

① This prints *[2,12,14]*.

18.4.4. The `any` function

```
any :: (a -> Bool) -> [a] -> Bool
```

The list `any` function takes a predicate and a list, and it returns `True` if *any* element in the given list satisfies the predicate. It returns `False` otherwise. That is, `any p` is equivalent to `or . map p`.

For instance,

```
anyOdd :: [Int] -> Bool
anyOdd = any odd                          ①
```

① The `anyOdd xs` function is equivalent to `or (map odd xs)`.

```
main = do
  print $ anyOdd [10, 11, 12]          ①
  print $ anyOdd [12, 14, 16]          ②
```

① It prints *True*.

② It prints *False*.

18.4.5. The `all` function

```
all :: (a -> Bool) -> [a] -> Bool
```

The list `all` function takes a predicate and a list, similar to the `any` function, and it returns `True` if *all* elements in the given list satisfy the predicate. Otherwise, it returns `False`. That is, `all p` is equivalent to `and . map p`. Or, `all p xs` is equivalent to `and (map p xs)`.

For example,

```
allOdds :: [Int] -> Bool
allOdds = all odd                      ①
```

① The `allOdds xs` function is equivalent to `and (map odd xs)`.

```
main = do
  print $ allOdds [10, 11, 12]         ①
  print $ allOdds [11, 13, 15]         ②
  print $ all (> 5) [6, 8, 10, 20]     ③
```

① It prints *False*.

② It prints *True*.

③ It prints *True*.

18.5. Fold and Scan Functions

This section describes the `foldl`, `foldl1`, `scanl`, `scanl1`, `foldr`, `foldr1`, `scanr`, and `scanr1` functions.

18.5.1. The `foldl` function

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

The list `foldl` function takes a binary operator, a starting value (typically, the left-identity of the operator), and a list, and it reduces the list using the binary operator, from left to right. That is, `foldl f z [x1, x2, …, xn]` is equivalent to `(…((z `f` x1) `f` x2) …) `f` xn`. For example,

```
main = do
  print $ foldl (++) ""
    ["To", "Be", "Or", "Not"]          ①
  print $ foldl (+) 0
    ([1, 2, 3, 4, 5] :: [Int])         ②
```

① The output: *"ToBeOrNot"*

② The output: *15*

18.5.2. The `foldl1` function

```
foldl1 :: (a -> a -> a) -> [a] -> a
```

The list `foldl1` function is a variant of `foldl` that has no starting value argument. It throws an error when it is applied to an empty list. For example,

```
main = do
  print $ foldl1 (++)
    ["To", "Be", "Or", "Not"]        ①
  print $ foldl1 (+)
    ([1, 2, 3, 4, 5] :: [Int])       ②
```

① The output: *"ToBeOrNot"*

② The output: *15*

18.5.3. The `scanl` function

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

The list `scanl` function is similar to `foldl`, but returns a list of successive reduced values from the left. That is, `scanl f z [x1, x2, …]` is equivalent to `[z, z `f` x1, (z `f` x1) `f` x2, …]`. Note that `foldl f z xs` is the same as `last (scanl f z xs)`.

```
main = do
  print $ scanl (++) ""
    ["To", "Be", "Or", "Not"]        ①
  print $ scanl (+) 0
    ([1, 2, 3, 4, 5] :: [Int])       ②
```

① The output: *["","To","ToBe","ToBeOr","ToBeOrNot"]*

② The output: *[0,1,3,6,10,15]*

18.5.4. The `scanl1` function

```
scanl1 :: (a -> a -> a) -> [a] -> [a]
```

The list `scanl1` function is similar to `scanl`, but again without the starting element. `scanl1 f [x1, x2, …]` is equivalent to `[x1, x1 `f` x2, …]`.

```
main = do
  print $ scanl1 (++)
    ["To", "Be", "Or", "Not"]          ①
  print $ scanl1 (+)
    ([1, 2, 3, 4, 5] :: [Int])         ②
```

① The output: *["To","ToBe","ToBeOr","ToBeOrNot"]*

② The output: *[1,3,6,10,15]*

18.5.5. The `foldr` function

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

The `foldr` function takes a binary operator, a starting value (typically the right-identity of the operator), and a list, and it reduces the list using the binary operator, from right to left. `foldr f z […, xn1, xn]` is equivalent to `(… `f` (xn1 `f` (xn `f` z))…)`.

For example,

```
main = do
  print $ foldr (++) ""
    ["To", "Be", "Or", "Not"]          ①
  print $ foldr (+) 0
    ([1, 2, 3, 4, 5] :: [Int])         ②
```

① The output: *"ToBeOrNot"*

② The output: *15*

18.5.6. The `foldr1` function

```
foldr1 :: (a -> a -> a) -> [a] -> a
```

The list `foldr1` function is a variant of `foldr` that has no starting value argument. It raises an error when it is applied to an empty list.

```
main = do
  print $ foldr1 (++)
    ["To", "Be", "Or", "Not"]          ①
  print $ foldr1 (+)
    ([1, 2, 3, 4, 5] :: [Int])         ②
```

① The output: *"ToBeOrNot"*

② The output: *15*

18.5.7. The `scanr` function

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

The list `scanr` function is similar to `foldr`, but it returns a list of successive reduced values from the right. That is, `scanr f z […, xn1, xn]` is equivalent to ``[…, xn1 `f` (z `f` xn), z `f` xn, z]``. Note that `foldr f z xs` is the same as `head (scanr f z xs)`. For example,

```
main = do
  print $ scanr (++) ""
    ["To", "Be", "Or", "Not"]          ①
  print $ scanr (+) 0
    ([1, 2, 3, 4, 5] :: [Int])         ②
```

① The output: *["ToBeOrNot","BeOrNot","OrNot","Not",""]*

② The output: *[15,14,12,9,5,0]*

18.5.8. The `scanr1` function

```
scanr1 :: (a -> a -> a) -> [a] -> [a]
```

The list `scanr1` function is similar to `scanr`, but again without the starting element. `scanr1 f […, xn2, xn1, xn]` is equivalent to `[…, xn2 `f` (xn1 `f` xn), xn1 `f` xn, xn]`.

```
main = do
  print $ scanr1 (++)
    ["To", "Be", "Or", "Not"]           ①
  print $ scanr1 (+)
    ([1, 2, 3, 4, 5] :: [Int])          ②
```

① The output: *["ToBeOrNot","BeOrNot","OrNot","Not"]*

② The output: *[15,14,12,9,5]*

This book can be rather "dense", depending on your background. It covers a lot of topics, but possibly not with enough depth. For example, the folding functions discussed in this section are very important tools in Haskell, and it will require some deliberate studies if you haven’t used this kind of functional programming style before. Although we claim that Haskell is a much simpler language, syntactically, than other widely-used programming languages, learning still takes time. The readers are encouraged to go through each of the above examples, step by step, so that you understand how "left folding" vs "right folding" work, etc.

18.6. Iterate and Repeat Functions

This section describes the `iterate`, `repeat`, `replicate`, and `cycle` functions.

18.6.1. The `iterate` function

```
iterate :: (a -> a) -> a -> [a]
```

The list `iterate` function is recursively defined as `iterate f x = x : iterate f (f x)`, which is an infinite list of repeated applications of `f` to `x`, e.g., `[x, f x, f (f x), …]`. For example,

```
main = do
  print $ take 5 $ iterate (* 2) 2        ①
```

① The output: *[2,4,8,16,32]*

18.6.2. The `repeat` function

```
repeat :: a -> [a]
```

The list `repeat` function returns an infinite list by indefinitely repeating a given argument. That is, `repeat x = xs where xs = x:xs`. For example,

```
main = do
  print $ take 5 $ repeat 21              ①
```

① The output: *[21,21,21,21,21]*

18.6.3. The `replicate` function

```
replicate :: Int -> a -> [a]
```

The list `replicate` function is defined to be `replicate n x = take n (repeat x)`. For example,

```
main = do
  print $ replicate 5 42                ①
```

① The output: *[42,42,42,42,42]*

18.6.4. The `cycle` function

```
cycle :: [a] -> [a]
```

The list `cycle` function takes a list and returns the infinite repetition of the given list. It returns an error when the list is empty. It returns the same list when the list is an infinite list. For example,

```
main = do
  print $ take 10 $ cycle [1, 2, 3]     ①
```

① The output: *[1,2,3,1,2,3,1,2,3,1]*

18.7. Zip and Unzip Functions

This section describes the `zip`, `zip3`, `zipWith`, `zipWith3`, `unzip`, and `unzip3` functions from the Prelude, which deal with lists of pairs (2-tuples) and triplets (3-tuples).

18.7.1. The `zip` function

```
zip :: [a] -> [b] -> [(a,b)]
```

The list `zip` function takes two lists and returns a list of pairs, each pair comprising the corresponding elements from two lists. If one input list is shorter than the other, then excess elements of the longer list are discarded.

```
main = do
  print $ zip [1, 2, 3] ['a', 'b']       ①
```

① The output: *[(1,'a'),(2,'b')]*

18.7.2. The `zip3` function

```
zip3 :: [a] -> [b] -> [c] -> [(a,b,c)]
```

The list `zip3` function takes three lists and returns a list of triplets, by taking one element from each list. The length of the resulting list is the same as that of the shortest input list.

```
main = do
  print $ zip3 [1, 2] ['a', 'b'] ["hi"]  ①
```

① The output: *[(1,'a',"hi")]*

18.7.3. The `zipWith` function

```
zipWith :: (a->b->c) -> [a]->[b]->[c]
```

The list `zipWith` function takes a binary function and two lists, and it returns a new list by applying the given function to the corresponding elements in the two input lists.

```
main = do
  print $ zipWith (+) [1, 2, 3] [3, 6]   ①
```

① The output: *[4,8]*

18.7.4. The `zipWith3` function

```
zipWith3 :: (a->b->c->d) -> [a]->[b]->[c]->[d]
```

The list `zipWith3` function takes a ternary function and three lists, and it returns a new list by combining the corresponding elements in the three input lists with the given function.

```
sum3 :: Int -> Int -> Int -> Int
sum3 x y z = x + y + z                   ①

main = do
  print $ zipWith3 sum3 [1, 2] [2] [3]   ②
```

① We define a simple ternary function for illustration. The most general type for this kind of function would be `sum3 :: Num a => a -> a -> a -> a`.

② The output: *[6]*

18.7.5. The `unzip` function

```
unzip :: [(a,b)] -> ([a],[b])
```

The list unzip function takes a list of pairs and returns a pair of lists.

```
main = do
  print $ unzip [(1, 2), (3, 4), (5, 6)] ①
```

① The output: *([1,3,5],[2,4,6])*

18.7.6. The unzip3 function

```
unzip3 :: [(a,b,c)] -> ([a],[b],[c])
```

The list unzip3 function takes a list of triplets and returns a triplet of three lists.

```
main = do
  print $ unzip3 [(1, 2, 3), (4, 5, 6)]  ①
```

① The output: *([1,4],[2,5],[3,6])*

18.8. Special Class Functions

Some list functions are defined over particular types or classes.

18.8.1. The Bool list functions

The and and or functions deal with Bool lists.

```
and, or :: [Bool] -> Bool
```

The and function returns the conjunction of all elements in a given Boolean list. Likewise, the or function returns the disjunction of all elements in a Boolean list. For example,

```
main = do
  print $ and []                         ①
  print $ and [True, False, False]
  print $ and $ replicate 10 True
  print $ and (False : repeat True)
  print $ and (False : repeat False)
  print $ and (True : repeat False)
  -- print $ and (True : repeat True)    ②
```

① The outputs are, from the top, *True*, *False*, *True*, *False*, *False*, and *False*. Note that `and []` returns `True`.

② This will hang.

```
main = do
  print $ or []                          ①
  print $ or [True, True, False]
  print $ or $ replicate 10 False
  print $ or (True : repeat False)
  print $ or (True : repeat True)
  print $ or (False : repeat True)
  -- print $ or (False : repeat False)   ②
```

① The outputs are, from the top, *False*, *True*, *False*, *True*, *True*, and *True*. Note that `or []` returns `False`.

② This will hang.

18.8.2. The `Eq` list functions

The `elem`, `notElem`, and `lookup` functions deal with lists whose elements belong to the `Eq` class.

```
elem, notElem  :: (Eq a) => a -> [a] -> Bool
```

The `elem` function takes a value and a list and it returns `True` if the value is an element of the given list. Otherwise, it returns `False`. The `notElem` function is a negation of `elem`. For example,

```
main = do
  print $ elem 3 [1, 2, 3, 4]          ①
  print $ elem 6 [1, 2, 3, 4]          ②
  print $ notElem 3 [1, 2, 3, 4]       ③
  print $ notElem 6 [1, 2, 3, 4]       ④
```

① The output: *True*

② The output: *False*

③ The output: *False*

④ The output: *True*

The `lookup` function

```
lookup :: (Eq a) => a -> [(a,b)] -> Maybe b
```

The `lookup` function takes a value and an *association list* (e.g., a list of pairs), and if there exists a pair in the list whose first element is the same as the given value, then it returns the second element `v` of the found pair, as `Just v`. If there are found multiple pairs with the same given value in the list, the first pair is used. If no such pair is found, then it returns `Nothing`. For example,

```
main = do
  let dict = [(1, 'a'), (2, 'b'), (5, 'e'), (2, 'v')]
  print $ lookup 1 dict                ①
  print $ lookup 4 dict                ②
  print $ lookup 2 dict                ③
```

① The output: *Just 'a'*

② The output: *Nothing*

③ The output: *Just 'b'*. Note that there are two pairs with its first element equal to `2`.

18.8.3. The `Ord` list functions

The `maximum` and `minimum` functions operate on non-empty and finite lists whose element types belong to the `Ord` class.

```
maximum, minimum :: (Ord a) => [a] -> a
```

The `maximum` and `minimum` functions return the maximum value or minimum value from a given list, respectively. For example,

```
main = do
  print $ maximum [10, -5, 40, 20]       ①
  print $ minimum [10, -5, 40, 20]       ②
```

① The output: *40*

② The output: *-5*

18.8.4. The `Num` list functions

The `sum` and `product` functions operate on lists whose element types belong to the `Num` class.

```
sum, product :: (Num a) => [a] -> a
```

The `sum` function computes the sum of a finite list of numbers. The `product` function computes the product of a finite list of numbers. For example,

```
main = do
  print $ sum [1, 2, 3, 4, 5]          ①
  print $ product [1, 2, 3, 4, 5]      ②
```

① The output: *15*

② The output: *120*

18.8.5. The string `lines` and `words` functions

The `lines`, `words`, `unlines`, and `unwords` functions deal with `String` and `[String]`.

```
lines   :: String -> [String]
unlines :: [String] -> String
```

The `lines` function splits a given string into a list of strings using newline characters as separators. The `unlines` function does the reverse. It joins a given list of strings into one string, which comprises multiple lines with terminating newlines. For example,

```
main = do
  let verse =
        "April is the cruellest month, breeding\n\
        \Lilacs out of the dead land, mixing\n\
        \Memory and desire, stirring\n\
        \Dull roots with spring rain."   ①
  print $ lines verse                    ②
```

① Note the "multiline string" literal syntax.

② The output: *["April is the cruellest month, breeding","Lilacs out of the dead land, mixing","Memory and desire, stirring","Dull roots with spring rain."]*

```
main = do
  let stanzas =
        [ "Frisch weht der Wind"
        , "Der Heimat zu"
        , "Mein Irisch Kind,"
        , "Wo weilest du?"
        ]
  print $ unlines stanzas                ①
```

① The output: *"Frisch weht der Wind\nDer Heimat zu\nMein Irisch Kind,\nWo weilest du?\n"*

The `words` and `unwords` functions

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

The `words` function splits a given string into a list of strings, similar to `lines`, but it uses white spaces as separators. The `unwords` function joins a given list of strings into one string with separating spaces. For example,

```
main = do
  let toBe = "To be or not to be."
  print $ words toBe                     ①
  let question = ["That", "is", "the", "question"]
  print $ unwords question               ②
```

① The output: *["To","be","or","not","to","be."]*

② The output: *"That is the question"*

Chapter 19. Data Types

19.1. Datatypes

We discuss a few different ways to declare new types or type synonyms in Haskell in an earlier part of the book. We describe the top-level `data` declaration syntax in some more detail in this chapter.

An algebraic datatype can be declared with the `data` keyword. It has the following general syntax:

```
data cx => T u1 ... uk =
  K1 t11 ... t1k1
  | ...
  | Kn tn1 ... tnkn
```

This declaration introduces a new data type `T` with one or more data constructors `K1`, …, `Kn` (or, just "constructors"). In this notation, `cx` denotes a context, and `u1` … `uk` represent type parameters. The type of each constructor `Ki` is (roughly) `ti1 -> … -> tiki -> (T u1 … uk)` within a proper context. For example,

```
data Num a => Result a
  = Tie
  | Win a
  | Loss a a
```

This declaration introduces a new data type `Result` with three constructors, `Tie`, `Win`, and `Loss`. The type of `Tie` is `Result a` for an implicit type variable `a`, whereas the types of `Win` and `Loss` are `(Num a) => a -> Result a` and `(Num a) => a -> a -> Result a`, respectively, for any type `a` that is an instance of the `Num` class.

The `data` declaration can optionally include a `deriving` clause, which is discussed in the next chapter, in the context of derived instances.

19.1.1. Field access

A data constructor of arity `k` creates an object with `k` components, in the specified order. These components are normally accessed positionally, e.g, using pattern matching.

For instance, using the above `Result` datatype,

```
scored :: Result Int -> Int
scored (Loss s _) = s
scored _ = error "Not a loss"
```

This `scored` function returns the first field of the `Loss` data constructor. For example,

```
main = do
  let result = Loss (2 :: Int) (1 :: Int)
  print $ scored result
```

Alternative to this positional access method, one can assign field labels to the components of a data object. This is called a "record". A labeled field of a record can be referenced by its label, independently of its position within the constructor. The record syntax is described next.

19.2. Record Syntax

A datatype declaration may optionally assign labels to the fields of a constructor, using the record syntax, `C { … }`. These field labels can be used to construct, select, and update fields. For example,

```
data Contact = Contact { name, phone :: String, address ::
Int, zipCode :: String }
```

These labels are referred to as selector or accessor functions because they are used to access the named fields. They must start with a lowercase letter or underscore (because they are functions), and they cannot have the same name as another function in scope.

This particular data declaration is more or less equivalent to the following without using field labels.

```
data Contact = Contact String String Int String
```

19.2.1. Field selection

Field labels create selector functions, which are top level bindings in a module.

A selector can extract the corresponding field from an object. More specifically, a field label `f` introduces a selector function defined as:

```
f x = case x of
  C1 p11 ... p1k -> e1
  ...
  Cn pn1 ... pnk -> en
```

where

- `C1` … `Cn` are the constructors of the given datatype that contains a field labeled with `f`,
- `pij` is `y` or `_` depending on whether `f` labels the `j`-th component of `Ci`, and

- `ei` is `y` or `undefined` depending on whether some field in `Ci` has a label of `f` or not, respectively.

For example, in the following datatype declaration,

```
data Data
  = Cons1 { f1 :: String, f2 :: Int }
  | Cons2 { f2 :: Int, f3 :: Bool }
  | Cons3 Int Int
```

The `f1`, `f2`, and `f3` labels are field selectors, (implicitly) defined as follows:

```
f1 :: Data -> String
f1 x = case x of
  Cons1 y _ -> y
```

```
f2 :: Data -> Int
f2 x = case x of
  Cons1 _ y -> y
  Cons2 y _ -> y
```

```
f3 :: Data -> Bool
f3 x = case x of
  Cons2 _ y -> y
```

Note that, as shown in this example,

- Record and non-record syntax constructors can be mixed in a single `data` declaration, and
- The same field labels can be used across multiple data constructors as long as they have the same types.

19.2.2. Record construction

A record constructor may be used to construct a value by specifying their components by name rather than by position, using the curly braces syntax. Unlike the braces used in declaration lists, however, the `{` and `}` characters must be explicitly included, and they cannot be omitted using the layout rules.

For instance, using the same `Data` type,

```
main = do
  let d1 = Cons1 {f1 = "Hell", f2 = 333} ①
  let d2 = Cons2 {f2 = 666, f3 = False}
  let d3 = Cons3 333 666
  print (d1, d2, d3)                     ②
```

① Note that the field order is not significant in the record syntax. That is, `Cons1 {f1 = "Hell", f2 = 333}` is equivalent to `Cons1 {f2 = 333, f1 = "Hell"}`.

② The `Data` type needs to be an instance of `Show` in order to be able to call `print`. See the section on `deriving`.

Note that the field selectors can be used just like any other top-level functions, as described above. Using the same example,

```
main = do
  let d1 = Cons1 {f1 = "Hello", f2 = 333}
  print $ f1 d1                          ①
  print $ f2 d1                          ②
```

① This will print *"Hell"*.

② This will print *333*.

19.2.3. Updating records

Values of a record syntax constructor of a datatype can be "non-destructively updated". That is, one can create a new value based on the field values of an exiting value belonging to the same record syntax constructor, by selectively updating only some (or, all) of the fields. For example,

```
main = do
  let d2 = Cons2 {f2 = 666, f3 = False}
  let d2' = d2 {f2 = 999}
  print d2'                                ①
```

① `d2'` has a value `{f2 = 999, f3 = False}`.

19.3. Abstract Datatypes

The visibility of a datatype's constructors (outside of the module in which the datatype is defined) is controlled by the form of the datatype's name in the export list, as we explain in the Modules chapter. This effectively allows creating abstract datatypes (ADTs) that cannot be directly constructed (outside the given module). For example, here's a simple *queue* data type, defined in a module named `Queue`:

Queue.hs

```
module Queue
  ( add
  , remove
  , empty
  ) where                                  ①

data QueueType a
  = NullQueue
  | Queue a (QueueType a)
  deriving (Show)
```

```
add :: a -> QueueType a -> QueueType a
add = Queue

remove :: QueueType a -> (Maybe a, QueueType a)
remove NullQueue = (Nothing, NullQueue)
remove (Queue v NullQueue) = (Just v, NullQueue)
remove (Queue v q) = (fst qq, Queue v (snd qq))
  where
    qq = remove q

empty :: QueueType a
empty = NullQueue
```

① Notice the conventional formatting. There is no difference between this and the module declaration written in one line.

In this example, we declare a datatype `QueueType` with two constructors, and define three functions, `add`, `remove`, and `empty`. Note that we export neither the type `QueueType` nor its constructors, `NullQueue` and `Queue`. Hence, a value of `QueueType` cannot be directly constructed outside this module. But, values of `QueueType` can still be used using the exported functions. For instance,

Main.hs

```
main = do
  let q1 = Queue.add (5 :: Int) $ Queue.add (3 :: Int)
Queue.empty
  print q1                                ①
  let (v, _) = Queue.remove q1
  print v                                 ②
```

① This will print *Queue 5 (Queue 3 NullQueue)*.

② This will print *Just 3*.

Chapter 20. Classes

The `class`, or typeclass, in Haskell is comparable to constructs like interfaces, traits, or protocols in other programming languages.

A `class` in Haskell is essentially a collection of types, just like a type is a collection of values. A `class` specifies a set of functions, or "behaviors". A type that belongs to a certain `class` needs to implement (either explicitly or implicitly) all functions of the `class`.

Alternatively, another way to look at the `class` in Haskell is from the viewpoint of "function overloading". A function can be defined with parameters from certain collection of types, and not just specific types. As long as the parameter set belongs to this "collection", they may be valid types for the given function.

Haskell accomplishes overloading through `class` and `instance` declarations.

20.1. Class Declarations

A class declaration introduces a new class and the operations on it, called the *class methods*. Here's a general syntax:

```
class cx => C u where cdecls
```

This declaration introduces a new class with name `C` and a *single* type variable `u`. The context `cx` specifies the superclasses of `C`, if any.

The `where` clause (e.g., the *where cdecls* part above), different from the `where` binding, is optional, but if provided, it can contain any of the following three declarations.

20.1.1. New class methods

The class declaration introduces new class methods, in the top-level namespace. The class methods of a class declaration are those with an explicit type signature `vi :: cxi => ti` in cdecls. E.g.,

```
class cx => C u where
  v1 :: cx1 => t1
  ...
  vn :: cxn => tn
```

For instance, we can define a class that provides "literate values" for numeric types as follows:

```
class Num a => Value a where
  value :: a -> String                    ①
```

① A class method for the example class `Value`. Note that this is syntactically more or less the same as the type signature declaration for a function binding. In fact, this introduces a function name, `value`, at the top-level scope.

20.1.2. Default class methods

The `where` clause may contain a *default class method* implementation for any of the class method `vi`. The default class method for `vi` is used if no binding is given in a particular `instance` declaration. For example,

```
class Num a => Value a where
  value :: a -> String                    ①
  value x = "High"                        ②
```

① An example class method, as above.

② A default class method for the class method `value`. Syntactically, orders are significant, but it is typical to put a default class method immediately below the corresponding class method, just like we (always) put the function binding below its type signature declaration.

20.1.3. Fixity declaration

The class declaration `where` clause may also contain a fixity declaration for any of the class methods. Since class methods declare top-level values, the fixity declaration for a class method may alternatively appear at top level, outside the class declaration.

20.2. Instance Declarations

An instance declaration which makes the type `T` to be an instance of class `C` is called a *C-T instance declaration*. For example, for a class `C` declared as `class cx => C u where { cbody }`, the general form of the corresponding instance declaration for type `T` is,

```
instance cx' => C (T u1 ... uk) where { d }
```

The type `(T u1 … uk)` must take the form of a type constructor `T` applied to simple type variables `u1`, … `uk`. When the type constructor is nullary, the parentheses may be omitted. The declarations `d` may contain bindings only for the class methods of `C`.

For instance, using the `Value` class example from the previous section,

```
instance Num => Value Int where
  -- value :: a -> String            ①
  value x = "High"                   ②
```

① The class method `value` for the `Value` class. You cannot redeclare it in an `instance` declaration, but sometimes it is useful to see its signature while implementing it in a particular instance. You can put it in a comment, as in this example, or you can use a GHC language extension.

② An example function binding for the class method, `value`.

The instance body declarations may not contain any type signatures or fixity declarations, since these have already been given in the class declaration. The GHC language extension *InstanceSigs* may be used if you want to explicitly include the method's type signature (the class method) in an instance declaration.

If no binding is given for a class method, then the class method of this instance is bound to `undefined` unless the corresponding default class method exists in the class declaration.

20.3. Deriving

As indicated earlier, `data` and `newtype` declarations can include an optional `deriving` clause. If it is included with one or more classes, then derived `instance` declarations are automatically generated for the datatype for each of the specified classes.

Derived instances can be declared for the `Eq`, `Ord`, `Enum`, `Bounded`, `Show`, and `Read` classes in the Prelude, and possibly for other classes in the standard library.

Chapter 21. Standard Classes

The following type classes are defined by the Haskell Prelude:

- Eq,
- Ord,
- Enum,
- Bounded,
- Read,
- Show,
- Functor,
- Monad, and
- Other numeric classes such as Num, Real, etc.

The Functor and Monad classes are explained later in the book, in separate chapters. The Applicative Functor, for Applicative for short, from the GHC language extension, is also widely used, but we do not include it in this book.

21.1. The Eq Class

The Eq class defines equality (==) and inequality (/=) methods:

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

- All basic datatypes except for functions and IO are instances of this class.
- Instances of Eq can be derived for any user-defined datatype whose constituents are also instances of Eq.

For example,

```
data Fruit = Apple | Orange
instance Eq Fruit where
  -- (==) :: Fruit -> Fruit -> Bool
  Apple == Apple = True                     ①
  Orange == Orange = True
  _ == _ = False
```

① Note that we provide a binding for `(==)`, but not for `(/=)`, in this example. The class `Eq` includes default class methods for both `(==)` and `(/=)`, using the negation of each other. That is, if a binding is provided for one in an instance, then we can rely on the default class method for the other.

Or, using `deriving`,

```
data Fruit = Apple | Orange
  deriving(Eq)
```

21.2. The Ord Class

The `Ord` class is used for totally ordered datatypes:

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

- All basic datatypes except for functions, `IO`, and `IOError`, are instances of this class.
- Instances of `Ord` can be derived for any user-defined datatype whose constituent types are in `Ord`.

For example,

```
data Sound = Do | Re

instance Eq Sound where                  ①
  -- (==) :: Sound -> Sound -> Bool
  Do == Do = True
  Re == Re = True
  _ == _ = False

instance Ord Sound where                 ②
  -- compare :: Sound -> Sound -> Ordering
  compare Do Do = EQ
  compare Re Re = EQ
  compare Do _ = LT
  compare _ Re = LT
  compare Re _ = GT
  compare _ Do = GT
```

① Note that, since `Eq` is a superclass of `Ord`, `Sound` needs to be an instance of `Eq` before it can be an instance of `Ord`.

② We rely on the default class methods for other methods of `Ord`.

21.3. The Enum Class

Class `Enum` defines operations on sequentially ordered types:

```
class Enum a where
  succ, pred     :: a -> a
  toEnum         :: Int -> a
  fromEnum       :: a -> Int
  enumFrom       :: a -> [a]
  enumFromThen   :: a -> a -> [a]
  enumFromTo     :: a -> a -> [a]
  enumFromThenTo :: a -> a -> a -> [a]
```

For example,

```
data Ternary = T0 | T1 | T2
  deriving (Show)
```

```
instance Enum Ternary where
  -- toEnum :: Int -> Ternary
  toEnum x = case x of
    0 -> T0
    1 -> T1
    _ -> T2

  -- fromEnum :: Ternary -> Int
  fromEnum t = case t of
    T0 -> 0
    T1 -> 1
    T2 -> 2
```

21.4. The Bounded Class

The `Bounded` class is used to name the upper limit and lower limit of the values of a type:

```
class Bounded a where
  minBound, maxBound :: a
```

- The types `Int`, `Char`, `Bool`, `()`, `Ordering`, and all tuples are instances of `Bounded`.
- The `Bounded` class may be derived for any enumeration type.
- `Bounded` may also be derived for single-constructor datatypes whose constituent types are in `Bounded`.

For example,

```
data Drink = Tall | Grande | Venti

instance Bounded Drink where
  -- minBound :: Drink
  minBound = Tall

  -- maxBound :: Drink
  maxBound = Venti
```

21.5. The Show Class

The `Show` class is used to convert values to strings:

```
type ShowS = String -> String            ①

class Show a where
  showsPrec :: Int -> a -> ShowS
  show      :: a -> String
  showList  :: [a] -> ShowS
```

① Declared in the Prelude. Note that `ShowS` is a function type, which takes a string and returns a string.

All Prelude types, except the function types and the `IO` type, are instances of Show. For example,

```
data Weather = Sunny | Rainy

instance Show Weather where
  -- show :: Weather -> String
  show Sunny = "Sunny"
  show Rainy = "Rainy"
```

Or, by `deriving`,

```
data Weather = Sunny | Rainy
  deriving(Show)
```

21.6. The Read Class

The `Read` class is used to convert values from strings:

```
type ReadS a = String -> [(a,String)]    ①

class Read a where
  readsPrec :: Int -> ReadS a
  readList  :: ReadS [a]
```

① A convenience type, defined in the Prelude.

All Prelude types, except function types and `IO`, are instances of `Read`. For example,

```
instance Read Weather where
  -- readsPrec :: Int -> ReadS Weather

  readsPrec _ r =
    if r == "Sunny"
      then [(Sunny, "")]
      else [(Rainy, "")]
```

Or, using `deriving`,

```
data Weather = Sunny | Rainy
  deriving(Read)
```

21.7. The Num Class

The Num class is defined as follows:

```
class (Eq a, Show a) => Num a  where
  (+), (-), (*)     :: a -> a -> a
  negate            :: a -> a
  abs, signum       :: a -> a
  fromInteger       :: Integer -> a
```

For example, using the following simple datatype,

```
data Binary = Zero | One
  deriving (Show, Eq)
```

We can make Binary an instance of Num:

```
instance Num Binary where
  -- abs :: Binary -> Binary
  abs a = a
  -- signum :: Binary -> Binary
  signum a = a
  -- fromInteger :: Integer -> Binary
  fromInteger n = if n <= 0 then Zero else One
  -- negate :: Binary -> Binary
  negate a = a
  -- (+) :: Binary -> Binary -> Binary
  Zero + Zero = Zero
  _ + _ = One
  -- (*) :: Binary -> Binary -> Binary
  One * One = One
  _ * _ = Zero
```

Chapter 22. Functors

A `Functor` represents a parametric type that can be mapped over. In fact, the list is an archetypical example of parametric types that support mapping. For example,

```
main = do
  let x = [1, 2, 3] :: [Int]
  let y = map (* 3) x
  print y
```

Note that, in this example, a value `[1, 2, 3]` of `[Int]` (a list of `Int`) has been mapped to another value `[3, 6, 9]` of the same type, using the `map` function (`map :: (a -> b) -> [a] -> [b]`). The `Functor` class is essentially a generalization of the types like lists. In addition to lists, `IO` and `Maybe` in the Prelude are in this class.

22.1. The `Functor` Class

The types belonging to the `Functor` typeclass need to support a mapping function, `fmap`, defined as follows:

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b          ①
```

① If this notation is not very clear to you, `f a` represents a parametrized type `f` with a type variable `a`, e.g., similar to `Maybe a`, etc. The most commonly used parametrized type in Haskell, namely, the list, has a special syntax, `[a]`. This is merely a syntactic sugar for `[] a`, which has the form `f a`. Note the similarity between the list's `map` function and `Functor`'s `fmap` function. In fact, as indicated, a list is an instance of `Functor` with `fmap` defined to be the good ol' `map` function.

In addition, instances of `Functor` should satisfy the following laws:

```
fmap id = id
fmap (f . g) = fmap f . fmap g
```

22.2. Functor Instances

22.2.1. The `Maybe` functor

Here's the standard implementation of `fmap` for `Maybe`:

```
instance Functor Maybe where
  -- fmap :: (a -> b) -> Maybe a -> Maybe b
  fmap f Nothing  = Nothing
  fmap f (Just x) = Just (f x)
```

One can easily verify that this implementation satisfies the `Functor` laws. For instance, both `fmap $ id Nothing` and `id Nothing` yield `Nothing`, and `fmap $ id $ Just x` and `id $ Just x` yield `Just x`. Hence `fmap id = id` for this `fmap` function. The second law `fmap (f . g) = fmap f . fmap g` can be likewise easily verified.

Some more examples:

```
main = do
  let m1 = Nothing :: Maybe Int
  print $ fmap (+ 42) m1                ①
  let m2 = Just 624 :: Maybe Int
  print $ fmap (+ 42) m2                ②
```

① This will print *Nothing*.

② This will print *Just 666*.

Chapter 23. Monads

The `Monad` class represents parametric types that support certain operations, in particular, binding (`>>=`) and `return` operations,

```
class Monad m where
  (>>=)  :: m a -> (a -> m b) -> m b      ①
  return :: a -> m a                      ②
```

① Again, `m a` refers to a parameterized type `m` with a type parameter `a`. A type `m`, which is an instance of `Monad`, needs to implement these methods for an arbitrary type variable `a`.

② Notice the `return` function. Haskell does not have the `return` statement which is found in virtually all imperative programming languages. The `return` class method of a Monad type `m` takes a value of type `a` and returns a value of type `m a`.

The binding operation `>>=` is a generalization of `concatMap` (or, "flat map") defined over a list parametric type,

```
concatMap :: (a -> [b]) -> [a] -> [b]    ①
```

① Again notice the similarity between `>>=` and the list's `concatMap` function (despite the flip of the two arguments).

For instance,

```
main = do
  let x = [1, 2, 3] :: [Int]
  let y = concatMap (\e -> [e, 2 * e]) x
  print y                                 ①
```

① This will output *[1,2,2,4,3,6]*.

Informally speaking, the `Monad` class is a generalization of parametric types like lists which support the "mapping and then flattening" operation. In the Prelude, in addition to lists, `Maybe` and `IO` are instances of `Monad`.

23.1. The Monad Class

The `Monad` typeclass defines the basic operations over a monad:

```
class Monad m where
  (>>=)   :: m a -> (a -> m b) -> m b     ①
  (>>)    :: m a -> m b -> m b
  return  :: a -> m a
  fail    :: String -> m a
  m >> k   =  m >>= \_ -> k               ②
  fail s   = error s
```

① These top four lines are class methods.

② The bottom two lines are default class methods. Hence, `(>>)` and `fail` need not be implemented in instance declarations.

Furthermore, instances of `Monad` should satisfy the following laws:

```
return a >>= k = k a
m >>= return = m
m >>= (\x -> k x >>= h) = (m >>= k) >>= h
```

Instances of both `Monad` and `Functor` should additionally satisfy the following law (in addition to the `Functor` laws):

```
fmap f xs = xs >>= return . f
```

23.2. Monad Instances

23.2.1. The `Maybe` monad

Here's the standard implementations of the `>>=`, `return`, and `fail` functions for `Maybe`:

```
instance Monad Maybe where
  (Just x) >>= k = k x
  Nothing  >>= k = Nothing
  return         = Just
  fail s         = Nothing
```

One can easily verify that these implementations satisfy the Monads laws. We will leave it as an exercise to the readers.

Here's an example use of the bind `>>=` operator with the `Maybe` monad:

```
main = do
  let m1 = Nothing :: Maybe Int
  print $ m1 >>= Just                ①
  let m2 = Just 666 :: Maybe Int
  print $ m2 >>= Just                ②
```

① This will print *Nothing*. Note that although `m1` is `Nothing`, `m1 >>= Just` does not fail. It merely returns `Nothing`.

② This will print *Just 666*.

Chapter 24. Do Expressions

A `do` expression provides a more conventional, more imperative programming-style, syntax in a monadic context. Syntactically, a `do` expression has the following general form:

```
do { STATEMENTS }
```

where `STATEMENTS` can be one or more of any of the following:

- An expression,
- A monadic assignment of the form, `pattern <- expression`,
- A `let` declaration (without `in`), and
- An empty statement (`;`).

The last statement in `STATEMENTS` must be an expression, which becomes the value of the overall `do` expression. Variables bound by `let` have fully polymorphic types while those defined by `<-` are lambda bound and thus they are monomorphic.

Empty statements are ignored. Otherwise, the `do` expressions are evaluated as follows:

- `do { exp }` is the same as `exp`.
- `do { exp; stmts }` is evaluated to `exp >> do { stmts }`.
- `do { pat <- exp; stmts }` is evaluated to `let ok pat = do { stmts }; ok _ = fail … in exp >>= ok`.
- `do { let decls; stmts }` is equivalent to `let decls in do { stmts }`.

We have been using `do` expressions throughout this book. We will see some more examples in the last chapter on IO.

Chapter 25. Basic Input/Output

The I/O system in Haskell is *purely functional*, and yet it has all of the expressive power found in imperative programming languages. Haskell uses a `Monad` to integrate I/O operations, or actions, into a purely functional context.

25.1. I/O Operations

The `IO` type is an instance of the `Monad` class. The two monadic binding functions are used to compose a series of I/O operations:

```
(>>)  :: IO a -> IO b          -> IO b
(>>=) :: IO a -> (a -> IO b) -> IO b
```

- The `>>` operator is used when the result of the first operation is uninteresting, for example when it is `()`.
- The `>>=` operation passes the result of the first operation as an argument to the second operation.

Furthermore, the `return` function is used to define the result of an I/O operation.

25.2. Exceptions

An I/O operation may raise an exception, a value of type `IOError`, instead of returning a result. One can use the Prelude `userError` function to create an `IOError`, which is discussed next.

The readers are encouraged to consult the official *Report* or other references if you would like to learn more on the IO Monad and exception handling. In the next and final chapter, we discuss some of the I/O functions in the Standard Prelude and how to use them.

Chapter 26. I/O Functions

The Prelude includes the following IO-related functions:

```
ioError, userError, catch
putChar, putStr, putStrLn, print
getChar, getLine, getContents, interact, readIO, readLn
readFile, writeFile, appendFile
```

26.1. Error Functions

26.1.1. The `userError` function

```
userError :: String -> IOError
```

The IO `userError` function returns an `IOError` value with a given string as an error message. For instance

```
demoError :: String -> IOError
demoError msg =
  userError $ "User Error: " ++ msg
```

26.1.2. The `ioError` function

```
ioError :: IOError -> IO a
```

The IO `ioError` function is used to raise an `IOError` in the IO monad. For example,

```
main = do
  ioError $ demoError "Urghh"
```

26.1.3. The catch function

```
catch :: IO a -> (IOError -> IO a) -> IO a
```

The IO `catch` function takes an IO action and a handler function, and if the IO action returns an `IOError` it raises the error in the IO monad.

26.2. Output Functions

26.2.1. The putChar function

```
putChar :: Char -> IO ()
```

The IO `putChar` function writes a given `Char` to the standard output device.

```
main = do
  putChar 'H'; putChar 'e'; putChar 'l'
  putChar 'l'; putChar 'o'; putChar '\n'
```

26.2.2. The putStr function

```
putStr :: String -> IO ()
```

The IO `putStr` function takes a string argument and it writes it to the standard output device.

26.2.3. The `putStrLn` function

```
putStrLn :: String -> IO ()
```

The IO `putStrLn` function works the same way as `putStr`, but it appends a newline character.

```
main = do
  putStr "Hello "
  putStrLn "Haskell!"
```

26.2.4. The `print` function

```
print :: Show a => a -> IO ()
```

The IO `print` function outputs a value of any `Show` type to the standard output device. We have been using the `print` function in various examples throughout this book.

26.3. Input Functions

26.3.1. The `getChar` function

```
getChar :: IO Char
```

The `getChar` function reads a character from the standard input device. It returns the value as `IO Char`. In the following example, we create a simple function `echoChar`, which repeatedly reads a character from the terminal and prints it back unless it is `'x'`. When `'x'` is inputted, we simply return with `()`.

```
echoChar :: IO ()
echoChar = do
  c <- getChar                              ①
  case c of
    'x' -> return ()
    _ -> do putChar c; echoChar             ②
```

① Note that the monadic assignment `<-`, in the context of the `do` expression, effectively does a safe conversion of `IO Char` to `Char` in this example. That is, the type of `c` is `Char`.

② We recursively call `echoChar` in this example.

26.3.2. The `getLine` function

```
getLine :: IO String
```

The `getLine` function reads a line of text from the standard input device and it returns the value as an `IO String` Monad. Here's an essentially the same function, `echoLine`, which "echoes" one line at a time, instead of one character at a time.

```
echoLine :: IO ()
echoLine = do
  line <- getLine                           ①
  case line of
    "exit" -> return ()
    _ -> do
      putStrLn line
      echoLine
```

① Using the similar monadic assignment, we effectively convert `IO String` to `String` in this example.

26.3.3. The `getContents` function

```
getContents :: IO String
```

The `getContents` function returns all user input as a single string.

```
main = do
  content <- getContents                    ①
  putStr content
```

① The `getContents` function continues to read the input until it encounters EOF (e.g., Ctrl+D). Note that this particular `do` expression is equivalent to the following using the monadic binding operator.

```
main = getContents >>= putStr
```

26.3.4. The `readIO` function

```
readIO :: Read a => String -> IO a
```

The `readIO` function reads and parses a string, and it returns an `IO` monad value of a `Read` type. It raises an exception when the parse fails. The `repeatNTimes` function in the next example reads two strings as an `Int` (`n`) and a list `[Int]`, replicates the list by `n` times, and returns the result as `IO [Int]`.

```
repeatNTimes :: String -> String -> IO [Int]
repeatNTimes rep list = do
  n <- readIO rep
  xs <- readIO list
  return $ concat $ replicate n xs
```

```
main = do
  list <- repeatNTimes "3" "[1, 2, 3]"
  print list                              ①
```

① This will print *[1,2,3,1,2,3,1,2,3]*.

26.3.5. The `readLn` function

```
readLn :: Read a => IO a
```

The `readLn` function combines `getLine` and `readIO`. For example,

```
main = (readLn :: IO Int) >>= print       ①
```

① This read an input as an `Int` and prints out the value if parse is successful. Otherwise, it throws an error.

26.3.6. The `interact` function

```
interact :: (String -> String) -> IO ()
```

The `interact` function takes a function of type `String -> String` as its argument. The entire input from the standard input device is passed to this function as its argument, and the resulting string is outputted on the standard output device. For example, here's another version of the `echo line` function, which converts all input characters to uppercase letters.

```
import Data.Char (toUpper)

main = interact $ map toUpper
```

26.4. File Functions

`FilePath` is declared to be a type synonym for `String` in the Prelude.

26.4.1. The `readFile` function

```
readFile :: FilePath -> IO String
```

The `readFile` function reads a file and returns the content of the file as a string. For example, using the following function in the current directory,

```
$ cat hello.txt
Hello, world
ditto
```

```
main = do
  content <- readFile "hello.txt"         ①
  print $ lines content                   ②
```

① If the named file is not found, it will throw an error.

② If successful, it will print *["Hello, world","ditto"]*.

26.4.2. The `writeFile` function

```
writeFile :: FilePath -> String -> IO ()
```

The `writeFile` function takes a file path and content string, and it writes the content to the given file. If the file does not exist, it creates a new file. If a file with with the given name exists, it overwrites. For example,

```
main = do
  let quote = "The future belongs to those who believe in the
beauty of their dreams."
  writeFile "world.txt" (quote ++ "\n")  ①
  readFile "world.txt" >>= print         ②
```

① This IO action creates a file named *world.txt* in the current directory, if it does not exist, and it writes the string `quote` to the file.

② This will print *The future belongs to those who believe in the beauty of their dreams.* to the terminal.

26.4.3. The `appendFile` function

```
appendFile :: FilePath -> String -> IO ()
```

The `appendFile` function takes a file path and a content string as two arguments, and it writes the content at the end of the given file. If the file does not exist, it creates a new file. For example,

```
main = do
  let quote2 = "The best way to predict the future is to
invent it."
  appendFile "world.txt" quote2          ①
  future2 <- readFile "world.txt"
  print $ words future2                  ②
```

① We use the same file used in the previous example. This IO action will append the given content, `quote2` after the current content.

② Output: *["The","future","belongs","to","those","who","believe","in","the","beauty","of","their","dreams.","The","best","way","to","predict","the","future","is","to","invent","it."].*

Epilog

Haskell is a beautiful language. That is, once you get to know it. It is a shame that only a tiny fraction of the whole developer community end up using, and enjoying, programming languages like Haskell.

If you are reading this, *congratulations!* You passed the most difficult part of learning Haskell. Once you become familiar with this relatively foreign syntax of Haskell, *the world is your oyster.* You will quickly find out that you can do so much more with so much less with Haskell. And, more importantly, you will enjoy programming more, with Haskell.

Programming languages are not just for utility, just like natural languages are not just for utility. We enjoy Shakespeare, for instance, although it has no practical value. In this age of super AI and machine learning, when programming, as a human labor, is becoming possibly obsolete (although not any time soon), programming can still be useful, and enjoyable, like an art.

Haskell is a "higher-level" programming language. Functional programming is about *what*, rather than *how*. In imperative programming, you, as a programmer, have to tell exactly how things are done to the computer. That is why we, not the computer, learn algorithms and what not.

In the higher level programming, in the near future, we will not have to concern ourselves with *exactly how.* We will just need to tell computers (or, AIs) *what* to do. They will then figure out how best to do it. (Hopefully.) In our view, functional programming is a stepping stone to that future. Languages like Haskell, which are more abstract and more high-level, can be the best tool for our next progress. *We will see.*

But, for now, go out and do some functional programming! ☺

A. How to Use This Book

> Tell me and I forget. Teach me and I remember. Involve me and I learn.
>
> — Benjamin Franklin

The books in this "Mini Reference" series are written for a wide audience. It means that some readers will find this particular book "too easy" and some readers will find this book "too difficult", depending on their prior experience related to programming. That’s quite all right. Different readers will get different things out of this book. At the end of the day, learning is a skill, which we all can learn to get better at. Here are some quick pointers in case you need some advice.

First of all, books like this are bound to have some errors, and some typos. We go through multiple revisions, and every time we do that there is a finite chance to introduce new errors. We know that some people have strong opinions on this, but you should get over it. Even after spending millions of dollars, a rocket launch can go wrong. All non-trivial software have some amount of bugs.

Although it’s a cliche, there are two kinds of people in this world. Some see a "glass half full". Some see a "glass half empty". *This book has a lot to offer.* As a general note, we encourage the readers to view the world as "half full" rather than to focus too much on negative things. *Despite* some (small) possible errors, and formatting issues, you will get *a lot* out of this book if you have the right attitude.

There is this book called *Algorithms to Live By*, which came out several years ago, and it became an instant best seller. There are now many similar books, copycats, published since then. The book is written for "laypeople", and illustrate how computer science concepts like specific algorithms can be useful in everyday life.

Inspired by this, we have some concrete suggestions on how to best read this book. This is *one* suggestion which you can take into account while using this book. As stated, ultimately, whatever works for you is the best way for you.

Most of the readers reading this book should be familiar with some basic algorithm concepts. When you do a graph search, there are two major ways to traverse all the nodes in a graph. One is called the "depth first search", and the other is called the "breadth first search". At the risk of oversimplifying, when you read a tutorial style book, you go through the book from beginning to end. Note that the book content is generally organized in a tree structure. There are chapters, and each chapter includes sections, and so forth. Reading a book sequentially often corresponds to the *depth first traversal*.

On the other hand, for reference-style books like this one, which are written to cover broad and wide range of topics, and which have many interdependencies among the topics, it is often best to adopt the *breadth first traversal*.

This advice should be especially useful to new-comers to the language. The core concepts of any (non-trivial) programming language are all interconnected. That's the way it is. When you read an earlier part of the book, which may depend on the concepts explained later in the book, you can either ignore the things you don't understand and move on, or you can flip through the book to go back and forth. It's up to you. One thing you don't want to do is to get stuck in one place, and be frustrated and feel resentful (toward the book).

The best way to read books like this one is through "multiple passes", again using a programming jargon. The first time, you only try to get the high-level concepts. At each iteration, you try to get more and more details. It is really up to you, and only you can tell, as to how many passes would be required to get much of what this book has to offer.

Again, *good luck!*

Index

F

G

H

I

M

N

O

P

Q

R

S

T

Z

About the Author

Harry Yoon has been programming for over three decades. He has used over 20 different programming languages in his academic and professional career. His experience spans broad areas from scientific programming and machine learning to enterprise software and Web and mobile app development.

He *occasionally* hangs out on social media:

- Instagram: @codeandtips [https://www.instagram.com/codeandtips/]
- TikTok: @codeandtips [https://tiktok.com/@codeandtips]
- Twitter: @codeandtips [https://twitter.com/codeandtips]
- YouTube: @codeandtips [https://www.youtube.com/@codeandtips]
- Reddit: r/codeandtips [https://www.reddit.com/r/codeandtips/]

About the Series

We are creating a number of books under the series title, *A Hitchhiker's Guide to the Modern Programming Languages.* We cover essential syntax of the 12 select languages in 100 pages or so, Go, C#, Python, Typescript, Rust, C++, Java, Julia, Javascript, Haskell, Scala, and Lua.

These are all very interesting and widely used languages that can teach you different ways of programming, and more importantly, different ways of thinking.

All Books in the Series

- Go Mini Reference [https://www.amazon.com/dp/B09V5QXTCC/]
- Modern C# Mini Reference [https://www.amazon.com/dp/B0B57PXLFC/]
- Python Mini Reference [https://www.amazon.com/dp/B0B2QJD6P8/]
- Typescript Mini Reference [https://www.amazon.com/dp/B0B54537JK/]
- Rust Mini Reference [https://www.amazon.com/dp/B09Y74PH2B/]
- C++20 Mini Reference [https://www.amazon.com/dp/B0B5YLXLB3/]
- Modern Java Mini Reference [https://www.amazon.com/dp/B0B75PCHW2/]
- Julia Mini Reference [https://www.amazon.com/dp/B0B6PZ2BCJ/]
- Javascript Mini Reference [https://www.amazon.com/dp/B0B75RZLRB/]
- Haskell Mini Reference [https://www.amazon.com/dp/B09X8PLG9P/]
- Scala 3 Mini Reference [https://www.amazon.com/dp/B0B95Y6584/]
- Lua Mini Reference [https://www.amazon.com/dp/B09V95T452/]

Community Support

We are building a website for programmers, from beginners to more experienced. It covers various coding-related topics from algorithms to machine learning, and from design patterns to cybersecurity, and more. You can also find some sample code in the GitLab repositories.

- www.codeandtips.com
- gitlab.com/codeandtips

Mailing List

Please join our mailing list, join@codingbookspress.com, to receive coding tips and other news from **Coding Books Press**, including free, or discounted, book promotions. If we find any significant errors in the book, then we will also send you an updated version of the book (in PDF). Advance review copies will be made available to select members on the list before new books are published so that their input can be incorporated, if feasible.

Request for Feedback

If you find any errors or typos, or if any part of the book is not very clear to you, or if you have any general suggestions or comments regarding the book, then please let us know. Although we cannot answer all the questions and emails, we will try our best to address the issues that are brought to our attention.

- feedback@codingbookspress.com

Please note that creating quality books like this takes a huge amount of time and effort, and we really appreciate the readers' feedback.