Haskell Lessons Documentation

Release 0.1

Justus Adam

May 04, 2017

CHAPTERS:

Some links to motivational Haskell source files:

- Makefile
- A dataflow compiler pass
- Parser for a format string
- The acommodating test suite
- A short script to strip directory prefixes

CHAPTER

ONE

THE TOOLS WE WILL NEED

This first lesson is all about the various tools we will use to develop Haskell code.

1.1 The compiler (GHC)

Haskell is a compiled language. As such you do not require any special tooling at runtime. However to develop and build Haskell projects you will require a compiler to generate an executable binary file from your code. Furthermore you will most likely require a library management tool, since the Haskell "base" library, which is bundeled with the compiler will most likely not be sufficient for most tasks[1].

There are several Haskell compilers out there, however very few are well maintained. As such the Glasgow Haskell Compiler, or GHC for short, has developed as the de-facto standard Haskell compiler. It is by far the most mature, stable and feature rich Haskell compiler. In this course we will use optional extensions of the Haskell language which not every compiler implements. I therefore highly recommend using the GHC as your Haskell compiler.

For more information about the various parts of the GHC see the compiler reference pages. There you will find information on compiler flags, the interactive prompt GHCi, including the debugger, profiling, and the GHC Haskell extensions. We will discuss all these topics in the future.

We will rarely interact directly with the compiler, as there are very nice build tools out there which we will make use of instead.

1.1.1 The interactive Interpreter (GHCi)

The GHC also supports the interpreted execution of code. For one this allows you to directly run a Haskell source file with the `runghc` or `runhaskell` program. Furthermore any standard installation of GHC includes a program in which you can interactively type Haskell code, inspect it and run it. The program is called GHCi (`ghci` is the executable name) which stands for "GHC interpreter". (ghci reference pages) GHCi is very simlar to programs like the python or ruby interpreter with the notable difference that the code you type is type checked, like normal Haskell programs, before it is executed. The GHCi also includes a debugger for Haskell code (similar to gdb) which we will study in a later chapter.

We first use GHCi to explore some Haskell code before we get started with source files.

Some notable ways to interact with GHCi are:

`<expr>` Simply submitting a Haskell expression evaluates the code and tries to print the result value.

`let <name> = <expr>` Binding the value of an expression to a name (we will learn about this *later*).

[1] This is one of the unfortunate things about Haskell, that the base library lacks many desirable things. Examples of good standard libraries would be those of python and go.

`:?` Probably the most important command. Displays the help menu. The help menu lists available commands (a selection of which will follow here) and what they do.

`:browse <module>` Displays the contents of the module with the entered name.

`:type <expr>` Prints the type of the expression `expr`.

`:info <name>` Displays information about the name, such as the source module, or the type if it is a function.

Some, but not all, of these commands also work in a shortened form (`:t` for `:type` for instance)

1.2 Documentation resources

The standard Haskell tool suite includes a tool called "haddock" which can be used to generate documentation for your source code from special in line comments. This documentation is available online for published packages.

1.2.1 Hackage

Hackage is the de-facto standard online Haskell package database. Anyone can make an account and start uploading their own Haskell packages. My username for instance is justus.

On hackage you may search for packages, browse the different versions available for each package, see a packages dependencies and also browse the generated haddock documentation.

1.2.2 Hoogle

Hoogle is a search engine for the hackage documentation. Whereas on hackage you may only search the database by package name on hoogle you can search the contents more directly by searching for function names, module names, package names and even *type signatures*.

1.2.3 Stackage

The *stackage* site, which hosts resources to be used with the tool `stack` functions similarly to a combination of Hackage and Hoogle. It hosts the documentation, including a Hoogle search, for each package snapshot. I therefore recommend to use stackage to browse haskell packages and documentation, unless the package you want information on is not on Stackage.

1.3 Editors

Any text editor is fine for Haskell development. Though it is desirable to have at least some Haskell source code highlighting. Many editors also offer extra features via one of the Haskell ide servers `ghc-mod` and `intero`.

I personally use *visual studio code*, becuase it is clean and fast and becuase I maintain its Haskell highlighting plugin and constantly improve it. However I have heard that the editor best supporting Haskell is supposedly emacs. Atom also has good Haskell support because of the atom-haskell group on github.

For those who wish to go hard on Haskell, there is a graphical editor written in Haskell itself, called leksah as well as a command line editor called yi. Also a special mention is to be given to Haskell for Mac a particularly beautiful graphical Haskell IDE with native stack support for OSX and tailored towards learning Haskell.

And lastly I want to mention ghcid. Its a very simple, command line based program which simply attempts to load your code into the interpreter and shows you the errors it encouters. It automatically refreshes whenever you save a

source file. This gives you some very bare bones ide features. The big advantage is that, unlike the other ide programs, `ghcid` is incredibly reliable.

Also for those who like their code to be a bit prettier I recommend using *font ligatures* with a font that supports it, for instance my favourite is github.com/tonsky/FiraCode. In this font there are some multi character symbols which in my opinion make the code a bit more readable.

CHAPTER

TWO

FUNDAMENTALS OF THE HASKELL SYNTAX

This lesson gets us started with the basics of Haskell syntax. Haskell is an old language (older than Java) and also one people like to experiment with. As a result a lot of extra syntax has accumulated in over the years. Some of it in regular use, some of it more obscure and not well known. Most of this extra syntax is hidden behind language extensions. We may come to learn some of it in future lessons, however for now we will simply start with the ML style core of the Haskell syntax.

2.1 Comments

There are two types of comments in Haskell. **Line comments** start with a double dash `--` (must be followed by a space or word character) and extends to the end of the current line. **Block comments** start with the sequence `{-` and extend until the end sequence `-}`.

2.2 Type literals

In Haskell type literals always start with an uppercase letter. Examples from Haskell's base library are:

`Int` A fixed size integer

`Integer` An unbounded integer.

`Float` A floating point number.

`Char` A character.

`String` A string of characters.

`Bool` A boolean.

After that the allowed characters are word characters, digits, the underscore `_` and the apostrophe `'` (often called *"prime"*).[5] Therefore a name such as `Isn't_4_bool` is a valid type name.

In general Haskell is a type inferred language, meaning you rarley have to specify the type of a value or expression (although it is common practice to annotate top level types and values with type signatures). You can however annotate any value and expression you want with a type signature. The special operator `::` can be used to achieve this (see also next section for examples). This is particularly useful when chasing down the source of type errors as you can fix expressions to a certain type you expect them to be.

[5] GHC also allows you to define data constructors and types as operators.

2.3 Value literals

Supported literals are:

Numbers `1`, `3.0` etc. These are however overloaded, meaning depending on the inferred type a literal `3` can be an `Int` or `Integer` for instance. If you wish to specify the type you can annotate the literal like so `(3 :: Int)`, `(3 :: Integer)` or `(3.5 :: Float)`, `(3.5 :: Double)`.

Characters Character literals are constructed by surrounding a character or escape sequence with single quotes.

`'a'`, `'H'`, `'5'`, `'.'`.

The escape character `\` is used to produce special values, such as the newline character (`'\n'`) For a literal `\` character use `'\\'`. For a literal `'` character use `'\''`.

Strings String literals are constructed by surrounding a sequence of characters or escape sequences with double quotes.

`"Hello World"`, `"Foo\nBar"`

The same escape sequences as for characters apply with addition of the escaped double quotes `\"`.[1]

Lists List literals are a sequence of comma separated values surrounded by square brackets.

`[1,2,3,4]`

All elements of a list must have the same type.

More on the list type in the next section.

Tuples Tuples are a sequence of comma separated values surrounded by parentheses.

`(5, "A string", 'c')`

Unlike lists the elements of a tuple can have different types.

More on the tuple type in the next section.

Note that there are not *special* literals for booleans in Haskell as they are just a regular *data structure*. The literals for `Bool` are `True` and `False`.

2.4 Bindings

Bindings (often also called variables) are names referring to pieces of data. It is similar to the concept of variables in other languages, however in Haskell bindings are **always** immutable. Since these ergo they are **not** "variable" (they cannot vary). This is why I prefer the name "binding" as it **binds** a value to an identifier, not a variable, as it cannot "vary".

Bindings must always start with a lowercase letter. Then, like the types, it may contain word characters, digits, the underscore and the apostrophe.[3]

There are several ways to bind a value. The first one we will learn (because it is the way to bind values in GHCi) is called `let` with the concrete syntax `let name = value`.

```
let myInt = 5
let aBool = False
let someString = "Hello World"
```

[1] There is a language extension in GHC which allows overloading of strings (much like the numeric literals), see overloaded strings.

[3] The naming convention in Haskell is camel case. Meaning in each identifier (type variable, type or binding) all words composing the name are chained directly, with each new word starting with an upper case letter, except for the first word, who's case is determined by the syntax contstraints (upper case for types, lower case for type variables and bindings).

You can use `let` in GHCi to bind a value and then print it by simply entering the name again and pressing enter.[4] In the `let` construct you may also, optionally, specify a type signature for the binding.

```
let myInt :: Int
    myInt = 5
```

Note that the second occurrence of `myInt` must be properly indented. We will explore the indentation rules in more detail later.

2.5 `if` expressions

In Haskell `if` is not a statement, but an expression, meaning that it returns a value. Therefore `if` always has a type, and also always has an `else` case, which must return a value of the same type. For instance we can assign the result of `if` to a binding.

```
let aBool = False

let anInt = if aBool then 8 else 9
```

Parentheses are not required and one may write any expression on the branches and for the condition of an `if`.

2.6 Function application

The syntax for applying functions to arguments in Haskell is the simplest imaginable. Its called *juxtaposition* or somethimes *prefix notation*. Meaning we simply write the function and follow it up by the arguments separated by whitespace. Optionally we can surround the whole construct with parentheses. This is especially useful when we need the result of a function call as an argument.

```
succ 5 == 6
takeDirectory "/etc/hosts" == "/etc"
elem (pred 6) [1..10]
not True == False
```

Haskell also supports binary operators. For instance the addition operator `(+)` and the equality operator `(==)`. Note that to apply the operator we use its bare form +, however if we mean a reference to the function we surround it *directly* with parentheses.

```
4 + 5 == 9
[1,2,3] ++ [4,5,6] == [1,2,3,4,5,6]

map (uncurry (+)) [(1,2), (4,5)]
```

Infix operators can also be used in the prefix notation by surrounding them with parentheses `(+)`. And prefix functions can also be used like infix operators by surrounding them with backticks `4 `elem` [1..10]`.

Function application *always binds stronger* to its arguments than operator application. For operators users may define a prescedence in which they are applied. Thus `(+)` for instance is applied before `(==)`.

[4] Note that in GHCi, as in many Haskell constructs you may also **rebind** a binding. This may look like you have altered the binding, however this is not the case. It creates a wholly new binding, which simply shadows the older binding in the current scope. When the scope is exited the value stored for this name remains the old value. You will also know that it is a new binding by the fact that the new binding can have a different type than the old one.

footnotes

CHAPTER

THREE

FUNCTIONS

3.1 Function literals

Function literals in Haskell are also often called **lambda functions**. The syntax is a slash `\` followed by a list of space separated paramters, follwed by an ASCII arrow `->` upon which follows the body of the function. Function bodies in Haskell are always an expression, and as such require no `return` keyword. Think of an implicit `return` at the beginning of the function body, this will help you understand better how to write these functions.

```
\ param -> param
```

Here for instance we have a function which takes one parameter as input and return it. This function is also known as `id`.

```
-- we often call an unspecified parameter 'x'
id = \x -> x
```

Haskell is a functional language. As such functions may be used just like any other value including being assigned to bindings. The type of our binding is now the function type `->`.

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

When we have a value of the function type we may apply it to an argument of the type *left* of the arrow to obtain a value of the type *right* of the arrow. Ergo `Int -> Bool` applied to `Int` gives a `Bool`. Similarly `a -> a` applied to `Int` gives an `Int` again. And `a -> a` applied to a `Bool` gives a `Bool`.

To apply a function we use the simplest syntax of all, juxtaposition. Also called *postfix notation* or "the function followed by the arguments, all space separated".

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

myBool = id True
myBool2 = (\x -> x) True
myInt = id 5

myBool == myBool2 == True && myInt == 5
```

Lets look at another example fuction:

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

The `const` function takes a first parameter `x` and a second parameter, which we ignore. The underscore `_` as a parameter or binding name is used to indicate that we ignore the value. And finally the function returns the first parameter.

Note that the type of the function is now `a -> b -> a`. We see here that the function type `->` occurs twice and this is deliberate because we may rewrite our function as follows:

```
const :: a -> (b -> a)
const = \x -> \_ -> x
```

Now we can see the analogy. We first consume the first parameter and return a function. This second function is then applied to the second parameter returning the final value. The two versions `\x _ -> x` and `\x -> \_ -> x` and their type signatures are equivalent in Haskell, hence the same type.

The practical upshot of this is that haskell makes it extremely easy to do what is often called "partially applied functions". This means supplying fewer arguments to a function than would be required to produce its final value. Technically this is not even possible in Haskell, since, as we have seen above, every Haskell function only takes one argument but may return a curried function to simulate being given a second argument. To fully grasp the possibilities that partial application offers it is instrumental to internalise this aforementioned concept.

Partial application is best described using examples:

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

alwaysFive = const 5

alwaysFive "a string" == alwaysFive 6 == alwaysFive () == 5

plusThree = (+ 3)

plusThree 5 == 8
plusThree 10 == 13
```

Aside

This is particularly useful when combined with higher order functions.

For instance we can increment a whole list of integers using the partial application of + to `1`.

```
map (+ 1) [4,5,8] == [5,6,9]
```

Or to find the index of a particular element in a list: (partial application of ==)

```
find (== 6) [3,6,8] == Just 2
```

Note that these are advanced examples, there is no need to undestand them yet, we will cover those in detail later.

3.2 Syntactic sugar for function definitions

There are a few common patterns in Haskell when defining functions. The first is for creating function values.

```
myFunction = \a b -> doSomthing

let anotherFunction = \x -> expr
```

This pattern is very common. Therefore we have some syntactic sugar in the Haskell laguage which allows us to omit both the backslash \ and the arrow -> by moving the function arguments before the equal sign.

```
myFunction a b = doSomthing

let anotherFunction x = expr
```

CHAPTER

FOUR

TYPES

4.1 Type variables

Types in Haskell may be parameterized over another type, which is not known at the time of defining the former type. This system is very similar to generics in many languages, but much more powerful as the type information is fully preserved.

The naming rules for type variables are the same as for *Bindings*.[1]

The whole type is then written as first the type name followed by a space and then followed by the parameters, also space separated. This is also called juxtaposition.

As an example for a parameterized type is the `Either a b` type. The name of the type is `Either` and it is parameterized by a type variable `a` and a type variable `b`. Note that there is no special significance to the name of the type variables themselves. It would be semantically equivalent to call the type `Either one the_other`. Only if we were to name both variables the same would we change the meaning, because `Either a a` would mean **both** types `Either` is parameterized over are the **same** type.

We have now seen the type in its generic form. By instantiating the type variables we can create a concrete form. For instance `Either Int String` or `Either Bool Char`. Note that `Either a b` does not mean that `a` and `b` **have** to be distinct, but they are allowed to. `Either Int Int` is also a perfectly valid concrete form of `Either a b`.

At compile time all of the type parameters must be known, i.e. only concrete form of types are allowed. The compiler will infer the concrete values of the type variables for you.

Note that if you wish to annotate a type which uses type variables you will have to fill in the concrete types for those variables *unless* they are unused. An example:

As you can see from the definition of `Either` each type variable is used in one of the constructors. If you now create one of theses values and wish to annotate it with a type you have to fill in the respective type variable. However you do not have to fill in the second variable. For instance if you create a `Left` value, lets say containing a `String` it does not matter what type `b` is in the resulting `Either`, because the `Left` constructor only uses the `a` variable and therefore the compiler will allow you to write anything for `a` including a type variable (which means it can be anything). If however you have an expression like the `if` which may either return `Left` or `Right` you have to fill in both types properly.

```
data Either a b = Left a | Right b

x :: Either String b
x = Left "A String"
y :: Either a Int
```

[1] The naming convention in Haskell is camel case. Meaning in each identifier (type variable, type or binding) all words composing the name are chained directly, with each new word starting with an upper case letter, except for the first word, who's case is determined by the syntax contstraints (upper case for types, lower case for type variables and bindings).

```
y = Right 1

x_and_y :: Either String Int
x_and_y = if someBool then x else y
```

We could also have annotated `x` and `y` with concrete types for the respective other variable, however in that case we must make it the type the `if` expression expects it to be or we get a type error. Therefore is is usually advisable to leave the type unspecified unless necessary.

```
data Either a b = Left a | Right b

-- these definitions are ok
-- because the type lines up with the if expression

x :: Either String Int
x = Left "A String"
y :: Either String Int
y = Right 1

-- these definitions are problematic
-- they would cause a type error

x :: Either String Bool
x = Left "A String"
y :: Either (Either String String) Int
y = Right 1

x_and_y :: Either String Int
x_and_y = if someBool then x else y
```

If you don't know the type of an expression but wish to annotate it or you don't know the value of one of the type variables you can use a so called "type hole" to have the compiler figure it out for you. If you annotate an expression with _ the compiler will throw an error and tell you what it infers the type for _ to be. You can use multiple _ at the same time each of which will cause a compile error with information about the inferred type. This can be used for full type signatures or even just parts of it, including type variables. GHC generally tries to infer the most general type for you.

```
-- infer a full type signature
x :: _
x = Left "A String"

-- Infer a variable
y :: Either a _
y = Right 1
```

4.2 User defined types

Defining types in Haskell takes three forms.

4.2.1 Aliases

The `type` keyword allows us to define a new name for an existing type. This can have two different purposes:

1. **It allows us to define shorter names for long type.** For instance

```
type MakerM a = StateT (ALongStateName String Bool (HashMap Text Int))␣
↪(LoggingT IO) a
```

2. **We can abstract our API from the concrete type.** If our program uses a Map like structure for instance, but we are not sure yet that we want to stick with a concrete Map type we might write the following:

```
type MyMap key value = HashMap key value
-- or (omitting the `value` variable)
type MyMap key = HashMap key
-- or (omitting both the `value` and `key` variable)
type MyMap = HashMap
```

We can then later replace it with a different map type if we like and we do not need to change all of our type signatures.

```
type MyMap = Map
```

As you can see from these examples like in function signatures type aliases support polymorphism via type variables and the type variables support partial application like functions.

4.2.2 Algebraic datatypes

Algebraic datatypes are the "normal" user defined datatypes in Haskell. They are richer than datatypes from other languages such as Java classes or C structs in that each type can have more (or less) than one representation. Some modern languages such as Rust and Swift also support those types of data. They call them Enums.

A type is defined using the `data` keyword, followed by the name of the type, which must begin with an upper case letter (see also *here*), followed by an equal sign. This is followed by any number of `|` separated *constructor definitions*.

```
data Coordinates = LongAndLat Int Int

data File = TextFile String | Binary Bytes
```

A constructor definition takes the form of first the constructor itself, followed by any number of type arguments, which are the types of the fields in the constructor. The naming constraints for the constructor are the same as for *Types*.[#type-operators]

Constructors serve two purposes.

1. **They are used, through normal function application, to *construct* a value of their type.** You can think of any constructor (like `Coordinates`) as a function, which takes arguments according to the number and type of its fields and produces a value of its type.

```
LongAndLat :: Int -> Int -> Coordinates
```

These constructors can be used just just like any other function, which includes partial application and being arguments to higher order functions.

```
LongAndLat 8 :: Int -> Coordinates

map (LongAndLat 9) [0,9,15] == [LongAndLat 9 0, LongAndLat 9 9, LongAndLat 9␣
↪15]
```

2. They are used in a pattern match to *deconstruct* a value of their type and gain access to its fields. (See *next section*)

It is very important to know the difference between a *type(name)* and a *constructor* in Haskell. Also not that it is allowed for a type and a constructor with the same name to be in scope, as the distinction between the two can be made from the context in which they are used. Type names only ever occur in a place where a type can occur, such as in the definition of another type and type signatures whereas a *Constructor* can occur in any expression.

4.2.3 Newtypes

Newtypes are basically a stricter version of the `type` alias. To be more concrete a `newtype` is a wrapper for another type which completely hides the wrapped type.

The syntax is very similar to a `data` definition, with two important restrictions.

1. The newtype must have exactly *one* constructor.
2. The constructor must have exactly *one* field.

What is so special about the newtype is that even though it may look like a `data` definition the newtype does not exist at runtime and thus has no runtime overhead. It is typically used to impose some restrictions on the creation of a type.

Whereas aliases created with `type` may be used in just the same way that the type they alias can be used a `newtype` creates a completely new type and the functions which work on the inner type *do not* work on the new type.

In the following example for instance we force the user to go through the `createEmail` function to construct an `Email` type. If we used a `type` alias the user could simply pass a `String` to the `sendEmail` function, because it is just an alias, but types created with `newtype` are distinct from the type they wrap and thus this would cause a type error.

```
newtype Email = Email String

createEmail :: String -> Either String Email
createEmail str =
    if conformsToEmailStandard str
        then Right (Email str)
        else Left "This is not a valid email"

sendEmail :: Email -> String -> IO ()
```

4.2.4 Using type variables

To use a type variable in a type you are defining yourself there is a very simple rule. You may use as many type variables as you like. Any type variable you use on the *right* side of the equal sign *must* also occur on the *left* side. Basically on the left you declare which variables the type is abstracted over and on the right you may use it as a type for your fields.[3]

Some examples:

```
data Maybe a = Just a | Nothing

data Either a b = Left a | Right b

newtype SetWrapper a = SetWrapper (Set a)
```

[3] It is possible to declare type variables on the left and then *not* use them on the right. This is often used to tag types with other types, but this is a topic for later.
There are also a language extensions which let you use type variables which only occur on the right side, however this is a very advanced topic. For now we may simply assume that this is never necessary.

Aside

There are more ways to control type variables in Haskell using a generalised concept of algebraic datatypes.

4.3 The `case` construct

The `case` construct together with function application basically comprises everything which you can do in Haskell. The `case` construct is used to deconstruct a type and gain access to the data contained within.

This is easiest to see with a user defined type

```
data MyType = Constr1 Int

aValue = Constr1 5 :: MyType
theIntWithin =
    case aValue of
        Constr1 i -> i

theIntWithin == 5
```

Any Haskell expression is allowed in the `case <expr> of` head of the construct. The body of the case statement is a number of `matchclause -> expr` pairs.

Each match clause is a combination of constructors and bindings for values. The expression to the right of the arrow may then use the values bound by these bindings.

A very simple case match (which does absolutely nothing) would be

```
case expr of
    x -> doSomething x
```

Which is the same as `doSomething expr`. We simply bind the expression to `x`.

However this is often used to create a default clause for a case match.

```
data MyType = Constr1 Int | Constr2 String

aValue = Constr1 5 :: MyType
theIntWithin =
    case aValue of
        Constr1 i -> i
        x -> 0
```

Match clauses are always matched in sequence, from top to bottom until a matching clause is found. A clause like `x`, which does not contain a constructor will always match. Therefore it is usually found as the last clause, often serving as a kind of default clause. If the default clause does not need the value we often use _ as binding to indicate that we do not use the value.

The `case` is an immensely powerful control structure as all other control structures can be defined in terms of `case` and function application. For instance we can define an `if` using case.

```
if cond a b =
    case cond of
        True -> a
        False -> b
```

You can also pattern match on all primitive, built-in types such as `Char`, `[]`, `String`, `Int`, `Float` and so on. Anything you can write as a literal you may use in a case pattern.

```
isC char = case char of
               'c' -> True
               _ -> False

isC 'l' == False
isC 'c' == True

is4 n = case n of
            4 -> True
            _ -> False

is4 4 == True
is4 0 == False
```

4.3.1 Different ways to write a case expression

Case expressions can either be written using indentation, or semicolons and braces in the same way we can do with `let`. Thereby we can use `;` to omit newlines and `{}` to omit the indentation. The following definitions are equivalent

```
case expr of
    -- note the indent of the match clauses
    Constr1 field1 field2 -> resultExpr
    Constr2 f -> resultExpr2

case expr of
    Constr1 field1 field2 ->
        -- note the deeper indent for the result expression
        resultExpr
    Constr2 f ->
        resultExpr2

-- indent is replaced with semicolons and braces
case expr of { Constr1 field1 field2 -> resultExpr; Constr2 f -> resultExpr2 }
```

4.3.2 Case match in function definition

A very common pattern in Haskell is to have a function and then directly perform a `case` match on one or more of the arguments. There is some syntactic sugar to make this more convenient.

If you define your function with the syntax where the arguments come before the = you can directly perform a pattern match on them there. Multiple `case` options are hereby achieved by defining the function once for each option. Note that in this pattern match constructors with more than zero fields need to be parenthesized (otherwise how can the compiler distinguish between field bindings and the next argument?).

```
data MyType = Constr1 Int | Constr2 String

-- before
getTheInt :: MyType -> Int
getTheInt t =
    case t of
        Constr1 i -> i
        Consrt2 _ -> 0
```

```
-- after
getTheInt2 :: MyType -> Int
getTheInt2 (Constr1 i) = i
getTheInt2 (Constr2 _) = 0

-- or, alternatively with a "_" default case
getTheInt2 :: MyType -> Int
getTheInt2 (Constr1 i) = i
getTheInt2 _ = 0
```

You can also match on multiple arguments at the same time. (I have aligned the arguments so you can better see the different patterns, this is only for readability and not necessary.)

```
addTheInts :: MyType -> MyType -> Int
addTheInts (Constr1 i1) (Constr1 i2) = i1 + i2
addTheInts (Constr i)   _            = i
addTheInts _            (Constr i)   = i
addTheInts _            _            = 0
```

4.4 Special types

There are some notable exceptions to the type naming rule. Those are the **list type**, which is `[]` or `[a]` which means "a list containing elements of type `a`" and the **tuple type** `(a,b)` for "a 2-tuple containing a value of type `a` and a value of type `b`". There are also larger tuples `(a,b,c)`, `(a,b,c,d)` etc.[2] These tuples are simply grouped data and very common in mathematics for instance. Should you not be familiar with the mathematical notion of tuples it may help to think of it as an unnamed struct where the fields are accessed by "index". And the last special type is the **function type** `a -> b`, which reads "a function taking as input a value of type `a` and producing a value of type `b`.

Some examples for concrete instances of special types:

```
myIntBoolTriple :: (Int, Int, Bool)
myIntBoolTriple = (5, 9, False)

aWordList = ["Hello", "Foo", "bar"] :: [String] -- Note: A different way to annotate␣
↪the type

-- Note: we can also nest these types
listOfTuples :: [(Int, String)]
listOfTuples =
    [ (1, "Marco")
    , (9, "Janine")
    ]
```

4.5 Record syntax

For convenience reasons there is some extra syntax for defining data types which also automatically creates some field accessor functions.

We can write the following:

[2] The source file for tuples in GHC defined tuples with up to 62 elements. Below the last declaration is a large block of perhaps 20 more declarations which is commented out, with a note above saying "Manuel says: Including one more declaration gives a segmentation fault."

```
data MyType =
    Constructor { field1 :: Int
                , field2 :: String
                }
```

This defines the type the same way as the other `data` construct. Meaning we can pattern match as usual on the constructor.

```
theData = Constructor 9 "hello" :: MyType
theInt = case theData of
            Constructor i _ -> i

theInt == 9
```

But additionally it also defines two functions `field1` and `field2` for accessing the fields.

Aka it generates code similar to the following:

```
data MyType = Constructor Int String

field1 :: MyType -> Int
field1 (Constructor i _) = i

field2 :: MyType -> String
field2 (Constructor _ s) = s
```

Also the two accessor functions `field1` and `field2` may be used in a special *record update syntax* to create a new record from an old one with altered field contents. Additionally the record may be created with a special record creation syntax.

```
data MyType =
    Constructor { field1 :: Int
                , field2 :: String
                }

v1 = Constructor 9 "Hello" :: MyType

-- record creation syntax
v2 = Constructor { field2 = "World", field1 = 4 } :: MyType

-- update syntax
v3 = v2 { field1 = 9 }
-- updating multiple fields at once
v4 = v2 { field1 = 9, field2 "Hello" }

v1 == v4

-- old records are unchanged
v2 /= v3 /= v4
```

And finally it also enables a special record pattern match using the fields.

```
theData = Constructor 9 "hello" :: MyType

theInt = case theData of
            Constructor{ field1 = i } -> i

theInt == 9
```

footnotes

CHAPTER

FIVE

MODULES

Haskell source code is structured into units which we call modules. When you type `import Data.List` you import the `Data.List` module and bring the definitions contained in that module into scope.

There belongs a source file to each module. The name of the source file is the name of the module and ends with `.hs`. For instance the `Prelude` module would be in a file called `Prelude.hs`.

Hierarchical modules, such as `Data.List` have a path matching the module name. For instance `Data.List` would be `List.hs` in a directory called `Data`.

A source file which is to be a Haskell module always starts with a module header. The module header is the keyword `module` followed by the name of the module, and optional export list and the keyword `where`.

```
module Data.Bool (Bool(True, False), bool, not) where
```

This header proceeds any other code in the source file.

5.1 Exporting

The export list, which follows the module name is a comma separated list of items to export. These items may be types, cosntructors, functions, type aliases, type classes and record fields/accessors. Only items which are exported in this list will be available when the module is imported.

Therefore the export list may be used to hide certain implementation detail from the importer. If the export list is omitted all top level declarations will be exported.

For types we can export just the type alone `Bool` or we export any number of constructors `Bool(False)`, `Bool(True, False)`.

5.2 Importing

To use the exported items from a module one must *import* them. The keyword for this is `import`. The import definitions follow the module header.

Similarly to the module definition an import declaration can have an optional import list. Only items in the import list will be in scope.

```
module MyModule where

import Data.Bool (not, (||))
```

If the import list is omitted all exported items from the module will be imported.

CHAPTER

SIX

TYPECLASSES

In addition to the *parametric* polymorphism of type variables Haskell offers *ad-hoc* polymorphism via a concept called *type classes*. Conceptually a *type class* groups a set of types for which there exists a common behaviour.

Practically a typeclass is the same as an interface in Java or C#. It defines a set of methods which a must be implemented for a certain type.

6.1 Defininig classes

The class is defined with the keyword `class` (think `interface` in Java) followed by a name for the class.[1] Following this is a type variable[2] which is a reference for the actual type. This variable is subsequently used in the method signatures to reference the type. For instance `Ord` is used to implement ordering for values.

```
class Ord a where
```

6.1.1 Member functions

In the body of the definition follows a number of declaraions for whats called *methods*. Methods are functions which must be implemented for a type to be member of this class. Below is an excerpt of the `Ord` typeclass as an example.

```
class Ord a where

    compare :: a -> a -> Ordering
    (<=) :: a -> a -> Bool
    max :: a -> a -> a
```

6.1.2 Superclasses

Classes can have a so called *superclasses*. This essentially just defines another class to be a dependency for the declaration of an instance of this class. In short: A class can depend on another class.

```
class Eq a => Ord a where

    compare :: a -> a -> Ordering
    (<=) :: a -> a -> Bool
    max :: a -> a -> a
```

[1] The naming schema for class names is the same as for types and constructors.
[2] Using the `MultiParamTypeClasses` language extensions allows one to define type classes over multiple parameters.

6.1.3 Default implementations

Lastly methods of the class can have default implementation in which may use both other methods of the class or methods of the superclasses.

```
class Eq a => Ord a where

    compare :: a -> a -> Ordering
    compare x y = if x == y then EQ
                  else if x <= y then LT
                  else GT

    (<=) :: a -> a -> Bool
    x <= y = case compare x y of { GT -> False; _ -> True }

    max :: a -> a -> a
    max x y = if x <= y then y else x
```

6.2 Constraining types

Unlike in Java where, if we wish to use an interface, we simply declare the type to *be* the interface in Haskell we *constrain* the type to *implement* the class. Constraints precede the arguments and return type in a type signature. Constraints are always placed on *type variables*. An ascii double arrow separates the constraints and the rest of the type signature.

```
max3 :: Ord a => a -> a -> a -> a
max3 a1 a2 a3 = a1 `max` a2 `max` a3
```

The advantage of this is that we can require *multiple* classes for a single type. In this case the constraints are listed comma separated and surrounded by parentheses.

```
showMax3 :: (Show a, Ord a) => a -> a -> a -> String
showMax3 a1 a2 a3 = show (max3 a1 a2 a3)
```

6.3 Implementing classes

The Haskell model of implementing classes is similar to that of Rust and Swift. Like in those languages an instance of the class (interface) can be declared anywhere. It does not have to happen at the place where the type is defined. The only constraint is that there must not be an existing instance to the class in scope. Typically the instances of the class are either defined where the type is defined or where the class is defined. This prevents situations where two instances of the same class for the same type are in scope.

Implementations of the class are done using the `instance` keyword, otherwise they are very similar to the class declaration. The `instance` keyword is followed by the class name and then the type name for which the instance is to be declared.

```
data MyType = TheSmallest | TheMiddle | TheLargest

instance Eq MyType where

instance Ord MyType where
```

In the body of the declaration follow definitions for each of the methods of the class.

```
data MyType = TheSmallest | TheMiddle | TheLargest

instance Eq MyType where
    TheSmallest == TheSmallest = True
    TheMiddle   == TheMiddle   = True
    TheLargest  == TheLargest  = True
    _           == _           = False

instance Ord MyType where

    compare TheSmallest TheSmallest = EQ -- EQual
    compare TheLargest TheLargest = EQ
    compare TheMiddle TheMiddle = EQ
    compare TheSmallest _ = LT -- Less Then
    compare TheLargest  _ = GT -- Greater Then
    compare _ TheLargest = LT
    compare _ TheSmallest = GT

    TheSmallest <= _ = True
    _ <= TheLargest = True
    TheLargest <= _ = False
    _ <= TheSmallest = False
```

6.3.1 Deriving classes

For some classes like `Eq`, `Ord`, `Show` and `Read` you may let GHC automatically define an instance for you. This is done using the `deriving` keyword after the type definition. The exact semantics of those derived classes can be found in the ghc manual.

```
data T = A | B Int deriving (Show, Eq, Ord)
```

footnotes

CHAPTER

SEVEN

I/O AND DO NOTATION

The Haskell language is very self contained due to its pure nature. Consecutive calls to a function with the same input *has* to produce the same result. This does not allow for interactions with the stateful environment, like accessing the hard disk, network or database.

To separate these stateful actions from the pure Haskell has a type called `IO`. The `IO` type is used to tag functions and values. For instance `IO Int` means that we can obtain an `Int` from this value if we let it execute some interaction with the environment.

A very common type is `IO ()` this means the function does I/O and then returns the `()` (unit) value. This value contains no information (similar to `null`) and `IO ()` is basically equivalent to `void`. It marks a function which we only want for its *effect*, not its returned *value*.

An nice example of this is the `getLine` function. As you may imagine it reads a single line from stdin and gives us back what was entered. Its type `IO String` then means that it returns a string after doing some interactions with the environment, in this case reading from the `stdin` handle.

7.1 do ing IO

IO actions can be chained using the `do` syntax. `do` syntax is basically what every function body in an imperative language is, a series of statemens and assignments. One important thing to note is that all statements in a `do` block are executed in sequence.

```
main = do
    putStrLn "Starting work"
    writeFile "Output" "work work work"
    putStrLn "Finished work"
```

As you can see we can use `do` to execute several `IO` actions in sequence. We can also obtain the values in from inside those tagged with `IO`.

```
main = do
    l <- getLine
    putStrLn ("You entered the line: " ++ l)
```

The `binding <- ioExpr` syntax means "execute the I/O from `ioExpr` and bind the result to `binding`". Since `<-` is only for `IO` tagged values you cannot use it for pure ones. To handle pure values use the statement for of `let`: `let binding = expr` (notice no `in`).

```
action :: IO ()
actions = do
    l <- getLine
```

```
    let computed = computeStruff l
    return computed
```

The `do` syntax does however not actualy execute the `IO`. It merely combines several `IO` actions into a larger `IO` action. The value from the last statement in a `do` block is what the whole thing returns. For instance if the last statement was *putStrLn "some string"* the type of the whole block would be `IO ()` (void). If it was `getLine` the type would be `IO String`. You can also return non-I/O values from within `do` by tagging them with `IO` using the `return` funciton.

7.2 Running `IO`

To execute the action there are two ways.

1. GHCi If you type an `IO` action into ghci it will execute it for you and print the returned value.
2. The `main` function.

   When you compile and run a Haskell program or interactively run a Haskell source file the compiler will search for a `main` function of type `IO ()` and execute all the I/O inside it. This means you must tie all the I/O you want to do somehow back to the main function. This is similar to a C program for instance where the `int main()` function is the only one automatically executed and all other routines have to be called from within it.

7.3 `do` Overload

There are more container and tag types which can be used similar to `IO`. To be more precise they can be used with the `do` notation, just like `IO` can.

Examples of such structures are `[a]`, `Maybe a`, `State s a` and `Reader e a`. Like `IO` all these structures represent some kind of context for the contained value `a`.

We will explore this in more detail later.

For now it suffices that in Haskell these structures are generalized with a typeclass called `Monad`. The `Monad m` typeclass requires two capabilities: `return :: a -> m a` to wrap a value `a` into the monad `m` and bind `(>>=) :: m a -> (a -> m b) -> m b` which basically states that the computations with context (the *Monad*) can be chained.

This is all that is necessary to enable them to use the `do` notation.

There is a nice library called monad-loops which implements many of the control structures one is used to from imperative languages in terms of `Monad`.

Also of interest should be the Control.Monad module from the base library which also contains some generic interactions for monads. For now it is enough to know that functions with the `:: Monad m =>` requirement can be used with `IO`.

CHAPTER

EIGHT

EXERCISES

8.1 Tools

Install the mentioned Haskell tools.

8.1.1 Installing the compiler

- Method 1 - platform first

    1. Install the correct Haskell platform distribution for your system (for instance the package `cabal-install`).
    2. Install stack. Either from source with `cabal install stack` or by downloading it from the website.

    Advantages:

    With this approach you'll have the `ghc` and `ghci` as well as `cabal` and `haddock` commands directly available to you on the command line after installation.

    Disadvantages:

    You'll have to keep your installation of these tools up to date. If you later run `stack install haddock` to get a newer version of haddock it will shadow the previously globally installed tool.

- Method 2 - stack first

    1. Install stack from the website.
    2. Use the `stack setup` command on the command line to have stack install ghc for you.

    Advantages:

    No stray `ghc` etc. executables.

    Disadvantages:

    You have to separately install haddock with `stack install haddock` if you need haddock directly.

- Make sure the stack executable is on your `$PATH`.

    Stack can update itself (if you want to). In this case it'll always install the new version in some user-local binary directory (`~/.local/bin` on UNIX). Therefore you also have to ensure this directory is on your `$PATH`. This is also important as all other executables which you install with stack are placed in this directory (including the ones you wrote yourself).

8.1.2 Install some Haskell support for your favourite editor

You may ask for help for finding said support.

Also consider the website https://wiki.haskell.org/IDEs and https://wiki.haskell.org/Editors.

8.2 Basics

1. Open GHCi.
2. Browse the `Prelude` module.
3. Be overwhelmed with how much stuff is in there.
4. Open Hackage
5. Find the `base` library.
6. Find the `Prelude` module.
7. Browse the documentation of the prelude module on Hackage.
8. Play with the various functions from the `Prelude` module in the interpreter.

8.3 A custom boolean

We want to define a boolean type. The standard library already has a type `Bool` but we will make your own.

Note: this exercise is intended to be solved using both a Haskell source file and ghci. My recommendation is to implement the code in a file (`Something.hs`) then open `ghci` and load the file with the `:load FileName.hs` (this has autocompletion for the file name as well) command. After that the types and functions you defined in the file will be in scope and you can play around with them.

Note: In ghci bindings must be created with `let binding = expr`.

8.3.1 Defining the type

Define a new type called `Boolean` with two constructors `Yes` and `No`.[1]

8.3.2 `if`

Booleans are used for `if` expressions. Therefore we will define our own `if`. Since `if` is a keyword in Haskell you can use a name like `if_` or `if'` or something else.

Your `if` function should take three arguments.

1. A `Boolean` (your custom boolean type) as condition.
2. A value which to return when the boolean is `Yes`.
3. A value which to return when the boolean is `No`.

1. Implement the function[2]

[1] Use the `data` keyword

[2] Use a `case` construct to match on the two `Boolean` constructors.

2. Add a signature to your `if`[3]

8.3.3 Boolean operations

We also need to be able to define more complex interactions. Implement a `not'`, `or'` and `and'` operation which, as the names suggest, do boolean *not*, *and* and *or*.[4]

Finally play around some with the operations you have defined. Make sure they are indeed correct.

footnotes

8.4 Implementing a library for safe html construction

We want to build a library which we can use to programmatically build a html website in Haskell and then render it.

Note: this exercise is intended to be solved using both a Haskell source file and ghci. My recommendation is to implement the code in a file (`Something.hs`) then open `ghci` and load the file with the `:load FileName.hs` (this has autocompletion for the file name as well) command. After that the types and functions you defined in the file will be in scope and you can play around with them.

Note: In ghci bindings must be created with `let binding = expr`.

8.4.1 A base type

First we need a basic `Html` type. For now this is just going to be a wrapper around a `String` containing the actiual html.

Define the `Html` type as a wrapper around `String`.[1]

Don't expose your constructor to the user of the library[2] so that they cannot unsafely create `Html` values from `String`.

Also create a function `render` or `renderHtml` which takes a `Html` value and returns it in rendered `String`. In this case that's simply the `String` contained in the `Html` value. You'll then be able to use this function in the subsequent tasks to look at the `Html` values and verify you have implemented your manipulation functions correctly.

8.4.2 Creating html from strings

Now we need the user to be able to create `Html` values from strings, but we want that to be safe. First we will enable them to create just html text nodes. Html text nodes may not contain any of the special html characters like `&`, `<`, `>`. Write a function `mkTextNode` which takes a `String` as input and verifies that none of the above mentioned characters are in it.[3] If one of the characters is found raise an `error` and if not return a `Html` value containing the string.

[3] Use a type variable for the two values. If you're stuck think about what you know about the two values (do they perhaps have the same type? Aka the type variable needs to be the same) and what type would the return be?

[4] A nested `case` match should be useful here. Alternatively you can match on both booleans simultaneously if you wrap them in a tuple *or* if you match them with the special function syntax on the arguments.

[1] You can use a `data` declaration, however since we only have one field in it you should use a `newtype`.

[2] Use the export list in your module to only export the type, not the constructor.

[3] Remember that the Haskell `String` type is just a list of characters. Look at the `Data.List` module in the `base` library documentation and find the function that allows you to test whether a certain character is in the string. (Hint: its the same function that tests whether a certain value is an *element* of the list.)

8.4.3 Concatenating html

Html elements can also be consecutive. Like `<div>...</div><span>...</span>`.

Write a function which takes as input *two* `Html` values and returns a `Html` value which is the concatenation of the two input `Html` values.[4]

8.4.4 Html containers

Now we want to be able to use things like html `div` and `span`. Write at least two functions which implement one of the html containers like `i`, `div` or `span`. I recommend calling the `mkDiv` and `mkSpan` etc. For now we will not add any attributes to these containers. They should accept a `Html` value as input and return a `Html` value. And what they should do is add the respective opening and closing tags around the html value they have received as input.[5]

8.4.5 Html documents

Now we want to model a whole html document. First we will need to model the doctype.

1. Create a `Doctype` type with constructors for some of the most common html versions: `Html` (for html4) `Html5` (for html5) and `XHtml`.
2. **For the document itself we will create a `Document` type.** This type should have three fields.
   (a) `doctype :: Doctype`
   (b) `headSection :: Html`
   (c) `bodySection :: Html`
   Implement this type using record syntax. This allows us to manipulate the fields later.
3. Lastly we need a way to render it.
   Create a `renderDocument` function which returns a string that is the concatenation of:
   - The correct doctype string for the `Doctype`
   - The head html wrapped in `<head></head>`
   - The body html wrapped in `<body></body>`

8.4.6 Making the html editable

Until now we have only used `String` for the internal html. However we can do better. We want to be able to edit our html safely after we have created it. Also we want support for attributes.

1. Change the `Html` datatype such that[8]
   (a) It can either be a text node which contains only a `String`
   (b) It is a container node (such as `div`) which contains a string for the `containerTag`, a list of attribute/value pairs `containerAttributes`[6] and a list of `containerChildren`[7] Use a record here with the mentioned field names.

[4] You'll have to unwrap the input `Html` values to get acces to the strings within. Look for an operator in `Data.List` which appends two lists together. You can use this operator to combine the strings as well. Finally wrap it all back up into a new `Html` value.
[5] You'll again have to unwrap the `Html`, prepend the start tag and append the end tag to it. Finally wrap it all back up into a new `Html` value
[8] You can implement the different types of html by making it an algebraic datatype (`data`) with one constructor for the text node and one for the container node. Use record syntax for the latter.
[6] Pairs are the same a tuples. Both attribute and its value should be of type `String`.
[7] Children are again `Html` values.

2. Rewrite the `render` function to use the new type, and also render the attributes.[9]
3. Rewrite the `mkDiv` etc. functions to create the new type.[10]

8.4.7 Doing some inspection

Now that we have this fancier Html tree we can do interesting things. Implement the following queries as functions (they all return `Bool`).

- Is a supplied `Html` value a text node
- Does the node have a specific tag (hint: the type signature should be `:: String -> Html -> Bool`)
- How many attributes does the node have? (assuming no attribute occurs twice in the attribute list)[11]
- Does the node have a specific attribute (hint: the type signature should be `:: String -> Html -> Bool`)[12]

8.4.8 Implement a monoid

Implement the `Data.Monoid.Monoid` typeclass for `Html`.

`mappend` stands for *append*. The `m` is only added so it does not clash names. Think about which of the functions we have already implemented that appends.

`mempty` stands for *empty*. Think about what an empty element for `Html` would be.

Hint: The empty element should be such that appending an empty element to anything it should no change the original thing.

8.4.9 A typeclass for rendering

Define a typeclass for our render function. I propose to call the typeclass `Renderable r`, with one member function `render :: r -> String`.

Implement the `Renderable` typeclass for `Html`, `Doctype` and `Document`.[14]

8.4.10 Escaping (advanced)

Change the text node creation so it doesn't fail when illegal characters are found but instead replaces them with the xml escape sequences. The important thing to keep in mind here is that you need to replace single characters by strings of characters.[13]

[9] Some things that may come in handy here is the `map` function and the `concat` function. The first can be used (with an appropriate function) to transform for instance the list of `Html` children into a list of `String`. The latter can be used to concatenate a list of `String` into a single `String`.
Haskell supports calling functions recursively. Meaning you can for instance call `render` from within `render` to render a nested `Html` value.

[10] This can be nicely done using a partially applied `Container` constructor.

[11] This is the same as the length of the attribute list.

[12] To see if an element of a list satisfies a predicate there are two ways. Either using `map` and `any` or using `find`. I leave you to find out how to use these ;)

[14] You can use the `render` function for other types or nested data inside the definition of `render`. For instance when rendering `Document` you can use `render` on the `Doctype` or a `Html` value to render it.

[13] I'd recommend either to use `concatMap` or `foldr`.

Character	Escape
&	&amp;
<	&lt;
>	&gt;

footnotes

8.5 Simple I/O

8.5.1 Reverser

Write an application which reads a line from stdin and prints the same string back, but reversed.

Modify it so it keeps repeating this process forever.

8.5.2 Bulk rename

You will need the filepath and directory library.

Implement a function which:

- takes as argument a string
- then scans the current directory and finds all files
- renames each file by prepending the string argument to the filename

Advanced

Get the input string from the application arguments[1].

8.5.3 Combine files

You will need the filepath and directory library.

Implement a function which:

- Scans the current directory
- finds all files
- reads each file and collects the contents in a list
- writes a combined output file with the contents of each of the files concatenated.

Advanced

Instead of collecting to a list open the output file first and write each files contents to the output handle right away.

footnotes

[1] Use the getArgs function.

CHAPTER

NINE

INDICES AND TABLES

- genindex
- modindex
- search