Haskell MOOC

Part 1

by Joel Kaasinen (Nitor) and John Lång (University of Helsinki)

1 Lecture 1: ...And so It Begins

1.1 About the Course

This is an online course on Functional Programming that uses the Haskell programming language. You can study at your own pace. All the material and exercises are openly available.

This course is aimed at beginners who wish to learn functional programming, but also people who have experience with functional programming and want to learn Haskell in particular. The course assumes no previous knowledge, but knowing at least one programming language beforehand will make the course easier.

Working on the exercises involves knowing how to use the command line, and basic usage of the Git version control system.

This is part 1 of a two-part course. Part 1 covers the basics of Haskell syntax and features. You will learn about recursion, higher-order functions, algebraic data types and some of Haskell' s advanced features. However, part 1 will stick to pure functional programming, without side-effects. I/O and Monads will be introduced in part 2.

The course is split into 8 lectures. They are roughly the same size, but some lectures have more material than others. Each lecture set ends with 10-30 small programming exercises on the topics of the lecture.

1.2 Read These

In addition to this course material, the following sources might be useful if you feel like you're missing examples or explanations.

- The course pages
- The course channel on Telegram
- The course repository on Github contains the exercises and this material
- Additional resources
    - A Gentle Introduction to Haskell - an older and shorter tutorial, but still worth reading
    - Learn You a Haskell for Great Good! – a nice free introduction to Haskell
    - The Haskell School of Expression - slightly older but still relevant introduction to functional programming
    - Haskell Programming from First Principles - $59 e-book on Haskell, slow and long
    - The IRC channel `#haskell` on libera.chat is a nice place for beginners

1.3 Haskell

Haskell is

Functional – the basic building blocks of programs are functions. Functions can return functions and take functions as arguments. Also, the only looping construct in Haskell is recursion.

Pure - Haskell functions are pure, that is, they don't have side effects. Side effects mean things like reading a file, printing out text, or changing global variables. All inputs to a function must be in its arguments, and all outputs from a function in its return value. This sounds restricting, but makes reasoning about programs easier, and allows for more optimizations by the compiler.

Lazy - values are only evaluated when they are needed. This makes it possible to work with infinite data structures, and also makes pure programs more efficient.

Strongly typed - every Haskell value and expression has a type. The compiler checks the types at compile-time and guarantees that no type errors can happen at runtime. This means no AttributeErrors (a la Python), ClassCastExceptions (a la Java) or segmentation faults (a la C). The Haskell type system is very powerful and can help you design better programs.

Type inferred - in addition to checking the types, the compiler can deduce the types for most programs. This makes working with a strongly typed language easier. Indeed, most Haskell functions can be written completely without types. However programmers can still give functions and values type annotations to make finding type errors easier. Type annotations also make reading programs easier.

Garbage-collected - like most high-level languages these days, Haskell has automatic memory management via garbage collection. This means that the programmer doesn't need to worry about allocating or freeing memory, the language runtime handles all of it automatically.

Compiled - even though we mostly use Haskell via the interactive GHCi environment on this course, Haskell is a compiled language. Haskell programs can be compiled to very efficient binaries, and the GHC compiler is very good at optimising functional code into performant machine code.

You'll learn what these terms mean in practice during this course. Don't worry if some of them sound abstract right now.

See also: The Haskell Wiki page on Functional programming.

1.3.1 Features

Here's a showcase of some of the Haskell's cool features:

Higher-order functions – functions can take functions as arguments:

```
map length ["abc","abcdef"]
```

This results in `[3,6]`.

Anonymous functions aka lambdas – you can define single-use helper functions without giving them a name

```
filter (\x -> length x > 1) ["abc","d","ef"]
```

This results in `["abc","ef"]`.

Partial application – you can define new functions by giving another function only some of the arguments it needs. For example this multiplies all elements in a list by 3:

```
map (*3) [1,2,3]
```

Algebraic datatypes – a syntax for defining datatypes that can contain a number of different cases:

```
data Shape = Point | Rectangle Double Double | Circle Double
```

Now the type `Shape` can have values like `Point`, `Rectangle 3 6` and `Circle 5`

Pattern matching – defining functions based on cases that correspond to your data definitions:

```
area Point = 0
area (Rectangle width height) = width * height
area (Circle radius) = 2 * pi * radius
```

Lists – Unlike many languages, Haskell has a concise built-in syntax for lists. Lists can be built from other lists using *list comprehensions*. Here' s a snippet that generates names of even length from a set of options for first and last names:

```
[whole | first <- ["Eva", "Mike"],
         last <- ["Smith", "Wood", "Odd"],
         let whole = first ++ last,
         even (length whole)]
```

This results in `["EvaSmith","EvaOdd","MikeWood"]`. Thanks to the laziness of Haskell, we can even create so-called *infinite lists*:

```
primes = [ n | n <- [2..] , all (\k -> n `mod` k /= 0) [2..n `div`
         2] ]
```

The first ten prime numbers can be then obtained by evaluating

```
take 10 primes
```

This evaluates to `[2,3,5,7,11,13,17,19,23,29]`.

Parameterized types – you can define types that are parameterized by other types. For example `[Int]` is a list of `Int`s and `[Bool]` is a list of booleans. You can define typed functions that work on all kinds of lists, for example `reverse` has the type `[a] -> [a]` which means it takes a list containing any type `a`, and returns a list of the same type.

Type classes – another form of polymorphism where you can give a function a different implementation depending on the arguments' types. For example the `Show` type class defines the function `show` that can convert values of various types to strings. The `Num` type class defines arithmetic operators like + that work on all number types (`Int`, `Double`, `Complex`, ...).

1.3.2 Some History

A brief timeline of Haskell:

- 1930s: Lambda Calculus
- 1950s-1970s: Lisp, Pattern Matching, Scheme, ML
- 1978 John Backus: Can programming be liberated from the von Neumann style?
- 1987 A decision was made to unify the field of pure functional languages
- 1990 Haskell 1.0
- 1991 Haskell 1.1 (let-syntax, sections)
- 1992 Haskell 1.2, GHC
- 1996 Haskell 1.3 (Monads, do-syntax, type system improvements)
- 1999 Haskell 98
- 2000's: GHC development, many extensions to the language
- 2009 The Haskell 2010 standard
- 2010's: GHC development, Haskell Platform, Haskell Stack

The word 'haskel' means wisdom in Hebrew, but the name of the Haskell programming language comes from the logician Haskell Curry. The name Haskell comes from the Old Norse words áss (god) and ketill (helmet).

1.3.3 Uses of Haskell

Here are some examples of software projects that were written in Haskell.

- The Darcs distributed version control system
- The Sigma spam-prevention tool at Facebook
- The implementations of the PureScript and Elm programming languages are written in Haskell
- The Pandoc tool for converting between different document formats – it's also used to produce this course material
- The PostgREST server that exposes a HTTP REST API for a PostgreSQL database
- Functional consulting companies like Galois and Well-Typed have a long history of developing critical systems for clients in Haskell

See The Haskell Wiki and this blog post for more!

1.4 Running Haskell

The easiest way to get Haskell is to install the `stack` tool, see https://haskellstack.org. The exercises on this course are intended to work with Stack, so you should use it for now.

By the way, if you're interested in what Stack is, and how it relates to other Haskell tools like Cabal and GHC, read more here or here. We'll get back to Haskell packages and using them in detail in part 2 of the course.

For now, after installing Stack, just run `stack ghci` to get an interactive Haskell environment.

Note! GHC 8.10.7 has a GHCi bug that makes editing lines impossible on ARM-based systems. As a workaround, use `TERM=dumb stack ghci`. More info here.

1.5 Let's Start!

GHCi is the interactive Haskell interpreter. Here's an example session:

```
$ stack ghci
GHCi, version 9.2.8: https://www.haskell.org/ghc/  :? for help
Prelude> 1+1
2
Prelude> "asdf"
"asdf"
Prelude> reverse "asdf"
"fdsa"
Prelude> :type "asdf"
"asdf" :: [Char]
Prelude> tail "asdf"
"sdf"
Prelude> :type tail "asdf"
tail "asdf" :: [Char]
Prelude> :type tail
tail :: [a] -> [a]
Prelude> :quit
Leaving GHCi.
```

By the way, the first time you run `stack ghci` it will download GHC and some libraries, so don't worry if you see some output and have to wait for a while before getting the `Prelude>` prompt.

Let's walk through this. Don't worry if you don't understand things yet, this is just a first brush with expressions and types.

```
Prelude> 1+1
2
```

The `Prelude>` is the GHCi prompt. It indicates we can use the functions from the Haskell base library called Prelude. We evaluate 1 plus 1, and the result is 2.

```
Prelude> "asdf"
"asdf"
```

Here we evaluate a string literal, and the result is the same string.

```
Prelude> reverse "asdf"
"fdsa"
```

Here we compute the reverse of a string by applying the function `reverse` to the value `"asdf"`.

```
Prelude> :type "asdf"
"asdf" :: [Char]
```

In addition to evaluating expressions we can also ask for their type with the `:type` (abbreviated `:t`) GHCi command. The type of `"asdf"` is a list of characters. Commands that start with `:` are part of the user interface of GHCi, not part of the Haskell language.

```
Prelude> tail "asdf"
"sdf"
Prelude> :t tail "asdf"
tail "asdf" :: [Char]
```

The `tail` function works on lists and returns all except the first element of the list. Here we see `tail` applied to `"asdf"`. We also check the type of the expression, and it is a list of characters, as expected.

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

Finally, here's the type of the `tail` function. It takes a list of any type as an argument, and returns a list of the same type.

```
Prelude> :quit
Leaving GHCi.
```

That's how you quit GHCi.

1.6 Expressions and Types

Just like we saw in the GHCi example above, *expressions* and *types* are the bread and butter of Haskell. In fact, almost everything in a Haskell program is an expression. In particular, there are no *statements* like in Python, Java or C.

An expression has a *value* and a *type*. We write an expression and its type like this: `expression :: type`. Here are some examples:

Expression	Type	Value
`True`	`Bool`	`True`
`not True`	`Bool`	`False`
`"as" ++ "df"`	`[Char]`	`"asdf"`

1.6.1 Syntax of Expressions

Expressions consist of functions *applied* to arguments. Functions are *applied* (i.e. called) by placing the arguments after the name of the function – there is no special syntax for a function call.

Haskell	Python, Java or C
`f 1`	`f(1)`
`f 1 2`	`f(1,2)`

Parentheses can be used to *group* expressions (just like in math and other languages).

Haskell	Python, Java or C
`g h f 1`	`g(h,f,1)`
`g h (f 1)`	`g(h,f(1))`
`g (h f 1)`	`g(h(f,1))`
`g (h (f 1))`	`g(h(f(1)))`

Some function names are made special characters and they are used as operators: between their arguments instead of before them. Function calls *bind tighter* than

operators, just like multiplication binds tighter than addition.

Haskell	Python, Java or C
`a + b`	`a + b`
`f a + g b`	`f(a) + g(b)`
`f (a + g b)`	`f(a+g(b))`

PS. in Haskell, function application *associates left*, that is, `f g x y` is actually the same as `(((f g) x) y)`. We'll get back to this topic later. For now you can just think that `f g x y` is `f` applied to the arguments `g`, `x` and `y`.

1.6.2 Syntax of Types

Here are some basic types of Haskell to get you started.

Type	Literals	Use	Operations
`Int`	`1`, `2`, `-3`	Number type (signed, 64bit)	`+`, `-`, `*`, `div`, `mod`
`Integer`	`1`, `-2`, `900000000000000000`	Unbounded number type	`+`, `-`, `*`, `div`, `mod`
`Double`	`0.1`, `1.2e5`	Floating point numbers	`+`, `-`, `*`, `/`, `sqrt`
`Bool`	`True`, `False`	Truth values	`&&`, `||`, `not`
`String` aka `[Char]`	`"abcd"`, `""`	Strings of characters	`reverse`, `++`

As you can see, the names of types in Haskell start with a capital letter. Some values like `True` also start with a capital letter, but variables and functions start with a lower case letter (`reverse`, `not`, `x`). We'll get back to the meaning of capital letters in Lecture 2.

Function types are written using the `->` syntax:

- A function of one argument: `argumentType -> returnType`
- ... of two arguments: `argument1Type -> argument2Type -> returnType`
- ... of three arguments:
  `argument1Type -> argument2Type -> argument3Type -> returnType`

Looks a bit weird, right? We'll get back to this as well.

1.6.3 Note About Misleading Types

Sometimes, the types you see in GHCi are a bit different than what you'd assume. Here are two common cases.

```
Prelude> :t 1+1
1+1 :: Num a => a
```

For now, you should read the type `Num a => a` as "any number type". In Haskell, number literals are *overloaded* which means that they can be interpreted as any number type (e.g. `Int` or `Double`). We'll get back to what `Num a` actually means when we talk about *type classes* later.

```
Prelude> :t "asdf"
"asdf" :: [Char]
```

The type `String` is just an alias for the type `[Char]` which means "list of characters". We'll get back to lists on the next lecture! In any case, you can use `String` and `[Char]` interchangeably, but GHCi will mostly use `[Char]` when describing types to you.

1.7 The Structure of a Haskell Program

Here's a simple Haskell program that does some arithmetic and prints some values.

```
module Gold where

-- The golden ratio
phi :: Double
phi = (sqrt 5 + 1) / 2

polynomial :: Double -> Double
polynomial x = x^2 - x - 1

f x = polynomial (polynomial x)

main = do
  print (polynomial phi)
  print (f phi)
```

If you put this in a file called `Gold.hs` and run it with (for example) `stack runhaskell Gold.hs`, you should see this output

```
0.0
-1.0
```

Let’s walk through the file.

```
module Gold where
```

There is one Haskell *module* per source file. A module consists of *definitions*.

```
-- The golden ratio
```

This is a comment. Comments are not part of the actual program, but text for human readers of the program.

```
phi :: Double
phi = (sqrt 5 + 1) / 2
```

This is a definition of the constant `phi`, with an accompanying *type annotation* (also known as a *type signature*) `phi :: Double`. The type annotation means that `phi` has type `Double`. The line with a equals sign (=) is called an *equation*. The left hand side of the = is the expression we are defining, and the right hand side of the = is the definition.

In general a definition (of a function or constant) consists of an optional *type annotation* and one or more *equations*

```
polynomial :: Double -> Double
polynomial x = x^2 - x - 1
```

This is the definition of a function called `polynomial`. It has a type annotation and an equation. Note how an equation for a function differs from the equation of a constant by the presence of a parameter `x` left of the = sign. Note also that ^ is the power operator in Haskell, not bitwise xor like in many other languages.

```
f x = polynomial (polynomial x)
```

This is the definition of a function called `f`. Note the lack of type annotation. What is the type of `f`?

```
main = do
  print (polynomial phi)
  print (f phi)
```

This is a description of what happens when you run the program. It uses do-syntax and the IO Monad. We' ll get back to those in part 2 of the course.

1.8 Working with Examples

When you see an example definition like this

```
polynomial :: Double -> Double
polynomial x = x^2 - x - 1
```

you should usually play around with it. Start by running it. There are a couple of ways to do this.

If a definition fits on one line, you can just define it in GHCi:

```
Prelude> polynomial x = x^2 - x - 1
Prelude> polynomial 3.0
5.0
```

For a multi-line definition, you can either use `;` to separate lines, or use the special `:{ :}` syntax to paste a block of code into GHCi:

```
Prelude> :{
Prelude| polynomial :: Double -> Double
Prelude| polynomial x = x^2 - x - 1
Prelude| :}
Prelude> polynomial 3.0
5.0
```

Finally, you can paste the code into a new or existing `.hs` file, and then `:load` it into GHCi. If the file has already been loaded, you can also use `:reload`.

```
-- first copy and paste the definition into Example.hs, then run
        GHCi
Prelude> :load Example.hs
[1 of 1] Compiling Main             ( Example.hs, interpreted )
Ok, one module loaded.
*Main> polynomial 3.0
5.0
-- now you can edit the definition
*Main> :reload
[1 of 1] Compiling Main             ( Example.hs, interpreted )
Ok, one module loaded.
*Main> polynomial 3
3.0
```

After you’ ve run the example, try modifying it, or making another function that is similar but different. You learn programming by programming, not by reading!

1.8.1 Dealing with Errors

Since Haskell is a typed language, you’ ll pretty quickly bump into type errors. Here’ s an example of an error during a GHCi session:

```
Prelude> "string" ++ True

<interactive>:1:13: error:
    • Couldn't match expected type ‘[Char]’ with actual type ‘Bool’
    • In the second argument of ‘(++)’, namely ‘True’
      In the expression: "string" ++ True
      In an equation for ‘it’: it = "string" ++ True
```

This is the most common type error, “Couldn’ t match expected type” . Even though the error looks long and scary, it’ s pretty simple if you just read through it.

- The first line of the error message, `<interactive>:1:13: error:` tells us that the error occurred in GHCi. If we had loaded a file, we might instead get something like `Sandbox.hs:3:17: error:`, where `Sandbox.hs` is the name of the file, `3` is the line number and `17` is the number of a character in the line.

- The line
  • `Couldn't match expected type ‘[Char]’ with actual type ‘Bool’` tells us that the immediate cause for the error is that there was an expression of type `Bool`, when GHCi was expecting to find an expression of type `[Char]` “. The location of this error was indicated in the first line of the error message.

Note that the expected type is not always right. Giving type annotations by hand can help debugging typing errors.

- The line `• In the second argument of ‘(++)’, namely ‘True’` tells that the expression that had the wrong type was the second argument of the operator `(++)`. We’ ll learn later why it’ s surrounded by parentheses.

- The full expression with the error was `"string" ++ True`. As mentioned above, `String` is a type alias for `[Char]`, the type of character lists. The first argument to ++ was a list of characters, and since ++ can only combine two lists of the same type, the second argument should’ ve been of type `[Char]` too.

- The line `In an equation for ‘it’: it = "string" ++ True` says that the expression occurred in the definition of the variable `it`, which is a default variable name that GHCi uses for standalone expressions. If we had a line `x = "string" ++ True` in a file, or a declaration `let x = "string" ++ True` in GHCi, GHCi would print `In an equation for ‘x’: x = "string" ++ True` instead.

There are also others types of errors.

```
Prelude> True + 1

<interactive>:6:1: error:
    • No instance for (Num Bool) arising from a use of ‘+’
    • In the expression: True + 1
      In an equation for ‘it’: it = True + 1
```

This is the kind of error you get when you try to use a numeric function like + on something that’ s not a number.

The hardest error to track down is usually this:

```
Prelude> True +

<interactive>:10:7: error:
    parse error (possibly incorrect indentation or mismatched
         brackets)
```

There are many ways to cause it. Probably you’ re missing some characters somewhere. We’ ll get back to indentation later in this lecture.

1.8.2 Arithmetic

There's one thing in Haskell arithmetic that often trips up beginners, and that's division.

In Haskell there are two division functions, the `/` operator and the `div` function. The `div` function does integer division:

```
Prelude> 7 `div` 2
3
```

The `/` operator performs the usual division:

```
Prelude> 7.0 / 2.0
3.5
```

However, you can only use `div` on whole number types like `Int` and `Integer`, and you can only use `/` on decimal types like `Double`. Here's an example of what happens if you try to mix them up:

```
halve :: Int -> Int
halve x = x / 2
```

```
error:
    • No instance for (Fractional Int) arising from a use of ‘/’
    • In the expression: x / 2
      In an equation for ‘halve’: halve x = x / 2
```

Just try to keep this in mind for now. We'll get back to the difference between `/` and `div`, and what `Num` and `Fractional` mean when talking about type classes.

1.9 How Do I Get Anything Done?

So far you've seen some arithmetic, reversing a string and so on. How does one write actual programs in Haskell? Many of the usual programming constructs like loops, statements and assignment are missing from Haskell. Next, we'll go through the basic building blocks of Haskell programs:

- Conditionals
- Local definitions

- Pattern matching
- Recursion

1.9.1 Conditionals

In other languages, `if` is a *statement*. It doesn't have a value, it just conditionally executes other statements.

In Haskell, `if` is an *expression*. It has a value. It selects between two other expressions. It corresponds to the `?:` operator in C or Java.

```
// Java
int price = product.equals("milk") ? 1 : 2;
```

Python's conditional expressions are quite close to haskell's `if`:

```
# Python
price = 1 if product == "milk" else 2
```

This is how the same example looks in Haskell:

```
price = if product == "milk" then 1 else 2
```

Because Haskell's `if` *returns* a value, you **always** need an `else`!

1.9.1.1 Functions Returning `Bool`

In order to write if expressions, you need to know how to get values of type `Bool`. The most common way is comparisons. The usual ==, <, <=, > and >= operators work in Haskell. You can do ordered comparisons (<, >) on all sorts of numbers, and equality comparisons (==) on almost anything:

```
Prelude> "foo" == "bar"
False
Prelude> 5.0 <= 7.2
True
Prelude> 1 == 1
True
```

One oddity of Haskell is that the not-equals operator is written `/=` instead of the usual `!=`:

```
Prelude> 2 /= 3
True
Prelude> "bike" /= "bike"
False
```

Remember that in addition to these comparisons, you can get `Bool` values out of other `Bool` values by using the `&&` (“and”) and `||` (“or”) operators, and the `not` function.

1.9.1.2 Examples

```
checkPassword password = if password == "swordfish"
                         then "You're in."
                         else "ACCESS DENIED!"
```

```
absoluteValue n = if n < 0 then -n else n
```

```
login user password = if user == "unicorn73"
                      then if password == "f4bulous!"
                           then "unicorn73 logged in"
                           else "wrong password"
                      else "unknown user"
```

1.9.2 Local Definitions

Haskell has two different ways for creating local definitions: `let...in` and `where`.

`where` adds local definitions to a definition:

```
circleArea :: Double -> Double
circleArea r = pi * rsquare
    where pi = 3.1415926
          rsquare = r * r
```

`let...in` is an expression:

```
circleArea r = let pi = 3.1415926
                   rsquare = r * r
               in pi * rsquare
```

Local definitions can also be functions:

```
circleArea r = pi * square r
    where pi = 3.1415926
          square x = x * x
```

```
circleArea r = let pi = 3.1415926
                   square x = x * x
               in pi * square r
```

We'll get back to the differences between `let` and `where`, but mostly you can use which ever you like.

1.9.3 A Word About Immutability

Even though things like `pi` above are often called *variables*, I've chosen to call them *definitions* here. This is because unlike variables in Python or Java, the values of these definitions can't be changed. Haskell variables aren't boxes into which you can put new values, Haskell variables name a value (or rather, an expression) and that's it.

We'll talk about immutability again later on this course, but for now it's enough to know that things like this don't work.

```
increment x = let x = x+1
              in x
```

This is just an infinite loop, because it tries to define a new variable `x` with the property `x = x+1`. Thus when evaluating `x`, Haskell just keeps computing `1+1+1+1+...` indefinitely.

```
compute x = let a = x+1
                a = a*2
            in a
```

```
error:
    Conflicting definitions for ‘a’
    Bound at: <interactive>:14:17
              <interactive>:15:17
```

Here we get a straightforward error when we’re trying to “update” the value of a.

As a remark, local definitions can *shadow* the names of variables defined elsewhere. Shadowing is not a side-effect. Instead, shadowing creates a new variable within a more restricted scope that uses the same name as some variable in the outer scope. For example, all of the functions `f`, `g`, and `h` below are legal:

```
x :: Int
x = 5

f :: Int -> Int
f x = 2 * x

g :: Int -> Int
g y = x where x = 6

h :: Int -> Int
h x = x where x = 3
```

If we apply them to the global constant `x`, we see the effects of shadowing:

```
f 1 ==> 2
g 1 ==> 6
h 1 ==> 3

f x ==> 10
g x ==> 6
h x ==> 3
```

It is best to always choose new names for local variables, so that shadowing never happens. That way, the reader of the code will understand where the variables that are used in an expression come from. Note that in the following example, `f` and `g` don’t shadow each others’ arguments:

```
f :: Int -> Int
f x = 2 * x + 1
```

```haskell
g :: Int -> Int
g x = x - 2
```

1.9.4 Pattern Matching

A definition (of a function) can consist of multiple *equations*. The equations are matched in order against the arguments until a suitable one is found. This is called *pattern matching*.

Pattern matching in Haskell is very powerful, and we'll keep learning new things about it along this course, but here are a couple of first examples:

```haskell
greet :: String -> String -> String
greet "Finland" name = "Hei, " ++ name
greet "Italy"   name = "Ciao, " ++ name
greet "England" name = "How do you do, " ++ name
greet _         name = "Hello, " ++ name
```

The function `greet` generates a greeting given a country and a name (both `Strings`). It has special cases for three countries, and a default case. This is how it works:

```
Prelude> greet "Finland" "Pekka"
"Hei, Pekka"
Prelude> greet "England" "Bob"
"How do you do, Bob"
Prelude> greet "Italy" "Maria"
"Ciao, Maria"
Prelude> greet "Greenland" "Jan"
"Hello, Jan"
```

The special pattern _ matches anything. It's usually used for default cases. Because patterns are matched in order, it's important to (*usually*) put the _ case last. Consider:

```haskell
brokenGreet _         name = "Hello, " ++ name
brokenGreet "Finland" name = "Hei, " ++ name
```

Now the first case gets selected for all inputs.

```
Prelude> brokenGreet "Finland" "Varpu"
"Hello, Varpu"
```

```
Prelude> brokenGreet "Sweden" "Ole"
"Hello, Ole"
```

GHC even gives you a warning about this code:

```
<interactive>:1:1: warning: [-Woverlapping-patterns]
    Pattern match is redundant
    In an equation for ‘brokenGreet’: brokenGreet "Finland" name = ...
```

Some more examples follow. But first let’s introduce the standard library function `show` that can turn (almost!) anything into a string:

```
Prelude> show True
"True"
Prelude> show 3
"3"
```

So, here’s an example of a function with pattern matching and a default case that actually uses the value (instead of just ignoring it with _):

```
describe :: Integer -> String
describe 0 = "zero"
describe 1 = "one"
describe 2 = "an even prime"
describe n = "the number " ++ show n
```

This is how it works:

```
Prelude> describe 0
"zero"
Prelude> describe 2
"an even prime"
Prelude> describe 7
"the number 7"
```

You can even pattern match on multiple arguments. Again, the equations are tried in order. Here’s a reimplementation of the `login` function from earlier:

```
login :: String -> String -> String
login "unicorn73" "f4bulous!" = "unicorn73 logged in"
login "unicorn73" _           = "wrong password"
login _           _           = "unknown user"
```

1.9.5 Recursion

In Haskell, all sorts of loops are implemented with recursion. Function calls are very efficient, so you don't need to worry about performance. (We'll talk about performance later).

Learning how to do simple things with recursion in Haskell will help you use recursion on more complex problems later. Recursion is also often a useful way for thinking about solving harder problems.

Here's our first recursive function which computes the factorial. In mathematics, factorial is the product of *n* first positive integers and is written as *n!*. The definition of factorial is

*n! = n * (n-1) * ... * 1*

For example, *4! = 4*3*2*1 = 24*. Well anyway, here's the Haskell implementation of factorial:

```
factorial :: Int -> Int
factorial 1 = 1
factorial n = n * factorial (n-1)
```

This is how it works. We use `==>` to mean "evaluates to".

```
factorial 3
  ==> 3 * factorial (3-1)
  ==> 3 * factorial 2
  ==> 3 * 2 * factorial 1
  ==> 3 * 2 * 1
  ==> 6
```

What happens when you evaluate `factorial (-1)`?

Here's another example:

```haskell
-- compute the sum 1^2+2^2+3^2+...+n^2
squareSum 0 = 0
squareSum n = n^2 + squareSum (n-1)
```

A function can call itself recursively multiple times. As an example let's consider the *Fibonacci sequence* from mathematics. The Fibonacci sequence is a sequence of integers with the following definition.

> The sequence starts with 1, 1. To get the next element of the sequence, sum the previous two elements of the sequence.

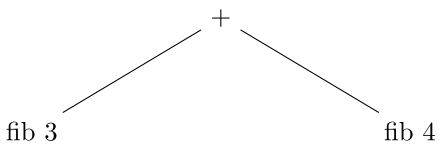
The first elements of the Fibonacci sequence are 1, 1, 2, 3, 5, 8, 13 and so on. Here's a function `fibonacci` which computes the `n`th element in the Fibonacci sequence. Note how it mirrors the mathematical definition.

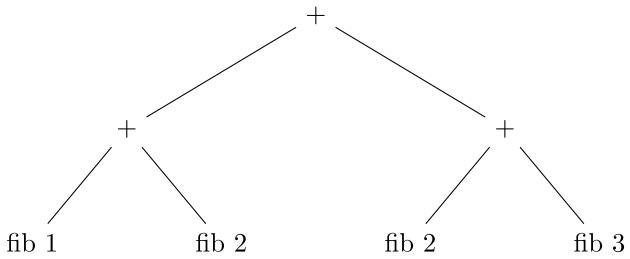
```haskell
-- Fibonacci numbers, slow version
fibonacci 1 = 1
fibonacci 2 = 1
fibonacci n = fibonacci (n-2) + fibonacci (n-1)
```

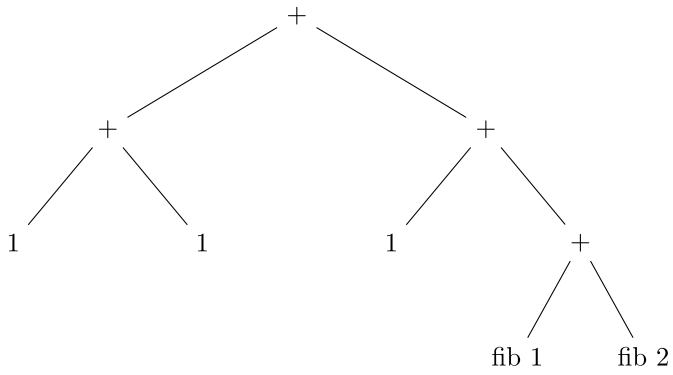
Here's how `fibonacci 5` evaluates:

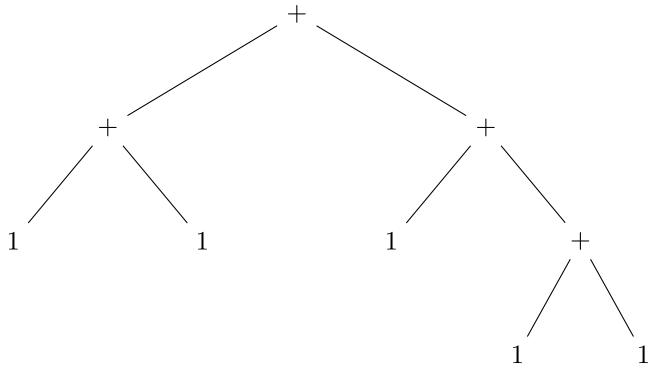
```haskell
fibonacci 5
  ==> fibonacci 3                 + fibonacci 4
  ==> (fibonacci 1 + fibonacci 2) + fibonacci 4
  ==> (    1       +       1    ) + fibonacci 4
  ==> (    1       +       1    ) + (fibonacci 2 + fibonacci 3)
  ==> (    1       +       1    ) + (fibonacci 2 + (fibonacci 1 +
         fibonacci 2))
  ==> (    1       +       1    ) + (    1       + (    1       +
         1      ))
  ==> 5
```

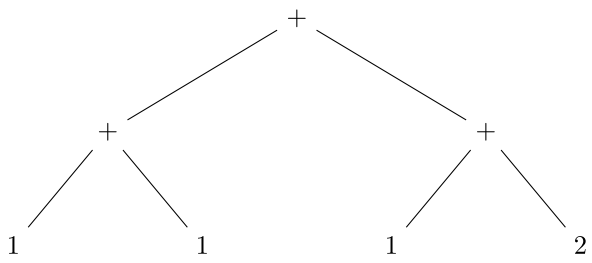
Note how `fibonacci 3` gets evaluated twice and `fibonacci 2` three times. This is not the most efficient implementation of the `fibonacci` function. We'll get back to this in the next lecture. Another way to think about the evaluation of the fibonacci function is to visualize it as a tree (we abbreviate `fibonacci` as `fib`):

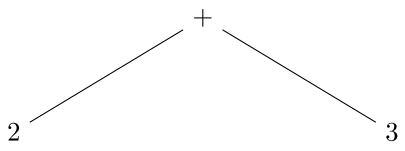

+
fib 3
fib 4


+
+
+
fib 1
fib 2
fib 2
fib 3


+
+
+
1
1
1
+
fib 1
fib 2


+
+
+
1
1
1
+
1
1





5

This tree then exaclty corresponds with the expression `(1 + 1) + (1 + (1 + 1))`. Recursion can often produce chain-like, tree-like, nested, or loopy structures and computations. Recursion is one of the main techniques in functional programming, so it's worth spending some effort in learning it.

1.10 All Together Now!

Finally, here's a complete Haskell module that uses ifs, pattern matching, local defintions and recursion. The module is interested in the *Collatz conjecture*, a famous open problem in mathematics. It asks:

> Does the Collatz sequence eventually reach 1 for all positive integer initial values?

The Collatz sequence is defined by taking any number as a starting value, and then repeatedly performing the following operation:

- if the number is even, divide it by two
- if the number is odd, triple it and add one

As an example, the Collatz sequence for 3 is: 3, 10, 5, 16, 8, 4, 2, 1, 4, 2, 1, 4, 2, 1 ... As you can see, once the number reaches 1, it gets caught in a loop.

```haskell
module Collatz where

-- one step of the Collatz sequence
step :: Integer -> Integer
step x = if even x then down else up
  where down = div x 2
        up = 3*x+1

-- collatz x computes how many steps it takes for the Collatz
        sequence
-- to reach 1 when starting from x
collatz :: Integer -> Integer
collatz 1 = 0
collatz x = 1 + collatz (step x)

-- longest finds the number with the longest Collatz sequence for
        initial values
-- between 0 and upperBound
longest :: Integer -> Integer
longest upperBound = longest' 0 0 upperBound

-- helper function for longest
longest' :: Integer -> Integer -> Integer -> Integer
-- end of recursion, return longest length found
longest' number _ 0 = number
-- recursion step: check if n has a longer Collatz sequence than the
        current known longest
longest' number maxlength n =
  if len > maxlength
  then longest' n len (n-1)
  else longest' number maxlength (n-1)
  where len = collatz n
```

We can load the program in GHCi and play with it.

```
$ stack ghci
GHCi, version 9.2.8: https://www.haskell.org/ghc/  :? for help
Prelude> :load Collatz.hs
[1 of 1] Compiling Collatz          ( Collatz.hs, interpreted )
Ok, one module loaded.
*Collatz>
```

Let’s verify that our program computes the start of the Collatz sequence for 3 correctly.

```
*Collatz> step 3
10
*Collatz> step 10
5
*Collatz> step 5
16
```

How many steps does it take for 3 to reach 1?

```
*Collatz> collatz 3
7
```

What' s the longest Collatz sequence for a starting value under 10? What about 100?

```
*Collatz> longest 10
9
*Collatz> longest 100
97
```

The lengths of these Collatz sequences are:

```
*Collatz> collatz 9
19
*Collatz> collatz 97
118
```

1.11 A Word About Indentation

The previous examples have been fancily indented. In Haskell indentation matters, a bit like in Python. The complete set of rules for indentation is hard to describe, but you should get along fine with these rules of thumb:

1. Things that are grouped together start from the same column
2. If an expression (or equation) has to be split on to many lines, increase indentation

While you can get away with using tabs, it is highly recommended to use spaces for all indenting.

Some examples are in order.

These all are ok:

```
i x = let y = x+x+x+x+x+x in div y 5

-- let and in are grouped together, an expression is split
j x = let y = x+x+x
              +x+x+x
      in div y 5

-- the definitions of a and b are grouped together
k = a + b
  where a = 1
        b = 1

l = a + b
  where
    a = 1
    b = 1
```

These are not ok:

```
-- indentation not increased even though expression split on many
         lines
i x = let y = x+x+x+x+x+x
in div y 5

-- indentation not increased even though expression is split
j x = let y = x+x+x
      +x+x+x
      in div y 5

-- grouped things are not aligned
k = a + b
  where a = 1
      b = 1

-- grouped things are not aligned
l = a + b
  where
    a = 1
     b = 1

-- where is part of the equation, so indentation needs to increase
l = a + b
where
  a = 1
  b = 1
```

If you make a mistake with the indentation, you'll typically get a parse error like this:

```
Indent.hs:2:1: error: parse error on input 'where'
```

The error includes the line number, so just go over that line again. If you can't seem to get indentation to work, try putting everything on just one long line at first.

1.12 Quiz

At the end of each lecture you'll find a quiz like this. The quizes aren't graded, they're just here to help you check you've understood the chapter. You can check your answer by clicking on an option. You'll see a green background if you were right, a red one if you were wrong. Feel free to guess as many times as you want, just make sure you understand why the right option is right in the end.

What is the Haskell equivalent of the C/Java/Python expression `combine(prettify(lawn),construct(house,concrete))`?

a. `combine prettify (lawn) construct (house concerete)`
b. `combine (prettify lawn (counstruct house concrete))`
c. `combine (prettify lawn) (construct house concrete)`

What is the C/Java/Python equivalent of the Haskell expression `send metric (double population + increase)`?

a. `send(metric(double(population+increase)))`
b. `send(metric(double(population)+increase))`
c. `send(metric,double(population)+increase)`
d. `send(metric,double(population+increase))`

Which one of the following claims is true in Haskell?

a. Every value has a type
b. Every type has a value
c. Every statement has a type

Which one of the following claims is true in Haskell?

a. It's impossible to reuse the name of a variable
b. It's possible to reassign a value to a variable
c. An `if` always requires both `then` and `else`

What does the function `f x = if even (x + 1) then x + 1 else f (x - 1)` do?

a. Maps every value `x` to the least even number greater than or equal to `x`
b. Maps every value `x` to the greatest even number less than or equal to `x`
c. Maps every value to itself

Why is `3 * "F00"` not valid Haskell?

a. `3` and `"F00"` have different types
b. All numeric values need a decimal point
c. `"F00"` needs the prefix "0x"

Why does `7.0 `div` 2` give an error?

a. Because `div` is not defined for the type `Double`
b. Because `div` is not defined for the type `Int`
c. Because `` `...` `` is used for delimiting strings.

1.13 Working on the Exercises

The course materials, including exercises, are available in a Git repository on GitHub at https://github.com/moocfi/haskell-mooc. If you're not familiar with Git, see GitHub's instructions on cloning a repository.

Once you've cloned the `haskell-mooc` repository, go into the `exercises` directory. To download and build dependencies needed for running the exercise tests (such as the correct version of GHC and various libraries), run following command in your terminal:

```
$ stack build
```

Do note that the dependencies are multiple gigabytes and it will take a while for the command to finish.

Note! Here are some fixes for common problems with `stack build`:

- If you get an error like `While building package zlib-0.6.2.3`, you need to install the zlib library headers. The right command for Ubuntu is `sudo apt install zlib1g-dev`.
- If you get an error like `Downloading lts-18.18 build plan ... RedownloadInvalidResponse`, your version of stack is too old. Run `stack upgrade` to get a newer one.

There are primarily two types of files in the `exercises` directory: exercise sets named `SetNX.hs` and accompanying test program for the exercises named `SetNXTest.hs`. Both are Haskell source files, but only the exercise file should to be edited when

solving the exercises. Instructions to all individual exercises are embedded in the exercise file as comments.

Use the tests file to check your answers. For example when you have solved some of the exercises in `Set1.hs`, run the following command:

```
$ stack runhaskell Set1Test.hs
```

The output of the tests looks something like this:

```
===== EXERCISE 1
+++++ Pass
===== EXERCISE 2
+++++ Pass
===== EXERCISE 3
*** Failed! Falsified (after 2 tests and 1 shrink):
quadruple 1
  Expected: 4
  Was: 2

----- Fail
===== EXERCISE 4
+++++ Pass
===== EXERCISE 5
+++++ Pass
===== EXERCISE 6
+++++ Pass
===== EXERCISE 7
+++++ Pass
===== EXERCISE 8
+++++ Pass
===== EXERCISE 9
+++++ Pass
===== EXERCISE 10
+++++ Pass
===== EXERCISE 11
+++++ Pass
===== EXERCISE 12
+++++ Pass
===== EXERCISE 13
+++++ Pass
===== EXERCISE 14
+++++ Pass
===== EXERCISE 15
+++++ Pass
===== EXERCISE 16
+++++ Pass
===== EXERCISE 17
+++++ Pass
===== EXERCISE 18
```

```
+++++ Pass
===== EXERCISE 19
+++++ Pass
===== TOTAL
11011111111111111111
18 / 19
```

In the example above, I've made a mistake in exercise 3.

To make debugging faster and more straightforward, I can load my exercise file in GHCi, which allows me to evaluate any top-level function manually. For instance I can verify the above mistake by:

```
$ stack ghci Set1.hs
GHCi, version 9.2.8: https://www.haskell.org/ghc/  :? for help
[1 of 2] Compiling Mooc.Todo        ( Mooc/Todo.hs, interpreted )
[2 of 2] Compiling Set1             ( Set1.hs, interpreted )
Ok, two modules loaded.
*Set1> quadruple 1
2
```

Once you're done with an exercise set, you can turn it in on the Submit page on the course pages. After that you can see the results of your submission on the Results page and your total score on the My status page.

Note! You may turn in an exercise set as many times as you want.

Note! If you don't want to use Stack or can't get it working, you should also be able to run the tests with Cabal like this:

```
$ cabal v2-build
$ cabal v2-exec runhaskell Set1Test.hs
```

1.13.1 Model Solutions

Once you've successfully completed all the exercises in a set, you can view the model solutions on the My status page. It's useful to glance at the model solutions, they might show you a technique you've missed!

1.14 Exercises

- Set1

2 Lecture 2: Either You Die a Hero...

- More about recursion
- Guards
- More types: lists, `Maybe`, `Either`
- Polymorphism

2.1 Recursion and Helper Functions

Often you'll find you need helper variables in recursion to keep track of things. You can get them by defining a helper function with more arguments. Analogy: arguments of the helper function are variables you update in your loop.

Here's an example of how you would convert a loop (in Java or Python) into a recursive helper function in Haskell.

Java:

```java
public String repeatString(int n, String str) {
    String result = "";
    while (n>0) {
        result = result+str;
        n = n-1;
    }
    return result;
}
```

Python:

```python
def repeatString(n, str):
    result = ""
    while n>0:
        result = result+str
        n = n-1
    return result
```

Haskell:

```haskell
repeatString n str = repeatHelper n str ""

repeatHelper n str result = if (n==0)
```

```
                            then result
                            else repeatHelper (n-1) str
          (result++str)
```

```
Prelude> repeatString 3 "ABC"
"ABCABCABC"
```

You might have noticed that the Java and Python implementations look a bit weird since they use while loops instead of for loops. This is because this way the conversion to Haskell is more straightforward.

This can be made a bit tidier by using pattern matching instead of an `if`:

```
repeatString n str = repeatHelper n str ""

repeatHelper 0 _   result = result
repeatHelper n str result = repeatHelper (n-1) str (result++str)
```

Here’s another example with more variables: computing fibonacci numbers efficiently.

Java:

```
public int fibonacci(int n) {
    int a = 0;
    int b = 1;
    while (n>1) {
        int c = a+b;
        a=b;
        b=c;
        n--;
    }
    return b;
}
```

Python:

```
def fibonacci(n):
    a = 0
    b = 1
    while n>1:
        c = a+b
        a = b
```

```
        b = c
        n = n-1
    return b
```

Haskell:

```haskell
-- fibonacci numbers, fast version
fibonacci :: Integer -> Integer
fibonacci n = fibonacci' 0 1 n

fibonacci' :: Integer -> Integer -> Integer -> Integer
fibonacci' a b 1 = b
fibonacci' a b n = fibonacci' b (a+b) (n-1)
```

Take a while to study these and note how the Haskell recursion has the same format as the loop.

Sidenote: Haskell programs often use the apostrophe to name helper functions and alternative versions of functions. Thus the name `fibonacci'` for the helper function above. Names like `foo'` are usually read *foo prime* (like in mathematics).

I said earlier that this version of fibonacci is more efficient. Can you see why? The answer is that there are less recursive calls. The expression `fibonacci' _ _ n` calls `fibonacci' _ _ (n-1)` once, and this means that we can compute `fibonacci' _ _ n` in `n` steps.

This type of recursion where a function just directly calls itself with different arguments is called *tail recursion*. As you've seen above, tail recursion corresponds to loops. This is why tail recursion is often fast: the compiler can generate a loop in machine code when it sees tail recursion.

2.2 Guards

Before we move on to new types, let's go over one more piece of Haskell syntax.

The `if then else` is often a bit cumbersome, especially when you have multiple cases. An easier alternative is Haskell's *conditional definition* or *guarded definition*. This is a bit like pattern matching in that you have multiple equations, but you can have arbitrary code deciding which equation to use. Guarded definitions look like this:

```
f x y z
  | condition1 = something
```

```
  | condition2 = other
  | otherwise  = somethingother
```

A condition can be any expression of type `Bool`. The first condition that evaluates to `True` is chosen. The word `otherwise` is just an alias for `True`. It is used to mark the default case.

2.2.1 Examples

Here are some examples of using guards. First off, we have a function that describes the given number. Note how it is important to have the `"Two"` case before the `"Even"` case.

```
describe :: Int -> String
describe n
  | n==2      = "Two"
  | even n    = "Even"
  | n==3      = "Three"
  | n>100     = "Big!!"
  | otherwise = "The number "++show n
```

Here is factorial, implemented with guards instead of pattern matching. Unlike the pattern-matching version, this one doesn’ t loop forever with negative inputs.

```
factorial n
  | n<0       = -1
  | n==0      = 1
  | otherwise = n * factorial (n-1)
```

You can even combine guards with pattern matching. Here’ s the implementation of a simple age guessing game:

```
guessAge :: String -> Int -> String
guessAge "Griselda" age
    | age < 47 = "Too low!"
    | age > 47 = "Too high!"
    | otherwise = "Correct!"
guessAge "Hansel" age
    | age < 12 = "Too low!"
    | age > 12 = "Too high!"
    | otherwise = "Correct!"
guessAge name age = "Wrong name!"
```

```
Prelude> guessAge "Griselda" 30
"Too low!"
Prelude> guessAge "Griselda" 60
"Too high!"
Prelude> guessAge "Griselda" 47
"Correct!"
Prelude> guessAge "Bob" 30
"Wrong name!"
Prelude> guessAge "Hansel" 10
"Too low!"
```

2.3 Lists

So far we' ve always worked with single values like number or booleans. Strings contain multiple characters, but still in some sense a string is just one piece of information. In order to be able to do actual programming, we need to handle variable numbers of items. For this we need *data structures*.

The basic datastructure in Haskell is the list. Lists are used to store multiple values of the same type (in other words, Haskell lists are homogeneous). This is what a list literal looks like:

```
[0,3,4,1+1]
```

A list type is written as `[Element]`, where `Element` is the type of the lists elements. Here are some more list expressions and their types:

```
[True,True,False] :: [Bool]
["Moi","Hei"] :: [String]
[] :: [a]                    -- more about this later
[[1,2],[3,4]] :: [[Int]]     -- a list of lists
[1..7] :: [Int]              -- range syntax, value [1,2,3,4,5,6,7]
```

Haskell lists are implemented as singly-linked lists. We' ll return to this later.

2.3.1 List Operations

The Haskell standard library comes with lots of functions that operate on lists. Here are some of the most important ones, together with their types. We' ll get back to what `[a]` actually means in a second, but for now you can imagine it means "any list".

```haskell
head :: [a] -> a            -- returns the first element
last :: [a] -> a            -- returns the last element
tail :: [a] -> [a]          -- returns everything except the first
        element
init :: [a] -> [a]          -- returns everything except the last
        element
take :: Int -> [a] -> [a]   -- returns the n first elements
drop :: Int -> [a] -> [a]   -- returns everything except the n first
        elements
(++) :: [a] -> [a] -> [a]   -- lists are catenated with the ++
        operator
(!!) :: [a] -> Int -> a     -- lists are indexed with the !!
        operator
reverse :: [a] -> [a]       -- reverse a list
null :: [a] -> Bool         -- is this list empty?
length :: [a] -> Int        -- the length of a list
```

Sidenote: the last two operations (`null` and `length`) actually have more generic types, but here I'm pretending that you can only use them on lists.

Lists can be compared with the familiar == operator.

Do you remember this from our first GHCi session?

```haskell
Prelude> :t "asdf"
"asdf" :: [Char]
```

This means that `String` is just an alias for `[Char]`, which means string is a list of characters. This means you can use all list operations on strings!

Some list operations come from the module `Data.List`. You can import a module in code or in GHCi with the `import Data.List` syntax. One example is the `sort` function which sorts a list:

```haskell
Prelude> import Data.List
Prelude Data.List> sort [1,0,5,3]
[0,1,3,5]
```

Note how the set of imported modules shows up in the GHCi prompt.

2.3.2 Examples

Here are some examples of working with lists. In this case, instead of showing you output from GHCi, I merely use `==>` to show what an expression evaluates to.

Indexing a list:

```
[7,10,4,5] !! 2
  ==> 4
```

Defining a function that discards the 3rd and 4th elements of a list using `take` and `drop`:

```
f xs = take 2 xs ++ drop 4 xs
```

```
f [1,2,3,4,5,6]  ==>  [1,2,5,6]
f [1,2,3]        ==>  [1,2]
```

Rotating a list by taking the first element and moving it to the end:

```
g xs = tail xs ++ [head xs]
```

```
g [1,2,3]      ==>  [2,3,1]
g (g [1,2,3])  ==>  [3,1,2]
```

Here's an example of the range syntax:

```
reverse [1..4] ==> [4,3,2,1]
```

2.4 A Word About Immutability

Because Haskell is pure, it also means that functions can't *mutate* (change) their inputs. Mutation is a side effect, and Haskell functions are only allowed output via their return value. This means that Haskell list functions always return a new list. In practice:

```
Prelude> list = [1,2,3,4]
Prelude> reverse list
[4,3,2,1]
Prelude> list
[1,2,3,4]
Prelude> drop 2 list
[3,4]
Prelude> list
[1,2,3,4]
```

This might seem very inefficient but it turns out it can be both performant and quite useful. We'll get back to how Haskell datastructures work in a later lecture.

2.5 A Word About Type Inference and Polymorphism

So what does a type like `head :: [a] -> a` mean? It means given a list that contains elements of any type `a`, the return value will be of the same type `a`.

In this type, `a` is a *type variable*. Type variables are types that start with a small letter, e.g. `a`, `b`, `thisIsATypeVariable`. A type variable means a type that is not yet known, or in other words a type that could be anything. Type variables can turn into *concrete types* (e.g. `Bool`) by the process of *type inference* (also called *unification*).

Let's have a look at some examples. If we apply `head` to a list of booleans, type inference will compare the type of head's argument, `[a]`, with the type of the actual argument, `[Bool]` and deduce that `a` must be `Bool`. This means that the return type of `head` will in this case also be `Bool`.

```
head :: [a] -> a
head [True,False] :: Bool
```

The function `tail` takes a list, and returns a list of the same type. If we apply `tail` to a list of booleans, the return value will also be a list of booleans.

```
tail :: [a] -> [a]
tail [True,False] :: [Bool]
```

If types don't match, we get a type error. Consider the operator ++ which takes two lists of the same type, as we can see from its type `[a] -> [a] -> [a]`. If we try to apply ++ to a list of booleans and a list of characters we get an error. This is what happens in GHCi:

```
Prelude> [True,False] ++ "Moi"

<interactive>:1:16:
    Couldn't match expected type `Bool' against inferred type `Char'
      Expected type: [Bool]
      Inferred type: [Char]
    In the second argument of `(++)', namely `"Moi"'
    In the expression: [True, False] ++ "Moi"
```

Type inference is really powerful. It uses the simple process of unification to get us types for practically any Haskell expression. Consider these two functions:

```
f xs ys = [head xs, head ys]
g zs = f "Moi" zs
```

We can ask GHCi for their types, and we will see that type inference has figured out that the two arguments to `f` must have the same type, since their heads get put into the same list.

```
Prelude> :t f
f :: [a] -> [a] -> [a]
```

The function `g`, which fixed one of the arguments of `f` to a string (i.e. `[Char]`) gets a narrower type. Type inference has decided that the argument `zs` to `g` must also have type `[Char]`, since otherwise the type of `f` would not match the call to `f`.

```
Prelude> :t g
g :: [Char] -> [Char]
```

2.5.1 Sidenote: Some Terminology

In a type like `[Char]` we call `Char` a *type parameter*. A type like the list type that needs a type parameter is called a *parameterized type*.

The fact that a function like `head` can be used with many different types of arguments is called *polymorphism*. The `head` function is said to be *polymorphic*. There are many forms of polymorphism, and this Haskell form that uses type variables is called *parametric polymorphism*.

2.5.2 Sidenote: Type Annotations

Since Haskell has type inference, you don't need to give any type annotations. However even though type annotations aren't required, there are multiple reasons to add them:

1. They act as documentation
2. They act as assertions that the compiler checks: help you spot mistakes
3. You can use type annotations to give a function a narrower type than Haskell infers

A good rule of thumb is to give top-level definitions type annotations.

2.6 The `Maybe` Type

In addition to the list type, Haskell has other parameterized types too. Let's look at a very common and useful one: the `Maybe` type.

Sometimes an operation doesn't have a valid return value (E.g. division by zero.). We have a couple of options in this situation. We can use an error value, like `-1`. This is a bit ugly, not always possible. We can throw an exception. This is impure. In some other languages we would return a special null value that exists in (almost) all types. However Haskell does not have a null.

The solution Haskell offers us instead is to change our return type to a `Maybe` type. This is pure, safe and neat. The type `Maybe a` has two *constructors*: `Nothing` and `Just`. `Nothing` is just a constant, but `Just` takes a parameter. More concretely:

Type	Values
`Maybe Bool`	`Nothing`, `Just False`, `Just True`
`Maybe Int`	`Nothing`, `Just 0`, `Just 1`, ...
`Maybe [Int]`	`Nothing`, `Just []`, `Just [1,1337]`, ...

You can think of `Maybe a` as being a bit like `[a]` except there can only be 0 or 1 elements, not more. Alternatively, you can think of `Maybe a` introducing a null value to the type `a`. If you're familiar with Java, `Maybe Integer` is the Haskell equivalent of Java's `Optional<Integer>`.

You can create `Maybe` values by either specifying `Nothing` or `Just someOtherValue`:

```
Prelude> :t Nothing
Nothing :: Maybe a
Prelude> Just "a camel"
Just "a camel"
Prelude> :t Just "a camel"
Just "a camel" :: Maybe [Char]   -- the same as Maybe String
Prelude> Just True
```

```
Just True
Prelude> :t Just True
Just True :: Maybe Bool
```

```
-- given a password, return (Just username) if login succeeds,
        Nothing otherwise
login :: String -> Maybe String
login "f4bulous!" = Just "unicorn73"
login "swordfish" = Just "megahacker"
login _           = Nothing
```

You use a `Maybe` value by pattern matching on it. Usually you define patterns for the `Nothing` and `Just something` cases. Some examples:

```
-- Multiply an Int with a Maybe Int. Nothing is treated as no
        multiplication at all.
perhapsMultiply :: Int -> Maybe Int -> Int
perhapsMultiply i Nothing = i
perhapsMultiply i (Just j) = i*j   -- Note how j denotes the value
        inside the Just
```

```
Prelude> perhapsMultiply 3 Nothing
3
Prelude> perhapsMultiply 3 (Just 2)
6
```

```
intOrZero :: Maybe Int -> Int
intOrZero Nothing = 0
intOrZero (Just i) = i

safeHead :: [a] -> Maybe a
safeHead xs = if null xs then Nothing else Just (head xs)

headOrZero :: [Int] -> Int
headOrZero xs = intOrZero (safeHead xs)
```

```
headOrZero []  ==> intOrZero (safeHead [])  ==> intOrZero Nothing
        ==> 0
```

```
headOrZero [1] ==> intOrZero (safeHead [1]) ==> intOrZero (Just 1)
          ==> 1
```

2.7 Sidenote: Constructors

As you can see above, we can pattern match on the constructors of `Maybe`: `Just` and `Nothing`. We'll get back to what constructors mean later. For now it's enough to note that constructors are special values that start with a capital letter that you can pattern match on.

Other constructors that we've already seen include the constructors of `Bool` – `True` and `False`. We'll introduce the constructors of the list type on the next lecture.

Constructors can be used just like Haskell values. Constructors that take no arguments like `Nothing`, and `False` are just constants. Constructors like `Just` that take an argument behave like functions. They even have function types!

```
Prelude> :t Just
Just :: a -> Maybe a
```

2.8 The `Either` type

Sometimes it would be nice if you could add an error message or something to `Nothing`. That's why we have the `Either` type. The `Either` type takes two type arguments. The type `Either a b` has two constructors: `Left` and `Right`. Both take an argument, `Left` an argument of type `a` and `Right` an argument of type `b`.

Type	Values
`Either Int Bool`	`Left 0`, `Left 1`, `Right False`, `Right True`, ...
`Either String [Int]`	`Left "asdf"`, `Right [0,1,2]`, ...
`Either Integer Integer`	`Left 0`, `Right 0`, `Left 1`, `Right 1`, ...

Here's a simple example: a `readInt` function that only knows a couple of numbers and returns a descriptive error for the rest. Note the Haskell convention of using `Left` for errors and `Right` for success.

```
readInt :: String -> Either String Int
readInt "0" = Right 0
```

```
readInt "1" = Right 1
readInt s = Left ("Unsupported string: " ++ s)
```

Sidenote: the constructors of `Either` are called `Left` and `Right` because they refer to the left and right type arguments of `Either`. Note how in `Either a b`, `a` is the left argument and `b` is the right argument. Thus `Left` contains a value of type `a` and likewise `Right` of type `b`. The convention of using `Right` for success is probably simply because right also means correct. No offense is intended to left-handed people.

Here's another example: pattern matching an `Either`. Just like with `Maybe`, there are two patterns for an `Either`, one for each constructor.

```
iWantAString :: Either Int String -> String
iWantAString (Right str)    = str
iWantAString (Left number) = show number
```

As you recall, Haskell lists can only contain elements of the same type. You can't have a value like `[1,"foo",2]`. However, you can use a type like `Either` to represent lists that can contain two different types of values. For example we could track the number of people on a lecture, with a possibility of adding an explanation if a value is missing:

```
lectureParticipants :: [Either String Int]
lectureParticipants = [Right 10, Right 13, Left "easter vacation", Righ
```

2.9 The case-of Expression

We've seen pattern matching in function arguments, but there's also a way to pattern match in an expression. It looks like this:

```
case <value> of <pattern> -> <expression>
                <pattern> -> <expression>
```

As an example let's rewrite the `describe` example from the first lecture using `case`:

```
describe :: Integer -> String
describe 0 = "zero"
describe 1 = "one"
```

```
describe 2 = "an even prime"
describe n = "the number " ++ show n
```

```
describe :: Integer -> String
describe n = case n of 0 -> "zero"
                       1 -> "one"
                       2 -> "an even prime"
                       n -> "the number " ++ show n
```

A more interesting example is when the value we're pattern matching on is not a function argument. For example:

```
-- parse country code into country name, returns Nothing if code not
        recognized
parseCountry :: String -> Maybe String
parseCountry "FI" = Just "Finland"
parseCountry "SE" = Just "Sweden"
parseCountry _ = Nothing

flyTo :: String -> String
flyTo countryCode = case parseCountry countryCode of Just country ->
        "You're flying to " ++ country
                                                     Nothing ->
        "You're not flying anywhere"
```

```
Prelude> flyTo "FI"
"You're flying to Finland"
Prelude> flyTo "DE"
"You're not flying anywhere"
```

We could write the `flyTo` function using a helper function for pattern matching instead of using the case-of expression:

```
flyTo :: String -> String
flyTo countryCode = handleResult (parseCountry countryCode)
  where handleResult (Just country) = "You're flying to " ++ country
        handleResult Nothing        = "You're not flying anywhere"
```

In fact, a case-of expression can always be replaced with a helper function. Here's one more example, written in both ways:

```haskell
-- given a sentence, decide whether it is a statement, question or
        exclamation
sentenceType :: String -> String
sentenceType sentence = case last sentence of '.' -> "statement"
                                              '?' -> "question"
                                              '!' -> "exclamation"
                                              _   -> "not a
        sentence"
```

```haskell
-- same function, helper function instead of case-of
sentenceType sentence = classify (last sentence)
  where classify '.' = "statement"
        classify '?' = "question"
        classify '!' = "exclamation"
        classify _   = "not a sentence"
```

```
Prelude> sentenceType "This is Haskell."
"statement"
Prelude> sentenceType "This is Haskell!"
"exclamation"
```

2.9.1 When to Use Case Expressions

You might be asking, what is the point of having another pattern matching syntax. Well, `case` expressions have some advantages over equations which we'll discuss next.

Firstly, and perhaps most importantly, `case` expressions enable us to pattern match against function outputs. We might want to write early morning motivational messages to working (lazy) Haskellers:

```haskell
motivate :: String -> String
motivate "Monday"    = "Have a nice week at work!"
motivate "Tuesday"   = "You're one day closer to weekend!"
motivate "Wednesday" = "3 more day(s) until the weekend!"
motivate "Thursday"  = "2 more day(s) until the weekend!"
motivate "Friday"    = "1 more day(s) until the weekend!"
motivate _           = "Relax! You don't need to work today!"
```

Using a `case` expression we can run a helper function against the argument and pattern match on the result:

```haskell
motivate :: String -> String
motivate day = case distanceToSunday day of
  6 -> "Have a nice week at work!"
  5 -> "You're one day closer to weekend!"
  n -> if n > 1
       then show (n - 1) ++ " more day(s) until the weekend!"
       else "Relax! You don't need to work today!"
```

By the way, there's also a third way, guards:

```haskell
motivate :: String -> String
motivate day
  | n == 6 = "Have a nice week at work!"
  | n == 5 = "You're one day closer to weekend!"
  | n > 1 = show (n - 1) ++ " more day(s) until the weekend!"
  | otherwise = "Relax! You don't need to work today!"
  where n = distanceToSunday day
```

We'll see in a moment how we can define `distanceToSunday` using equations and `case` expressions.

Secondly, if a helper function needs to be shared among many patterns, then equations don't work. For example:

```haskell
area :: String -> Double -> Double
area "square" x = square x
area "circle" x = pi * square x
  where square x = x * x
```

This won't compile because a the `where` clause only appends to the `"circle"` case, so the `square` helper function is not available in the `"square"` case. On the other hand, we can write

```haskell
area :: String -> Double -> Double
area shape x = case shape of
  "square" -> square x
  "circle" -> pi * square x
  where square x = x*x
```

Thirdly, `case` expressions may help to write more concise code in a situation where a (long) function name would have to be repeated multiple times using equations. As

we saw above, we might need a function which measures the distance between a given day and Sunday:

```
distanceToSunday :: String -> Int
distanceToSunday "Monday"    = 6
distanceToSunday "Tuesday"   = 5
distanceToSunday "Wednesday" = 4
distanceToSunday "Thursday"  = 3
distanceToSunday "Friday"    = 2
distanceToSunday "Saturday"  = 1
distanceToSunday "Sunday"    = 0
```

Using a `case` expression leads into much more concise implementation:

```
distanceToSunday :: String -> Int
distanceToSunday d = case d of
  "Monday"    -> 6
  "Tuesday"   -> 5
  "Wednesday" -> 4
  "Thursday"  -> 3
  "Friday"    -> 2
  "Saturday"  -> 1
  "Sunday"    -> 0
```

These three benefits make the `case` expression a versatile tool in a Haskeller's toolbox. It's worth remembering how `case` works.

(Representing weekdays as strings may get the job done, but it's not the perfect solution. What happens if we apply `motivate` to `"monday"` (with all letters in lower case) or `"keskiviikko"`? In Lecture 5, we will learn a better way to represent things like weekdays.)

2.10 Recap: Pattern Matching

Things you can use as patterns:

- `Int` and `Integer` constants like `(-1)`, `0`, `1`, `2`, ...
- `Bool` values `True` and `False`
- `Char` constants: `'a'`, `'b'`
- `String` constants: `"abc"`, `""`
- `Maybe` constructors: `Nothing`, `(Just x)`
- `Either` constructors: `(Left x)`, `(Right y)`
- The special _ pattern which means "anything, I don't care"

- Combinations of these patterns, like for example `(Just 1)`
- We' ll learn about other patterns, for example lists, in the next lectures.

Places where you can use patterns:

- Defining a function with equations:

```
f :: Bool -> Maybe Int -> Int
f False Nothing  = 1
f False _        = 2
f True  (Just i) = i
f True  Nothing  = 0
```

- In the `case of` expression:

```
case number of 0 -> "zero"
               1 -> "one"
               _ -> "not zero or one"
```

The only thing you really *need* pattern matching for is *getting the value* inside a `Just`, `Left` or `Right` constructor. Here are two more examples of this:

```
-- getElement (Just i) gets the ith element (counting from zero) of
         a list, getElement Nothing gets the last element
getElement :: Maybe Int -> [a] -> a
getElement (Just i) xs = xs !! i
getElement Nothing xs = last xs
```

```
Prelude> getElement Nothing "hurray!"
'!'
Prelude> getElement (Just 3) [5,6,7,8,9]
8
```

```
direction :: Either Int Int -> String
direction (Left i) = "you should go left " ++ show i ++ " meters!"
direction (Right i) = "you should go right " ++ show i ++ " meters!"
```

```
Prelude> direction (Left 3)
"you should go left 3 meters!"
```

```
Prelude> direction (Right 5)
"you should go right 5 meters!"
```

Other uses (that we've seen so far!) of pattern matching can also be accomplished with the == operator. However, things like `x==Nothing` won't work in all cases. We'll find out why when we talk about type classes in lecture 4.

2.11 Quiz

How many values does `f x = [x,x]` return?

a. Zero
b. One
c. Two

Why does the expression `Nothing 1` cause a type error?

a. Because `Nothing` takes no arguments
b. Because `Nothing` returns nothing
c. Because `Nothing` is a constructor

What is the type of the function
`f x y = if x && y then Right x else Left "foo"`?

a. `Bool -> Bool -> Either Bool String`
b. `String -> String -> Either String String`
c. `Bool -> Bool -> Either String Bool`

Which of the following functions could have the type `Bool -> Int -> [Bool]`

a. `f x y = [0, y]`
b. `f x y = [x, True]`
c. `f x y = [y, True]`

What is the type of this function? `justBoth a b = [Just a, Just b]`

a. `a -> b -> [Maybe a, Maybe b]`
b. `a -> a -> [Just a]`
c. `a -> b -> [Maybe a]`
d. `a -> a -> [Maybe a]`

2.12 Exercises

- Set2a

- Set2b

3 Lecture 3: Catamorphic

- Lists, lists, lists
- Functionality
- A bit about types

3.1 Functional Programming, at Last

Now with lists and polymorphism in our toolbox, we can finally start to look at functional programming.

In Haskell a function is a value, just like a number or a list is. Functions can be passed as parameters to other functions. Here's a toy example. The function `applyTo1` takes a function of type `Int->Int`, applies it to the number `1`, and returns the result.

```
applyTo1 :: (Int -> Int) -> Int
applyTo1 f = f 1
```

Let's define a simple function of type `Int->Int` and see `applyTo1` in action.

```
addThree :: Int -> Int
addThree x = x + 3
```

```
applyTo1 addThree
  ==> addThree 1
  ==> 1 + 3
  ==> 4
```

Let's go back to the type annotation for `applyTo1`.

```
applyTo1 :: (Int -> Int) -> Int
```

The parentheses are needed because the type `Int -> Int -> Int` would be the type of a function taking two `Int` arguments. More on this later.

Let's look at a slightly more interesting example. This time we'll implement a polymorphic function `doTwice`. Note how we can use it with various types of values and functions.

```
doTwice :: (a -> a) -> a -> a
doTwice f x = f (f x)
```

```
doTwice addThree 1
  ==> addThree (addThree 1)
  ==> 7
doTwice tail "abcd"
  ==> tail (tail "abcd")
  ==> "cd"
```

```
makeCool :: String -> String
makeCool str = "WOW " ++ str ++ "!"
```

```
doTwice makeCool "Haskell"
  ==> "WOW WOW Haskell!!"
```

3.1.1 Functional Programming on Lists

That was a bit boring. Luckily there are many useful list functions that take functions as arguments. By the way, functions that take functions as arguments (or return functions) are often called *higher-order functions*.

The most famous of these list-processing higher-order functions is `map`. It gives you a new list by applying the given function to all elements of a list.

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

```
map addThree [1,2,3]
  ==> [4,5,6]
```

The partner in crime for `map` is `filter`. Instead of transforming all elements of a list, `filter` drops some elements of a list and keeps others. In other words, `filter` selects the elements from a list that fulfill a condition.

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

Here’s an example: selecting the positive elements from a list

```
positive :: Int -> Bool
positive x = x>0
```

```
filter positive [0,1,-1,3,-3]
  ==> [1,3]
```

Note how both the type signatures of `map` and `filter` use polymorphism. They work on all kinds of lists. The type of `map` even uses two type parameters! Here are some examples of type inference using `map` and `filter`.

```
onlyPositive xs = filter positive xs
mapBooleans f = map f [False,True]
```

```
Prelude> :t onlyPositive
onlyPositive :: [Int] -> [Int]
Prelude> :t mapBooleans
mapBooleans :: (Bool -> b) -> [b]
Prelude> :t mapBooleans not
mapBooleans not :: [Bool]
```

One more thing: remember how constructors were just functions? That means you can pass them as arguments to other functions!

```
wrapJust xs = map Just xs
```

```
Prelude> :t wrapJust
wrapJust :: [a] -> [Maybe a]
```

```
Prelude> wrapJust [1,2,3]
[Just 1,Just 2,Just 3]
```

3.1.2 Examples of Functional Programming on Lists

How many "palindrome numbers" are between 1 and n?

```
-- a predicate that checks if a string is a palindrome
palindrome :: String -> Bool
palindrome str = str == reverse str

-- palindromes n takes all numbers from 1 to n, converts them to
         strings using show, and keeps only palindromes
palindromes :: Int -> [String]
palindromes n = filter palindrome (map show [1..n])
```

```
palindrome "1331" ==> True
palindromes 150 ==>
  ["1","2","3","4","5","6","7","8","9",
   "11","22","33","44","55","66","77","88","99",
   "101","111","121","131","141"]
length (palindromes 9999) ==> 198
```

How many words in a string start with "a"? This uses the function `words` from the module `Data.List` that splits a string into words.

```
countAWords :: String -> Int
countAWords string = length (filter startsWithA (words string))
  where startsWithA s = head s == 'a'
```

```
countAWords "does anyone want an apple?"
  ==> 3
```

The function `tails` from `Data.List` returns the list of all suffixes ("tails") of a list. We can use `tails` for many string processing tasks. Here's how `tails` works:

```
tails "echo"
  ==> ["echo","cho","ho","o",""]
```

Here's an example where we find what characters come after a given character in a string. First of all, we use `tails`, `map` and `take` to get all substrings of a certain length:

```
substringsOfLength :: Int -> String -> [String]
substringsOfLength n string = map shorten (tails string)
  where shorten s = take n s
```

```
substringsOfLength 3 "hello"
  ==> ["hel","ell","llo","lo","o",""]
```

There's some shorter substrings left at the end (can you see why?), but they're fine for our purposes right now. Now that we have `substringsOfLength`, we can implement the function `whatFollows c k s` that finds all the occurrences of the character `c` in the string `s`, and outputs the `k` letters that come after these occurrences.

```
whatFollows :: Char -> Int -> String -> [String]
whatFollows c k string = map tail (filter match (substringsOfLength
         (k+1) string))
  where match sub = take 1 sub == [c]
```

```
whatFollows 'a' 2 "abracadabra"
  ==> ["br","ca","da","br",""]
```

3.2 Partial Application

When using higher-order functions you can find yourself defining lots of small helper functions, like `addThree` or `shorten` in the previous examples. This is a bit of a chore in the long run, but luckily Haskell's functions behave a bit weirdly...

Let's start in GHCi:

```
Prelude> add a b = a+b
Prelude> add 1 5
6
Prelude> addThree = add 3
```

```
Prelude> addThree 2
5
```

So, we've defined `add`, a function of two arguments, and only given it one argument. The result is not a type error but a new function. The new function just stores (or remembers) the given argument, waits for another argument, and then gives both to `add`.

```
Prelude> map addThree [1,2,3]
[4,5,6]
Prelude> map (add 3) [1,2,3]
[4,5,6]
```

Here we can see that we don't even need to give a name to the function returned by `add 3`. We can just use it anywhere where a function of one argument is expected.

This is called *partial application*. All functions in Haskell behave like this. Let's have a closer look. Here's a function that takes many arguments.

```
between :: Integer -> Integer -> Integer -> Bool
between lo high x = x < high && x > lo
```

```
Prelude> between 3 7 5
True
Prelude> between 3 6 8
False
```

We can give `between` less arguments and get back new functions, just like we saw with `add`:

```
Prelude> (between 1 5) 2
True
Prelude> let f = between 1 5 in f 2
True
Prelude> map (between 1 3) [1,2,3]
[False,True,False]
```

Look at the types of partially applying `between`. They behave neatly, with arguments disappearing one by one from the type as values are added to the expression.

```
Prelude> :t between
between :: Integer -> Integer -> Integer -> Bool
Prelude> :t between 1
between 1 :: Integer -> Integer -> Bool
Prelude> :t between 1 2
between 1 2 :: Integer -> Bool
Prelude> :t between 1 2 3
between 1 2 3 :: Bool
```

Actually, when we write a type like `Integer -> Integer -> Integer -> Bool`, it means `Integer -> (Integer -> (Integer -> Bool))`. That is, a multi-argument function is just a function that returns a function. Similarly, an expression like `between 1 2 3` is the same as `((between 1) 2) 3`, so passing multiple arguments to a function happens via multiple single-argument calls. Representing multi-argument functions like this is called *currying* (after the logician Haskell Curry). Currying is what makes partial application possible.

Here's another example of using partial application with `map`:

```
map (drop 1) ["Hello","World!"]
  ==> ["ello","orld!"]
```

In addition to normal functions, partial application also works with operators. With operators you can choose whether you apply the left or the right argument. (Partially applied operators are also called *sections* or *operator sections*). Some examples:

```
Prelude> map (*2) [1,2,3]
[2,4,6]
Prelude> map (2*) [1,2,3]
[2,4,6]
Prelude> map (1/) [1,2,3,4,5]
[1.0,0.5,0.3333333333333333,0.25,0.2]
```

3.3 Prefix and Infix Notations

Normal Haskell operators are applied with *prefix notation*, which is just a fancy way to say that the function name comes before the arguments. In contrast, operators are applied with *infix notation* – the name of the function comes between the arguments.

An infix operator can be converted into a prefix function by adding parentheses around it. For instance,

```
(+) 1 2 ==> 1 + 2 ==> 3
```

This is useful especially when an operator needs to be passed as an argument to another function.

As an example, the function `zipWith` takes two lists, a binary function, and joins the lists using the function. We can use `zipWith (+)` to sum two lists, element-by-element:

```
Prelude> :t zipWith
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
Prelude> zipWith (+) [0,2,5] [1,3,3]
[1,5,8]
```

Without the ability to turn an operator into a function, we'd have to use a helper function – such as `add` above.

Note that omitting the parentheses leads into a type error:

```
Prelude> zipWith + [0,2,5,3] [1,3,3]

<interactive>:1:11: error:
    • Couldn't match expected type ‘[Integer]
                                    -> (a -> b -> c) -> [a] -> [b] -
      > [c]’
                  with actual type ‘[Integer]’
    • The function ‘[0, 2, 5, 3]’ is applied to one argument,
      but its type ‘[Integer]’ has none
      In the second argument of ‘(+)’, namely ‘[0, 2, 5, 3] [1, 3,
      3]’
      In the expression: zipWith + [0, 2, 5, 3] [1, 3, 3]
    • Relevant bindings include
        it :: (a -> b -> c) -> [a] -> [b] -> [c]
          (bound at <interactive>:1:1)
```

The reason for this weird-looking error is that GHCi got confused and thought that we were somehow trying to add `zipWith` and `[0,2,5,3] [1,3,3]` together. Logically, it deduced that `[0,2,5,3]` must be a function since it's being applied to `[1,3,3]` (remember that functions bind tighter than operators).

Unfortunately, error messages can sometimes be obscure, since the compiler cannot always know the "real" cause of the error (which is in this case was omitting the parentheses). Weird error messages are frustrating, but only the programmer knows what was the original intent behind the code.

Another nice feature of Haskell is the syntax for applying a binary function as if it was an infix operator, by surrounding it with backticks (`). For example:

```
6 `div` 2 ==> div 6 2 ==> 3
(+1) `map` [1,2,3] ==> map (+1) [1,2,3] ==> [2,3,4]
```

3.4 Lambdas

The last spanner we need in our functional programming toolbox is λ (lambda). Lambda expressions are *anonymous functions*. Consider a situation where you need a function only once, for example in an expression like

```
let big x = x>7 in filter big [1,10,100]
```

A lambda expression allows us to write this directly, without defining a name (`big`) for the helper function:

```
filter (\x -> x>7) [1,10,100]
```

Here are some more examples in GHCi:

```
Prelude> (\x -> x*x) 3
9
Prelude> (\x -> reverse x == x) "ABBA"
True
Prelude> filter (\x -> reverse x == x)
         ["ABBA","ACDC","otto","lothar","anna"]
["ABBA","otto","anna"]
Prelude> (\x y -> x^2+y^2) 2 3           -- multiple arguments
13
```

The Haskell syntax for lambdas is a bit surprising. The backslash character (`\`) stands for the greek letter lambda (λ). The Haskell expression `\x -> x+1` is trying to mimic the mathematical notation *λx. x+1*. Other languages use syntax like `x => x+1` (JavaScript) or `lambda x: x+1` (Python).

Note! You never *need* to use a lambda expression. You can always instead define the function normally using `let` or `where`.

By the way, lambda expressions are quite powerful constructs which have a deep theory of their own, known as Lambda calculus. Some even consider purely functional programming languages such as Haskell to be typed extensions of Lambda calculus with extra syntax.

3.5 Sidenote: The `.` and `$` Operators

The two most common operators in Haskell codebases are probably `.` and `$`. They are useful when writing code that uses higher-order functions. The first of these, the `.` operator, is the *function composition* operator. Here's its type

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

And here's what it does

```
(f.g) x ==> f (g x)
```

You can use function composition to build functions out of other functions, without mentioning any arguments. For example:

```
double x = 2*x
quadruple = double . double  -- computes 2*(2*x) == 4*x
f = quadruple . (+1)         -- computes 4*(x+1)
g = (+1) . quadruple         -- computes 4*x+1
third = head . tail . tail   -- fetches the third element of a list
```

We can also reimplement `doTwice` using `(.)`. Note how we can use `doTwice` both as applied only to a function, or as applied to a function and a value.

```
doTwice :: (a -> a) -> a -> a
doTwice f = f . f
```

```
let ttail = doTwice tail
in ttail [1,2,3,4]
  ==> [3,4]
```

```
(doTwice tail) [1,2,3,4] ==> [3,4]

doTwice tail [1,2,3,4] ==> [3,4]
```

Often function composition is not used when defining a new function, but instead to avoid defining a helper function. For instance, consider the difference between these two expressions:

```
let notEmpty x = not (null x)
in filter notEmpty [[1,2,3],[],[4]]
  ==> [[1,2,3],[4]]



filter (not . null) [[1,2,3],[],[4]]
  ==> [[1,2,3],[4]]
```

The other operator, `$` is more subtle. Let's look at its type.

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

It takes a function of type `a -> b` and a value of type `a`, and returns a value of type `b`. In other words, it's a function application operator. The expression `f $ x` is the same as `f x`. This seems pretty useless, but it means that the `$` operator can be used to eliminate parentheses! These expressions are the same:

```
head (reverse "abcd")
head $ reverse "abcd"
```

This isn't that impressive when it's used to eliminate one pair of parentheses, but together `.` and `$` can eliminate lots of them! For example, we can rewrite

```
reverse (map head (map reverse (["Haskell","pro"] ++
         ["dodo","lyric"])))
```

as

```
(reverse . map head . map reverse) (["Haskell","pro"] ++
         ["dodo","lyric"])
```

and then

```
reverse . map head . map reverse $ ["Haskell","pro"] ++
         ["dodo","lyric"]
```

Sometimes the operators `.` and `$` are useful as functions in their own right. For example, a list of functions can be applied to an argument using map and a section of `$`:

```
map ($"string") [reverse, take 2, drop 2]
  ==> [reverse $ "string", take 2 $ "string", drop 2 $ "string"]
  ==> [reverse "string", take 2 "string", drop 2 "string"]
  ==> ["gnirts", "st", "ring"]
```

If this seems complicated, don't worry. You don't need to use `.` and `$` in your own code until you're comfortable with them. However, you'll bump into `.` and `$` when reading Haskell examples and code on the internet, so it's good to know about them. This article might also help.

3.6 Example: Rewriting `whatFollows`

Now, let's rewrite the `whatFollows` example from earlier using the tools we just saw. Here's the original version:

```
substringsOfLength :: Int -> String -> [String]
substringsOfLength n string = map shorten (tails string)
  where shorten s = take n s

whatFollows :: Char -> Int -> String -> [String]
whatFollows c k string = map tail (filter match (substringsOfLength
         (k+1) string))
  where match sub = take 1 sub == [c]
```

To get started, let's get rid of the helper function `substringsOfLength` and move all the code to `whatFollows`:

```
whatFollows c k string = map tail (filter match (map shorten (tails
         string)))
  where shorten s = take (k+1) s
        match sub = take 1 sub == [c]
```

Now let's use partial application instead of defining `shorten`:

```
whatFollows c k string = map tail (filter match (map (take (k+1))
         (tails string)))
  where match sub = take 1 sub == [c]
```

Let's use `.` and `$` to eliminate some of those parentheses:

```
whatFollows c k string = map tail . filter match . map (take (k+1))
         $ tails string
  where match sub = take 1 sub == [c]
```

We can also replace `match` with a lambda:

```
whatFollows c k string = map tail . filter (\sub -> take 1 sub ==
         [c]) . map (take (k+1)) $ tails string
```

Finally, we don't need to mention the `string` parameter at all, since we can just express `whatFollows` as a composition of `map`, `filter`, `map` and `tails`:

```
whatFollows c k = map tail . filter (\sub -> take 1 sub == [c]) .
         map (take (k+1)) . tails
```

We can even go a bit further by rewriting the lambda using an operator section

```
    \sub -> take 1 sub == [c]
=== \sub -> (==[c]) (take 1 sub)
=== \sub -> (==[c]) ((take 1) sub)
=== \sub -> ((==[c]) . (take 1)) sub
=== ((==[c]) . (take 1))
=== ((==[c]) . take 1)
```

Now what we have left is:

```
whatFollows c k = map tail . filter ((==[c]) . take 1) . map (take
        (k+1)) . tails
```

This is a somewhat extreme version of the function, but when used in moderation the techniques shown here can make code easier to read.

3.7 More Functional List Wrangling Examples

Here are some more examples of functional programming with lists. Let's start by introducing a couple of new list functions:

```
takeWhile :: (a -> Bool) -> [a] -> [a]   -- take elements from a
        list as long as they satisfy a predicate
dropWhile :: (a -> Bool) -> [a] -> [a]   -- drop elements from a
        list as long as they satisfy a predicate
```

```
takeWhile even [2,4,1,2,3]   ==> [2,4]
dropWhile even [2,4,1,2,3]   ==> [1,2,3]
```

There's also the function `elem`, which can be used to check if a list contains an element:

```
elem 3 [1,2,3]   ==> True
elem 4 [1,2,3]   ==> False
```

Using these, we can implement a function `findSubstring` that finds the earliest and longest substring in a string that consist only of the given characters.

```
findSubstring :: String -> String -> String
findSubstring chars = takeWhile (\x -> elem x chars)
                    . dropWhile (\x -> not $ elem x chars)
```

```
findSubstring "a" "bbaabaaaab"              ==> "aa"
findSubstring "abcd" "xxxyyyzabaaxxabcd"    ==> "abaa"
```

The function `zipWith` lets you combine two lists element-by-element:

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

```
zipWith (++) ["John","Mary"] ["Smith","Cooper"]
  ==> ["JohnSmith","MaryCooper"]
zipWith take [4,3] ["Hello","Warden"]
  ==> ["Hell","War"]
```

Sometimes with higher-order functions it's useful to have a function that does nothing. The function `id :: a -> a` is the identity function and just returns its argument.

```
id 3 ==> 3
map id [1,2,3] ==> [1,2,3]
```

This seems a bit useless, but you can use it for example with `filter` or `dropWhile`:

```
filter id [True,False,True,True]  ==>  [True,True,True]
dropWhile id [True,True,False,True,False]  ==>  [False,True,False]
```

Another very simple but sometimes crucial function is the constant function, `const :: a -> b -> a`. It always returns its first argument:

```
const 3 True ==> 3
const 3 0    ==> 3
```

When partially applied it can be used when you need a function that always returns the same value:

```
map (const 5) [1,2,3,4] ==> [5,5,5,5]
filter (const True) [1,2,3,4] ==> [1,2,3,4]
```

3.8 Lists and Recursion

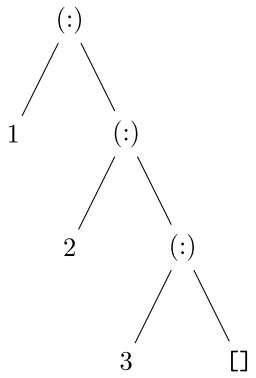
Here's a new operator, `:`

```
Prelude> 1:[]
[1]
Prelude> 1:[2,3]
[1,2,3]
Prelude> tail (1:[2,3])
[2,3]
Prelude> head (1:[2,3])
1
Prelude> :t (:)
(:) :: a -> [a] -> [a]
```

The `:` operator builds a list out of a head and a tail. In other words, `x : xs` is the same as `[x] ++ xs`. Why do we need an operator for this?

Actually, `:` is the *constructor* for lists: it returns a new linked list node. The other list constructor is `[]`, the empty list. All lists are built using `:` and `[]`. The familiar `[x,y,z]` syntax is actually just a nicer way to write `x:y:z:[]`, or even more explicitly, `x:(y:(z:[]))`. In fact `(++)` is defined in terms of `:` and recursion in the standard library.

Here's a picture of how `[1,2,3]` is structured in memory:



3.8.1 Building a List

Using `:` we can define recursive functions that build lists. For example here's a function that builds lists like `[3,2,1]`:

```
descend 0 = []
descend n = n : descend (n-1)
```

```
descend 4 ==> [4,3,2,1]
```

Here's a function that builds a list by iterating a function `n` times:

```
iterate f 0 x = [x]
iterate f n x = x : iterate f (n-1) (f x)
```

```
iterate (*2) 4 3 ==> [3,6,12,24,48]

let xs = "terve"
in iterate tail (length xs) xs
  ==> ["terve","erve","rve","ve","e",""]
```

Here's a more complicated example: splitting a string into pieces at a given character:

```
split :: Char -> String -> [String]
split c [] = []
split c xs = start : split c (drop 1 rest)
  where start = takeWhile (/=c) xs
        rest = dropWhile (/=c) xs
```

```
split 'x' "fooxxbarxquux"   ==>   ["foo","","bar","quu"]
```

3.8.2 Pattern Matching for Lists

Last lecture, it was said that constructors are things that can be pattern matched on. Above, it was divulged that the constructors for the list type are `:` and `[]`. We can put one and one together and guess that we can pattern match on `:` and `[]`. This is true! Here's how you can define your own versions of `head` and `tail` using pattern matching:

```
myhead :: [Int] -> Int
myhead [] = -1
myhead (first:rest) = first

mytail :: [Int] -> [Int]
mytail [] = []
mytail (first:rest) = rest
```

You can *nest* patterns. That is, you can pattern match more than one element from the start of a list. In this example, we use the pattern `(a:b:_)` which is the same as `(a:(b:_))`:

```
sumFirstTwo :: [Integer] -> Integer
-- this equation gets used for lists of length at least two
sumFirstTwo (a:b:_) = a+b
-- this equation gets used for all other lists (i.e. lists of length
        0 or 1)
sumFirstTwo _       = 0
```

```
sumFirstTwo [1]      ==> 0
sumFirstTwo [1,2]    ==> 3
sumFirstTwo [1,2,4]  ==> 3
```

Here's an example that uses many different list patterns:

```
describeList :: [Int] -> String
describeList []         = "an empty list"
describeList (x:[])     = "a list with one element"
describeList (x:y:[])   = "a list with two elements"
describeList (x:y:z:xs) = "a list with at least three elements"
```

```
describeList [1,3]        ==> "a list with two elements"
describeList [1,2,3,4,5]  ==> "a list with at least three elements"
```

List patterns that end with `:[]` can be typed out as list literals. That is, just like `[1,2,3]` is the same value as `1:2:3:[]`, the pattern `[x,y]` is the same as the pattern `x:y:[]`. Let's rewrite that previous example.

```
describeList :: [Int] -> String
describeList []         = "an empty list"
describeList [x]        = "a list with exactly one element"
describeList [x,y]      = "a list with exactly two elements"
describeList (x:y:z:xs) = "a list with at least three elements"
```

Another way we can nest patterns is pattern matching on the head while pattern matching on a list. For example this function checks if a list starts with `0`:

```
startsWithZero :: [Integer] -> Bool
startsWithZero (0:xs) = True
startsWithZero (x:xs) = False
startsWithZero []     = False
```

3.8.3 Consuming a List

Using pattern matching and recursion, we can recursively process a whole list. Here's how you sum all the numbers in a list:

```
sumNumbers :: [Int] -> Int
sumNumbers [] = 0
sumNumbers (x:xs) = x + sumNumbers xs
```

Here's how you compute the largest number in a list, this time using a helper function.

```
myMaximum :: [Int] -> Int
myMaximum [] = 0       -- actually this should be some sort of
         error...
myMaximum (x:xs) = go x xs
  where go biggest [] = biggest
        go biggest (x:xs) = go (max biggest x) xs
```

Note!, "go" is just a cute name for the helper function here. It's not special syntax.

It's often convenient to use nested patterns while consuming a list. Here's an example that counts how many `Nothing` values occur in a list of `Maybes`:

```
countNothings :: [Maybe a] -> Int
countNothings [] = 0
countNothings (Nothing : xs) = 1 + countNothings xs
countNothings (Just _  : xs) = countNothings xs
```

```
countNothings [Nothing,Just 1,Nothing]  ==>  2
```

3.8.4 Building and Consuming a List

Now that we can build and consume lists, let's do both of them at the same time. This function doubles all elements in a list.

```
doubleList :: [Int] -> [Int]
doubleList [] = []
doubleList (x:xs) = 2*x : doubleList xs
```

It evaluates like this:

```
doubleList [1,2,3]
=== doubleList (1:(2:(3:[])))
==> 2*1 : doubleList (2:(3:[]))
==> 2*1 : (2*2 : doubleList (3:[]))
==> 2*1 : (2*2 : (2*3 : doubleList []))
==> 2*1 : (2*2 : (2*3 : []))
=== [2*1, 2*2, 2*3]
==> [2,4,6]
```

Once you know pattern matching for lists, it's straightforward to define `map` and `filter`. Actually, let's just look at the GHC standard library implementations. Here's map:

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

and here's filter:

```
filter :: (a -> Bool) -> [a] -> [a]
filter _pred []    = []
filter pred (x:xs)
  | pred x         = x : filter pred xs
  | otherwise      = filter pred xs
```

(**Note!** Naming the argument `_pred` is a way to tell the reader of the code that this argument is unused. It could have been just _ as well.)

3.8.5 Tail Recursion and Lists

When a recursive function evaluates to a new call to that same function with different arguments, it is called *tail-recursive*. (The recursive call is said to be in *tail*

position.) This is the type of recursion that corresponds to an imperative loop. We've already seen many examples of tail-recursive functions, but we haven't really contrasted the two ways for writing the same function. This is `sumNumbers` from earlier in this lecture:

```
-- Not tail recursive!
sumNumbers :: [Int] -> Int
sumNumbers [] = 0
sumNumbers (x:xs) = x + sumNumbers xs
```

In the second equation the function + is at the top level, i.e. in tail position. The recursive call to `sumNumbers` is an argument of +. This is `sumNumbers` written using a tail recursive helper function:

```
-- Tail recursive version
sumNumbers :: [Int] -> Int
sumNumbers xs = go 0 xs
  where go sum [] = sum
        go sum (x:xs) = go (sum+x) xs
```

Note the second equation of `go`: it has the recursive call to `go` at the top level, i.e. in tail position. The + is now in an argument to `go`.

For a function like `sumNumbers` that produces a single value (a number), it doesn't really matter which form of recursion you choose. The non-tail-recursive function is easier to read, while the tail-recursive one can be easier to come up with. You can try writing a function both ways. The tail-recursive form might be more efficient, but that depends on many details. We'll talk more about Haskell performance in part 2 of this course.

However, when you're returning a list there is a big difference between these two forms. Consider the function `doubleList` from earlier. Here it is again, implemented first directly, and then via a tail-recursive helper function.

```
-- Not tail recursive!
doubleList :: [Int] -> [Int]
doubleList [] = []
doubleList (x:xs) = 2*x : doubleList xs
```

```
-- Tail recursive version
doubleList :: [Int] -> [Int]
doubleList xs = go [] xs
```

```
    where go result [] = result
          go result (x:xs) = go (result++[2*x]) xs
```

Here the direct version is much more efficient. The `(:)` operator works in constant time, whereas the `(++)` operator needs to walk the whole list, needing linear time. Thus the direct version uses linear time (*O(n)*) with respect to the length of the list, while the tail-recursive version is quadratic (*O(n²)*)!

One might be tempted to fix this by using `(:)` in the tail-recursive version, but then the list would get generated in the reverse order. This could be fixed with an application of `reverse`, but that would make the resulting function quite complicated.

There is another reason to prefer the direct version: laziness. We'll get back to laziness in part 2 of the course, but for now it's enough for you to know that **the direct way of generating a list is simpler, more efficient and more idiomatic**. You should try to practice it in the exercises. Check out the standard library implementations of `map` and `filter` above, even they produce the list directly without tail recursion!

3.9 Something Fun: List Comprehensions

Haskell has *list comprehensions*, a nice syntax for defining lists that combines the power of `map` and `filter`. You might be familiar with Python's list comprehensions already. Haskell's work pretty much the same way, but their syntax is a bit different.

Mapping:

```
[2*i | i<-[1,2,3]]
  ==> [2,4,6]
```

Filtering:

```
[i | i <- [1..7], even i]
  ==> [2,4,6]
```

In general, these two forms are equivalent:

```
[f x | x <- lis, p x]
map f (filter p lis)
```

List comprehensions can do even more. You can iterate over multiple lists:

```
[ first ++ " " ++ last | first <- ["John", "Mary"], last <-
        ["Smith","Cooper"] ]
  ==> ["John Smith","John Cooper","Mary Smith","Mary Cooper"]
```

You can make local definitions:

```
[ reversed | word <- ["this","is","a","string"], let reversed =
        reverse word ]
  ==> ["siht","si","a","gnirts"]
```

You can even do pattern matching in list comprehensions!

```
firstLetters string = [ char | (char:_) <- words string ]
```

```
firstLetters "Hello World!"
  ==> "HW"
```

3.10 Something Fun: Custom Operators

In Haskell an *operator* is anything built from the characters `!#$%&*+./<=>?@\^|-~`. Operators can be defined just like functions (note the slightly different type annotation):

```
(<+>) :: [Int] -> [Int] -> [Int]
xs <+> ys = zipWith (+) xs ys
```

```
(+++) :: String -> String -> String
a +++ b = a ++ " " ++ b
```

3.11 Something Useful: Typed Holes

Sometimes when writing Haskell it can be tricky to find expressions that have the right type. Luckily, the compiler can help you here! A feature called *Typed Holes* lets

you leave blanks in your code, and the compiler will tell you what type the expression in the blank should have.

Blanks can look like _ or `_name`. They can be confused with the "anything goes" pattern _, but the difference is that a hole occurs on the *right side* of a =, while an anything goes pattern occurs on the *left side* of a =.

Let's start with a simple example in GHCi:

```
Prelude> filter _hole [True,False]

<interactive>: error:
    • Found hole: _hole :: Bool -> Bool
      Or perhaps ‘_hole’ is mis-spelled, or not in scope
    • In the first argument of ‘filter’, namely ‘_hole’
      In the expression: filter _hole [True, False]
      In an equation for ‘it’: it = filter _hole [True, False]
    • Relevant bindings include
        it :: [Bool] (bound at <interactive>:5:1)
      Valid hole fits include
        not :: Bool -> Bool
          (imported from ‘Prelude’
           (and originally defined in ‘ghc-prim-0.6.1:GHC.Classes’))
        id :: forall a. a -> a
          with id @Bool
          (imported from ‘Prelude’ (and originally defined in
         ‘GHC.Base’))
```

The important part of this message is the very first line. This tells you what type Haskell is expecting for the hole.

```
<interactive>: error:
    • Found hole: _hole :: Bool -> Bool
```

The rest of the error message offers some suggestions for the value of `_hole`, for example `id` and `not`.

Let's look at a longer example, where we try to implement a function that filters a list using a list of booleans:

```
keepElements [5,6,7,8] [True,False,True,False] ==> [5,7]
```

We'll start with `zip` since we know that pairs up the elements of the two lists nicely. We add a typed hole `_doIt` and call it with the result of `zip` to see what we need to

do next.

```
keepElements :: [a] -> [Bool] -> [a]
keepElements xs bs = _doIt (zip xs bs)
```

```
<interactive>: error:
    • Found hole: _doIt :: [(a, Bool)] -> [a]
    ...
```

That looks like something that could be done with `map`. Let's see what happens:

```
keepElements :: [a] -> [Bool] -> [a]
keepElements xs bs = map _f (zip xs bs)
```

```
<interactive>: error:
    • Found hole: _f :: (a, Bool) -> a
    ...
      Valid hole fits include
        fst :: forall a b. (a, b) -> a
```

Great! GHC reminded us of the function `fst` that grabs the first out of a pair. Are we done now?

```
keepElements :: [a] -> [Bool] -> [a]
keepElements xs bs = map fst (zip xs bs)
```

```
Prelude> keepElements [5,6,7,8] [True,False,True,False]
[5,6,7,8]
```

Oh right, we've forgotten to do the filtering part. Let's try a typed hole again:

```
keepElements :: [a] -> [Bool] -> [a]
keepElements xs bs = map fst (filter _predicate (zip xs bs))
```

```
<interactive>: error:
    • Found hole: _predicate :: (a, Bool) -> Bool
    ...
      Valid hole fits include
        snd :: forall a b. (a, b) -> b
        ...
        ... lots of other suggestions
```

Again GHC has reminded us of a function that seems to do the right thing: just grab the second element out of the tuple. Now our function is finished and works as expected.

```
keepElements :: [a] -> [Bool] -> [a]
keepElements xs bs = map fst (filter snd (zip xs bs))
```

```
Prelude> keepElements [5,6,7,8] [True,False,True,False]
[5,7]
```

Remember typed holes when you get stuck with type errors when working on the exercises! Try replacing a function or variable with a typed hole. It might help you figure out what you need.

3.12 Quiz

What's the type of this function? `both p q x = p x && q x`

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

What's the (most general) type of this function? `applyInOut f g x = f (g (f x))`

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

Which one of the following functions adds its first argument to the second?

a. `f x x = x + x`
b. `f x = \y -> x + y`
c. `f = \x y -> x + x`

Which one of the following functions does not satisfy `f 1 ==> 1`?

a. `f x = (\y -> y) x`
b. `f x = \y -> y`
c. `f x = (\y -> x) x`

Which one of the following functions is correctly typed?

a. `f x y = not x; f :: (Bool -> Bool) -> Bool`
b. `f x = x ++ "a"; f :: Char -> String`
c. `f x = 'a' : x; f :: String -> String`

How many arguments does `drop 2` take?

a. Zero
b. One
c. Two

What does this function do? `f (_:x:_) = x`

a. Returns the first element of a list
b. Returns an arbitrary element of a list
c. Returns all except the first and last elements of a list
d. Returns the second element of a list

What is the result of `reverse $ take 5 . tail $ "This is a test"`?

a. `"i sih"`
b. `"set a"`
c. A type error

If `f :: a -> b`, then what is the type of `map (.f)`?

a. `[b -> c] -> [a -> c]`
b. `[c -> a] -> [c -> b]`
c. `(b -> c) -> [a -> c]`
d. `[a] -> [b]`

What is the type of the leftmost `id` in `id id`?

a. unspecified
b. `a`
c. `a -> a`
d. `(a -> a) -> (a -> a)`

What is the type of `const const`?

a. unspecified
b. `(c -> a -> b) -> a`
c. `c -> (a -> b -> a)`
d. `a -> b -> c -> a`

3.13 Exercises

- Set3a: normal list exercises
- Set3b: list recursion exercises

3.13.1 Common Errors

```
No instance for (Eq a) arising from a use of ‘==’
```

You’ ve probably tried to use `x==Nothing` to check if a value is `Nothing`. Use pattern matching instead. The reason for this error is that values of type `Maybe a` can’ t be compared because Haskell doesn’ t know how to compare values of the polymorphic type `a`. You’ ll find more about this in the next lecture. Use pattern matching instead of `==` for now.

4 Lecture 4: Real Classy

- Tuples
- Type classes
- Data structures: Map, Array

4.1 Sidenote: Tuples

Before we dive into type classes, let’ s introduce the last remaining built-in datatype in Haskell: the tuple. *Tuples* or *pairs* (or triples, quadruples, etc) are a way of bundling a couple of values of different types together. You can think of tuples as fixed-length lists (just like Python’ s tuples). Unlike lists, each element in the tuple can have a different type. The types of the elements are reflected in the type of the tuple. Here are some examples of tuple types and values:

Type	**Example value**
`(String,String)`	`("Hello","World!")`
`(Int,Bool)`	`(1,True)`

Type	Example value
`(Int,Int,Int)`	`(4,0,3)`

To get values out of tuples, you can use the functions `fst` and `snd`:

```
fst :: (a, b) -> a
snd :: (a, b) -> b
```

You can also pattern match on tuples. This is often the most convenient way, and also works for tuples of larger sizes. The `fst` and `snd` functions work only on pairs.

Tuples are very useful in combination with lists. Here are some examples using the `zip`, `unzip` and `partition` functions from the `Data.List` module.

```
zip :: [a] -> [b] -> [(a, b)]    -- two lists to list of pairs
unzip :: [(a, b)] -> ([a], [b])  -- list of pairs to pair of lists
partition :: (a -> Bool) -> [a] -> ([a], [a])    -- elements that
         satisfy and don't satisfy a predicate
```

```
zip [1,2,3] [True,False,True]
  ==> [(1,True),(2,False),(3,True)]
unzip [("Fred",1), ("Jack",10), ("Helen",13)]
  ==> (["Fred","Jack","Helen"],[1,10,13])
partition (>0) [-1,1,-4,3,2,0]
  ==> ([1,3,2],[-1,-4,0])
```

Here’s an example of pattern matching on tuples:

```
swap :: (a,b) -> (b,a)
swap (x,y) = (y,x)
```

Here’s an example of pattern matching on tuples and lists at the same time:

```
-- sum all numbers that are paired with True
sumIf :: [(Bool,Int)] -> Int
sumIf [] = 0
sumIf ((True,x):xs) = x + sumIf xs
sumIf ((False,_):xs) = sumIf xs
```

```
sumIf [(True,1),(False,10),(True,100)]
  ==> 101
```

4.2 Interlude: Folding

Consider the functions `sumNumbers :: [Int] -> Int`, `myMaximum :: [Int] -> Int`, and `countNothings :: [Maybe a] -> Int` again.

```
sumNumbers :: [Int] -> Int
sumNumbers [] = 0
sumNumbers (x:xs) = x + sumNumbers xs

myMaximum :: [Int] -> Int
myMaximum [] = 0
myMaximum (x:xs) = go x xs
  where go biggest [] = biggest
        go biggest (x:xs) = go (max biggest x) xs

countNothings :: [Maybe a] -> Int
countNothings [] = 0
countNothings (Nothing : xs) = 1 + countNothings xs
countNothings (Just _  : xs) = countNothings xs
```

They have one common characteristic. They take a list and produce a value that depends on the values of the elements in the given list. They "crunch" or *fold* a list of many values into a single value.

Prelude has a function called `foldr`, which performs a *right associative fold* over a `Foldable` data type. We'll learn more about `Foldable` soon. At this point, it suffices to think of lists, so we define

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

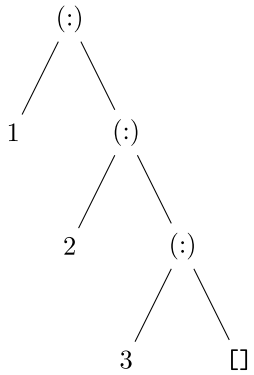
What this definition says, is that for an empty list `[] :: [a]`, `foldr` returns the default value `y :: b`. For any other list `x : xs`, `foldr` applies `f` to `x` and the result of `foldr f y xs` (i.e. folding over the rest of the list). It's a simple definition by recursion.

In other words, `foldr` calls its argument function `f` repeatedly with two arguments.

- The first argument is the current element from the list.

- The second argument is what `f` returned for the rest of the list.

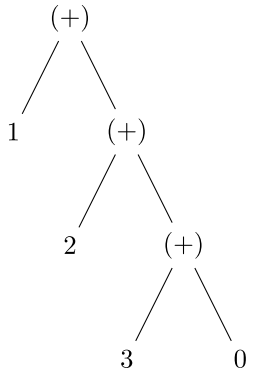
Consider the list `[1,2,3]`:



The expression `foldr (+) 0 [1,2,3]` evaluates as follows:

```
foldr (+) 0 [1,2,3] ==> foldr (+) 0 (1:2:3:[])
                    ==> 1 + (foldr (+) 0 (2:3:[]))
                    ==> 1 + (2 + (foldr (+) 0 (3:[])))
                    ==> 1 + (2 + (3 + (foldr (+) 0 [])))
                    ==> 1 + (2 + (3 + 0))
```

The result can be thought of as a tree:



One way to think about `foldr f y xs` is that it replaces the `(:)` operation with `f` and `[]` with `y`. In this case, `f` was `(+)` and `y` was `0`. If you write out how `sumNumbers [1,2,3]` behaves, you'll notice that it performs the same computation as `foldr (+) 0 [1,2,3]` does! More generally:

```
sumNumbers xs == foldr (+) 0 xs
```

Those more experienced with math may notice that we can prove this claim by *induction*: Firstly, `sumNumbers [] ==> 0` and `foldr (+) 0 [] ==> 0`, so in the base case `sumNumbers [] == foldr (+) 0 []`. Next, we may assume as our induction

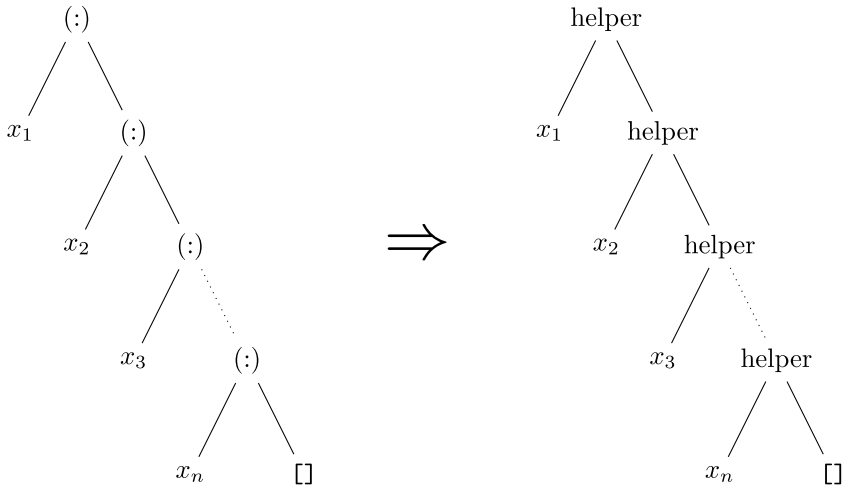
hypothesis that `sumNumbers xs == foldr (+) 0 xs` for any list `xs`. Then, for the list `x:xs`, we have `sumNumbers (x:xs) ==> x + sumNumbers xs`. Hence, `foldr (+) 0 (x:xs) ==> x + foldr (+) 0 xs ==> x + sumNumbers xs` by induction hypothesis. Therefore, by induction, the equation holds.

You don't need to read, write, or understand induction proofs in this course, but perhaps it is reassuring to know that properties and equalities of functions in Haskell can be (in principle) analysed mathematically, because Haskell is such a nice language. (Equalities and properties can be analysed in any programming language, but for Haskell, this analysis is especially convenient because Haskell is pure.)

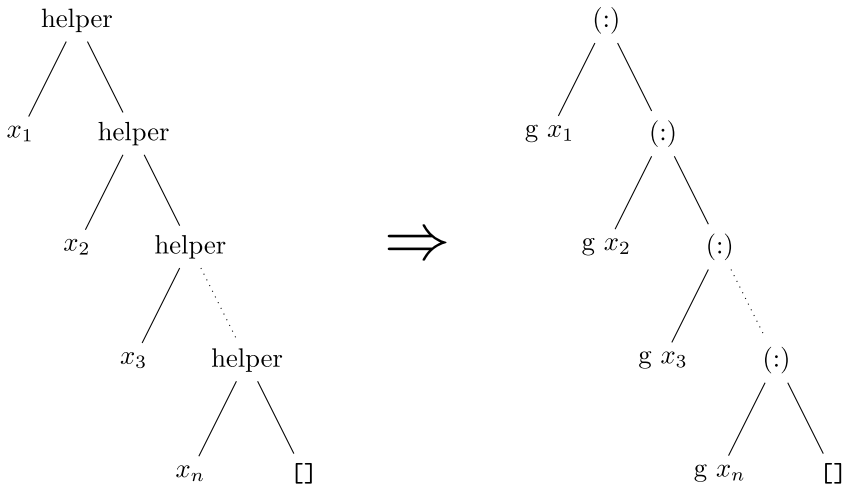
Another folding example is the `map` function:

```
map g xs = foldr helper [] xs
  where helper y ys = g y : ys
```

To see why this works, consider what `foldr helper [] [x1,x2,..,xn]` does:



Now, since `helper x xs ==> g x : xs` for every `x` and `xs`, we get that:



The resulting list, `[ g x1, g x2, g x3, ..., g xn ]`, is then exactly what we would have gotten with `map g xs`. (This could have been also proved by induction as we did for `sumNumbers`.) The lesson to take away is that folding is a particular, yet quite general, way to apply some transformation recursively into some structure (e.g. a list).

4.3 Type Classes

How can Haskell's `+` work on both `Int`s and `Double`s? Why can I compare all sorts of things with `==`? We've briefly mentioned constrained types earlier. Let's see what they really mean. Let's look at the types of `==` and `+`.

```
(==) :: (Eq a) => a -> a -> Bool
```

The type `(Eq a) => a -> a -> Bool` means: *for all types `a` that belong to the class `Eq`, this is a function of type `a -> a -> Bool`*. That is, if the type `a` is a member of the class `Eq`, you can give two values of type `a` to `==` and get a `Bool` result.

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

Similarly, the type `(Num a) => a -> a -> a` means: *for all types `a` that belong to the class `Num`, this is a function of type `a -> a -> a`*. That is, you can give two values of the same type `a` to `+` and get out a third value of type `a`, as long as `a` is a member of `Num`.

`Num` and `Eq` are type classes. A *type class* is a way to group together types that support similar operations.

Note! A type class is a collection of types. It doesn't have much to do with the classes of object oriented programming! In some situations, type classes can act like *interfaces* in object oriented programming. Unfortunately the functions in a type class are often called *methods*, adding to the confusion.

PS. remember how using type variables for polymorphism was called *parametric polymorphism*? The fancy word for what type classes achieve is *ad-hoc polymorphism*. The difference is that with parametric polymorphism the function (e.g. `head`) has the same implementation for all types, whereas with ad-hoc polymorphisms there are multiple implementations (consider `==` on numbers and strings).

4.4 Type Constraints

When you're working with a concrete type (not a type variable), you can just use type class functions (in this case, `(==)`):

```
f :: (Int -> Int) -> Int -> Bool
f g x = x == g x
```

Of course, if the type in question isn't a member of the right class, you get an error. For example:

```
addTrue :: Bool -> Bool
addTrue b = b + True
```

```
error:
    • No instance for (Num Bool) arising from a use of ‘+’
    • In the expression: b + True
      In an equation for ‘addTrue’: addTrue b = b + True
```

However in a *polymorphic* function, you need to add *type constraints*. This doesn't work:

```
f :: (a -> a) -> a -> Bool
f g x = x == g x
```

Luckily the error is nice:

```
error:
    • No instance for (Eq a) arising from a use of ‘==’
      Possible fix:
        add (Eq a) to the context of
          the type signature for:
            f :: (a -> a) -> a -> Bool
    • In the expression: x == g x
      In an equation for ‘f’: f g x = x == g x
```

To signal that `f` only works on types that are members of the `Eq` class, we add a type constraint `(Eq a) =>` to the type annotation.

```
f :: (Eq a) => (a -> a) -> a -> Bool
```

```
f g x = x == g x
```

If you don’ t have a type annotation, *type inference* can provide the constraints!

```
Prelude> f g x = x == g x
Prelude> :type f
f :: (Eq a) => (a -> a) -> a -> Bool
```

You can also have multiple constraints:

```
bothPairsEqual :: (Eq a, Eq b) => a -> a -> b -> b -> Bool
bothPairsEqual left1 left2 right1 right2 = left1 == left2 && right1
         == right2
```

4.5 Standard Type Classes

Here are some standard Haskell type classes you should know about.

4.5.1 `Eq`

We already saw the `Eq` class for equality comparisons. Here are the basic operations of the `Eq` class and some examples of their use. As you can see pretty much all the types we’ ve seen so far, except for functions, are members of `Eq`.

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

```
Prelude> 1 == 2
False
Prelude> 1 /= 2
True
Prelude> "Foo" == "Bar"
False
Prelude> [[1,2],[3,4]] == [[1,2],[3,4]]
True
Prelude> (\x -> x+1) == (\x -> x+2)

<interactive>:5:1: error:
    • No instance for (Eq (Integer -> Integer))
        arising from a use of ‘==’
```

```
        (maybe you haven't applied a function to enough arguments?)
    • In the expression: (\ x -> x + 1) == (\ x -> x + 2)
      In an equation for ‘it’: it = (\ x -> x + 1) == (\ x -> x + 2)
```

There are some other useful functions that use the `Eq` class, like `nub` from the module `Data.List`.

```
Prelude> import Data.List
Prelude Data.List> :t nub
nub :: Eq a => [a] -> [a]
Prelude Data.List> nub [3,5,3,1,1]      -- eliminates duplicates
[3,5,1]
```

4.5.2 Ord

The `Ord` class is for ordering (less than, greater than). Again, here are the basic operations and some examples of their use. Note the new `Ordering` type. It has values `LT` for "less than", `EQ` for "equal" and `GT` for "greater than".

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

```
Prelude> compare 1 1                -- 1 is EQual to 1
EQ
Prelude> compare 1 3                -- 1 is Less Than 3
LT
Prelude> compare 1 0                -- 1 is Greater Than 0
GT
Prelude> min 5 3
3
Prelude> max 5 3
5
Prelude> "aardvark" < "banana"      -- strings are compared
         alphabetically
True
Prelude> [1,2,3] > [2,5]            -- lists are compared like
         strings
False
```

```
Prelude> [1,2,3] > [1,1]
True
```

When we can compare values, we can also sort lists of them. The function `sort` from `Data.List` works on all types that belong to the `Ord` class.

```
Prelude> import Data.List
Prelude Data.List> :t sort
sort :: Ord a => [a] -> [a]
Prelude Data.List> sort [6,1,4,8,2]
[1,2,4,6,8]
Prelude Data.List> sort "black sphinx of quartz, judge my vow!"
         -- remember, strings are lists!
"      !,aabcdefghijklmnoopqrstuuvwxyz"
```

As a last example, let's sort a list of lists according to length. We'll need two helper functions:

```
-- from the module Data.Ord
-- compares two values "through" the function f
comparing :: (Ord a) => (b -> a) -> b -> b -> Ordering
comparing f x y = compare (f x) (f y)

-- from the module Data.List
-- sorts a list using the given comparison function
sortBy :: (a -> a -> Ordering) -> [a] -> [a]
```

Now the implementation of `sortByLength` is straightforward:

```
-- sorts lists by their length
sortByLength :: [[a]] -> [[a]]
sortByLength = sortBy (comparing length)
```

```
sortByLength [[1,2,3],[4,5],[4,5,6,7]]   ==>  [[4,5],[1,2,3],
         [4,5,6,7]]
```

4.5.3 `Num`, `Integral`, `Fractional`, `Floating`

The `Num` class contains integer arithmetic:

```
(+) :: Num a => a -> a -> a
(-) :: Num a => a -> a -> a
(*) :: Num a => a -> a -> a
negate :: Num a => a -> a    -- 0-x
abs :: Num a => a -> a       -- absolute value
signum :: Num a => a -> a    -- -1 for negative values, 0 for 0, +1
         for positive values
fromInteger :: Num a => Integer -> a
```

`Num` also shows up in the types of integer literals:

```
Prelude> :t 12
12 :: Num p => p
```

This means that a literal like `12` can be interpreted as a member of any type implementing `Num`. When GHC reads a number literal like, `12` it produces code that corresponds to `fromIntegral 12`.

```
Prelude> 1 :: Int
1
Prelude> 1 :: Double
1.0
Prelude> fromIntegral 1 :: Double
1.0
```

`Integral` is the class of types that represent whole numbers, like `Int` and `Integer`. The most interesting functions are `div` and `mod` for integer division and remainder. All types that belong to `Integral` also belong to `Num`.

```
div :: Integral a => a -> a -> a
mod :: Integral a => a -> a -> a
```

`Fractional` is the class for types that have division. All types that belong to `Fractional` also belong to `Num`.

```
(/) :: Fractional a => a -> a -> a
```

`Floating` contains some additional operations that only make sense for floating point numbers. All types that belong to `Floating` also belong to `Fractional` (and to

Num).

```
sqrt :: Floating a => a -> a
sin :: Floating a => a -> a
```

4.5.4 Read and Show

The Show and Read classes are for the functions show and read, that convert values to and from Strings.

```
show :: Show a => a -> String
read :: Read a => String -> a
```

```
Prelude> show 3
"3"
Prelude> read "3" :: Int
3
Prelude> read "3" :: Double
3.0
```

As you can see above, you often need to use a type annotation with read so that the compiler can choose the right implementation.

4.5.5 Sidenote: Foldable

One more thing! You might remember that it was mentioned earlier that the type of length isn't [a] -> Int but something more general. Let's have a look:

```
Prelude> :t length
length :: Foldable t => t a -> Int
```

This type looks a bit different than the ones we've seen before. The type variable t has an argument a. We'll look at type classes like this in more detail in part 2, but here's a crash course.

What Foldable represents is types that you can fold over. The true type of foldr is:

```
foldr :: Foldable t => (a -> b -> b) -> b -> t a -> b
```

We've succesfully used the fact that lists are `Foldable` since we've managed to use `length` and `foldr` on lists. However, `Maybe` is also `Foldable`! The `Foldable` instance for `Maybe` just pretends that values of `Maybe a` are like lists of length 0 or 1:

```
foldr (+) 1 Nothing   ==> 1
foldr (+) 1 (Just 3)  ==> 4
length Nothing        ==> 0
length (Just 'a')     ==> 1
```

We'll meet some more foldable types next.

4.6 More Data Structures

Now that we are familiar with the standard type classes, we can look at one of their applications: the `Map` and `Array` data structures.

4.6.1 `Data.Map`

The `Data.Map` module defines the `Map` type. Maps are search trees for key-value pairs. One way to look at this is that a value of type `Map k v` is roughly the same as a value of type `[(k,v)]`, a list of pairs. However, the operations on a map are more efficient than operations on a list.

Since `Data.Map` contains some function with the same names as `Prelude` functions, the namespace needs to be imported *qualified*:

```
import qualified Data.Map as Map
```

Now we can refer to the map type as `Map.Map`, and to various map functions like `Map.insert`. Here are the most important functions for maps:

```
-- Create a Map from a list of key-value pairs
Map.fromList :: Ord k => [(k, a)] -> Map.Map k a

-- Insert a value into a map. Overrides any previous value with the
        same key.
-- Returns a new map. Does not mutate the given map.
Map.insert :: Ord k => k -> a -> Map.Map k a -> Map.Map k a

-- Get a value from a map using a key. Returns Nothing if the key
        was not present in the map.
Map.lookup :: Ord k => k -> Map.Map k a -> Maybe a
```

```
-- An empty map
Map.empty :: Map.Map k a
```

The `Ord` constraint for the key type of the map is needed because maps are implemented as *ordered binary search trees*.

Note that like all Haskell values, maps are *immutable* meaning you can't change a map once you define it. However, map operations like `insert` produce a *new* map. To perform multiple map operations you need to reuse the return value. Here's a GHCi session operating on a map.

```
Prelude> import qualified Data.Map as Map
Prelude Map> values = Map.fromList [("z",3),("w",4)]
Prelude Map> Map.lookup "z" values
Just 3
Prelude Map> Map.lookup "banana" values
Nothing
Prelude Map> Map.insert "x" 7 values
fromList [("w",4),("x",7),("z",3)]
Prelude Map> values                                    -- note
        immutability!
fromList [("w",4),("z",3)]
Prelude Map> Map.insert "x" 1 (Map.insert "y" 2 values)   -- two
        insertions
fromList [("w",4),("x",1),("y",2),("z",3)]
Prelude Map>
```

Here's an example of representing a bank as a `Map String Int` (map from account name to account balance), and withdrawing some money from an account:

```
withdraw :: String -> Int -> Map.Map String Int -> Map.Map String
        Int
withdraw account amount bank =
  case Map.lookup account bank of
    Nothing  -> bank                                  -- account
        not found, no change
    Just sum -> Map.insert account (sum-amount) bank   -- set new
        balance
```

Here's how you might use the `withdraw` function in `GHCi`. Note how the maps get printed as `fromList` invocations. Also note how calling `withdraw ... bank` returns a *new* bank and doesn't change the existing bank.

```
GHCi> bank = Map.fromList [("Bob",100),("Mike",50)]
GHCi> withdraw "Bob" 80 bank
fromList [("Bob",20),("Mike",50)]
GHCi> bank                              -- note immutability
fromList [("Bob",100),("Mike",50)]
GHCi> withdraw "Bozo" 1000 bank
fromList [("Bob",100),("Mike",50)]
```

`Data.Map` defines all sorts of useful higher-order functions for updating maps. We can rewrite the `withdraw` function using `Data.Map.adjust`:

```
withdraw :: String -> Int -> Map.Map String Int -> Map.Map String
         Int
withdraw account amount bank = Map.adjust (\x -> x-amount) account
         bank
```

Note! There are separate `Data.Map.Strict` and `Data.Map.Lazy` implementations. When you import `Data.Map` you get `Data.Map.Lazy`. You can find the documentation for all the `Data.Map` functions in the docs for `Data.Map.Lazy`. We won't go into their differences here, but mostly you should use `Data.Map.Strict` in real code.

4.6.2 `Data.Array`

Another type that works kind of like a list but is more efficient for some operations is the array. Arrays are familiar from many other programming languages, but Haskell arrays are a bit different.

Unlike the `Data.Map` module, the `Data.Array` can just be imported normally:

```
import Data.Array
```

Now we can look at the type of the `array` function that constructs an array.

```
array :: Ix i => (i, i) -> [(i, e)] -> Array i e
```

There are a couple of things to notice here. First of all, the `Array` type is parameterized by *two* types: the index type and the element type. Most other programming languages only parameterize arrays with the element type, but the index type is always `int`. In Haskell, we can have, for example, an `Array Char Int`:

an array indexed by characters, or `Array Bool String`, an array indexed by booleans, or even `Array (Int,Int) Int`, a two-dimensional array of ints.

Not all types can be index types. Only types that are similar to integers are suitable. That is the reason for the `Ix i` class constraint. The `Ix` class collects all the types that can be used as array indices.

Secondly, the `array` function takes an extra `(i,i)` parameter. These are the minimum and maximum indices of the array. Unlike some other languages, where arrays always start at index 0 or 1, in Haskell you can define an array that starts from 7 and goes to 11. So here's that array:

```
myArray :: Array Int String
myArray = array (7,11) [(7,"seven"), (8,"eight"), (9,"nine"),
          (10,"ten"), (11,"ELEVEN")]
```

Listing all the indices and elements in order can be a bit cumbersome, so there's also the `listArray` constructor that just takes a list of elements in order:

```
listArray :: Ix i => (i, i) -> [e] -> Array i e
```

```
myArray :: Array Int String
myArray = listArray (7,11) ["seven", "eight", "nine", "ten",
          "ELEVEN"]
```

Arrays are used with two new operators:

```
-- Array lookup
(!) :: Ix i => Array i e -> i -> e
-- Array update
(//) :: Ix i => Array i e -> [(i, e)] -> Array i e
```

Here's an example `GHCi` session:

```
Prelude> import Data.Array
Prelude Data.Array> myArray = listArray (7,11) ["seven", "eight",
          "nine", "ten", "ELEVEN"]
Prelude Data.Array> myArray
array (7,11) [(7,"seven"),(8,"eight"),(9,"nine"),(10,"ten"),
          (11,"ELEVEN")]
Prelude Data.Array> myArray ! 8
```

```
"eight"
Prelude Data.Array> myArray // [(8,"ocho"),(9,"nueve")]
array (7,11) [(7,"seven"),(8,"ocho"),(9,"nueve"),(10,"ten"),
          (11,"ELEVEN")]
```

You might be wondering why the `(//)` operator does multiple updates at once. The reason is the main weakness of Haskell arrays: immutability. Since arrays can't be changed in place, `(//)` must copy the whole array. This is why in Haskell it's often preferable to use lists or maps to store data that needs to be updated. However, arrays may still be useful when constructed once and then used for a large number of lookups. We'll get back to how Haskell data structures work in the next lecture.

Note! In this course we'll use only `Array`, a simple array type that's specified in the Haskell standard. There are many other array types like the mutable `IOArray` and the somewhat obscure `DiffArray`. There are also type classes for arrays like `IArray` and `MArray`. In addition to arrays there is a wide family of `Vector` types that can be more practical than `Array` for real programs.

4.6.3 Sidenote: Folding over Maps & Arrays

The `Map` and `Array` type are instances of `Foldable` just like lists are! This means you can use functions like `length` and `foldr` on them:

```
length (array (7,11) [(7,"seven"),(8,"eight"),(9,"nine"),(10,"ten"),
          (11,"ELEVEN")])
  ==> 5
foldr (+) 0 (Map.fromList [("banana",3),("egg",7)])
  ==> 10
```

4.7 Reading Docs

Haskell libraries tend to have pretty good docs. We've linked to docs via Hackage (https://hackage.haskell.org) previously, but it's important to know how to find the docs by yourself too. The tool for generating Haskell documentation is called *Haddock* so sometimes Haskell docs are referred to as *haddocks*.

Hackage is the Haskell package repository (just like PyPI for Python, Maven Central for Java or NPM for Javascript). In addition to the actual packages, it hosts documentation for them. Most of the modules that we use on this course are in the package called `base`. You can browse the docs for the base package at https://hackage.haskell.org/package/base-4.16.4.0/.

When you're not quite sure where the function you're looking for is, Hoogle (https://hoogle.haskell.org/) can help. Hoogle is a search engine for Haskell

documentation. It is a great resource when you need to check what was the type of `foldr` or which packages contain a function named `reverse`.

Finally, since this course is using the `stack` tool, you can also browse the documentation for the libraries stack has installed for you with the commands

```
stack haddock --open
stack haddock --open <package>
```

This has the added benefit of getting exactly the right version of the documentation.

In summary, here are the main ways of reading Haskell library documentation:

- If you know the name of the package you browse the docs via https://hackage.haskell.org/.
- If you know the name of the function you can find it using https://hoogle.haskell.org/.
- If you're using `stack` you can use `stack haddock --open` or `stack haddock --open <package>` to open docs in your browser.

4.8 Quiz

What is the type of `swap . swap`?

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

What is the type of `\f g x -> (f x, g x)`?

a. `(a -> b) -> (c -> d) -> (a,c) -> (b, d)`
b. `(a -> b) -> (a -> c) -> a -> (b, c)`
c. `(a -> b) -> (b -> a) -> a -> (b, a)`

What is the type of `\t -> (fst . fst $ t, (snd . fst $ t, snd t))`?

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

What does the function `foldr (\x xs -> xs ++ [x]) []` do?

a. It doesn't change its input list at all
b. It changes the associativity of a list from left to right

c. It reverses its input list

What does the function `foldr (\(x, y) zs -> x : y : zs) []` do?

a. It turns a list of pairs into a pair of lists
b. It turns a pair of lists into a list of pairs
c. It turns a list of pairs into a list of elements

What is the type of `foldr (\n b -> n == 3 && b)`?

a. `(Foldable t, Eq a, Num a) => Bool -> t a -> Bool`
b. `(Foldable t, Eq a, Num a, Bool b) => b -> t a -> b`
c. `(Foldable t, Eq a, Num a) => Bool -> [ a ] -> Bool`

What is the type of `\x -> case x of (True, "Foo") -> show True ++ "Foo"`?

a. `Either Bool String -> String`
b. `(Bool, String) -> String`
c. `Show a => (Bool, String) -> a`

4.9 Exercises

- Set4a: type classes
- Set4b: folds

5 Lecture 5: You Need String for a Knot

- Type system
- Defining custom types

5.1 Algebraic Datatypes

Haskell has a system called *algebraic datatypes* for defining new types. This sounds fancy, but is rather simple. Let's dive in by looking at the standard library definitions of some familiar types:

```haskell
data Bool = True | False
data Ordering = LT | EQ | GT
```

With this syntax you too can define types:

```haskell
-- definition of a type with three values
data Color = Red | Green | Blue

-- a function that uses pattern matching on our new type
rgb :: Color -> [Double]
rgb Red = [1,0,0]
rgb Green = [0,1,0]
rgb Blue = [0,0,1]
```

```
Prelude> :t Red
Red :: Color
Prelude> :t [Red,Blue,Green]
[Red,Blue,Green] :: [Color]
Prelude> rgb Red
[1.0,0.0,0.0]
```

5.1.1 Fields

Types like `Bool`, `Ordering` and `Color` that just list a bunch of constants are called *enumerations* or *enums* in Haskell and other languages. Enums are useful, but you need other types as well. Here we define a type for reports containing an id number, a title, and a body:

```haskell
data Report = ConstructReport Int String String
```

This is how you create a report:

```
Prelude> :t ConstructReport 1 "Title" "This is the body."
ConstructReport 1 "Title" "This is the body." :: Report
```

You can access the fields with pattern matching:

```haskell
reportContents :: Report -> String
reportContents (ConstructReport id title contents) = contents
setReportContents :: String -> Report -> Report
setReportContents contents (ConstructReport id title _contents) =
         ConstructReport id title contents
```

5.1.2 Constructors

The things on the right hand side of a `data` declaration are called *constructors*. `True`, `False`, `Red` and `ConstructReport` are all examples of constructors. A type can have multiple constructors, and a constructor can have zero or more fields.

Here is a datatype for a standard playing card. It has five constructors, of which `Joker` has zero fields and the others have one field.

```
data Card = Joker | Heart Int | Club Int | Spade Int | Diamond Int
```

Constructors with fields have function type and can be used wherever functions can:

```
Prelude> :t Heart
Heart :: Int -> Card
Prelude> :t Club
Club :: Int -> Card
Prelude> map Heart [1,2,3]
[Heart 1,Heart 2,Heart 3]
Prelude> (Heart . (\x -> x+1)) 3
Heart 4
```

5.1.3 Sidenote: Deriving

By the way, there's something missing from our `Card` type. Look at how it behaves compared to `Ordering` and `Bool`:

```
Prelude> EQ
EQ
Prelude> True
True
Prelude> Joker
<interactive>:1:0:
    No instance for (Show Card)
      arising from a use of `print' at <interactive>:1:0-4
    Possible fix: add an instance declaration for (Show Card)
    In a stmt of a 'do' expression: print it
```

The problem is that Haskell does not know how to print the types we defined. As the error says, they are not part of the `Show` class. The easy solution is to just add a `deriving Show` after the type definition:

```
data Card = Joker | Heart Int | Club Int | Spade Int | Diamond Int
  deriving Show
```

```
Prelude> Joker
Joker
```

The `deriving` syntax is a way to automatically make your class a member of certain basic type classes, most notably `Read`, `Show` and `Eq`. We'll talk more about what this means later.

5.1.4 Algebraic?

So why are these datatypes called algebraic? This is because, theoretically speaking, each datatype can be a *sum* of constructors, and each constructor is a *product* of fields. It makes sense to think of these as sums and products for many reasons, one being that we can count the possible values of each type this way:

```
data Bool = True | False            -- corresponds to 1+1. Has 2
        possible values.
data TwoBools = TwoBools Bool Bool  -- corresponds to Bool*Bool,
        i.e. 2*2. Has 4 possible values.
data Complex = Two Bool Bool | One Bool | None
                                    -- corresponds to
        Bool*Bool+Bool+1 = 2*2+2+1 = 7. Has 7 possible values.
```

There is a rich theory of algebraic datatypes. If you're interested, you might find more info here or here.

5.2 Type Parameters

We introduced type parameters and parametric polymorphism when introducing lists in Lecture 2. Since then, we've seen other parameterized types like `Maybe` and `Either`. Now we'll learn how we can define our own parameterized types.

5.2.1 Defining Parameterized Types

The definition for `Maybe` is:

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

What's `a`? We define a parameterized type by mentioning a *type variable* (`a` in this case) on the left side of the `=` sign. We can then use the same type variable in fields for our constructors. This is analogous to polymorphic functions. Instead of defining separate functions

```
headInt :: [Int] -> Int
headBool :: [Bool] -> Bool
```

and so on, we define one function `head :: [a] -> a` that works for all types `a`. Similarly, instead of defining multiple types

```
data MaybeInt = NothingInt | JustInt Int
data MaybeBool = NothingBool | JustBool Bool
```

we define one type `Maybe a` that works for all types `a`.

Here's our first own parameterized type `Described`. The values of type `Described a` contain a value of type `a` and a `String` description.

```
data Described a = Describe a String

getValue :: Described a -> a
getValue (Describe x _) = x

getDescription :: Described a -> String
getDescription (Describe _ desc) = desc
```

```
Prelude> :t Describe
Describe :: a -> String -> Described a
Prelude> :t Describe True "This is true"
Describe True "This is true" :: Described Bool
Prelude> getValue (Describe 3 "a number")
3
Prelude> getDescription (Describe 3 "a number")
"a number"
```

5.2.2 Syntactic Note

In the above definitions, we've used `a` as a type variable. However any word that starts with a lower case letter is fine. We could have defined `Maybe` like this:

```
data Maybe theType = Nothing | Just theType
```

The rules for Haskell identifiers are:

- Type variables and names for functions and values start lower case (e.g. `a`, `map`, `xs`)
- Type names and constructor names start with upper case (e.g. `Maybe`, `Just`, `Card`, `Heart`)

Note that a type and its constructor can have the same name. This is very common in Haskell code for types that only have one constructor. In this material we try to avoid it to avoid confusion. Here are some examples:

```
data Pair a = Pair a a
data Report = Report Int String String
```

```
Prelude> :t Pair
Pair :: a -> a -> Pair a
```

Beware of mixing up types and constructors. Luckily types and constructors can never occur in the same context, so you get a nice error:

```
Prelude> Maybe                                -- trying to use a type
         name as a value
<interactive>:1:1: error:
    • Data constructor not in scope: Maybe

Prelude> undefined :: Nothing                 -- trying to use a
         constructor as a type
<interactive>:2:14: error:
    Not in scope: type constructor or class ‘Nothing’
```

5.2.3 Sidenote: Multiple Type Parameters

Types can have multiple type parameters. The syntax is similar to defining functions with many arguments. Here's the definition of the standard `Either` type:

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

5.3 Recursive Types

So far, all of the types we've defined have been of constant size. We can represent one report or one colour, but how could we represent a collection of things? We could use lists of course, but could we define a list type ourselves?

Just like Haskell functions, Haskell data types can be *recursive*. This is no weirder than having an object in Java or Python that refers to another object of the same class. This is how you define a list of integers:

```haskell
data IntList = Empty | Node Int IntList
  deriving Show

ihead :: IntList -> Int
ihead (Node i _) = i

itail :: IntList -> IntList
itail (Node _ t) = t

ilength :: IntList -> Int
ilength Empty = 0
ilength (Node _ t) = 1 + ilength t
```

We can use the functions defined above to work with lists of integers:

```
Prelude> ihead (Node 3 (Node 5 (Node 4 Empty)))
3
Prelude> itail (Node 3 (Node 5 (Node 4 Empty)))
Node 5 (Node 4 Empty)
Prelude> ilength (Node 3 (Node 5 (Node 4 Empty)))
3
```

Note that we can't put values other than `Int`s inside our `IntList`:

```
Prelude> Node False Empty

<interactive>:3:6: error:
    • Couldn't match expected type ‘Int’ with actual type ‘Bool’
    • In the first argument of ‘Node’, namely ‘False’
```

```
        In the expression: Node False Empty
        In an equation for ‘it’: it = Node False Empty
```

To be able to put any type of element in our list, let’s do the same thing with a type parameter. This is the same as the built in type `[a]`, but with slightly clunkier syntax:

```
data List a = Empty | Node a (List a)
  deriving Show
```

Note how we need to pass the the type parameter `a` onwards in the recursion. We need to write `Node a (List a)` instead of `Node a List`. The `Node` constructor has two arguments. The first has type `a`, and the second has type `List a`. Here are the reimplementations of some standard list functions for our `List` type:

```
lhead :: List a -> a
lhead (Node h _) = h

ltail :: List a -> List a
ltail (Node _ t) = t

lnull :: List a -> Bool
lnull Empty = True
lnull _     = False

llength :: List a -> Int
llength Empty = 0
llength (Node _ t) = 1 + llength t
```

```
Prelude> lhead (Node True Empty)
True
Prelude> ltail (Node True (Node False Empty))
Node False Empty
Prelude> lnull Empty
True
```

Note that just like with normal Haskell lists, we can’t have elements of different types in the same list:

```
Prelude> Node True (Node "foo" Empty)

<interactive>:5:12: error:
    • Couldn't match type ‘[Char]’ with ‘Bool’
```

```
    Expected type: List Bool
      Actual type: List [Char]
• In the second argument of ‘Node’, namely ‘(Node "foo" Empty)’
  In the expression: Node True (Node "foo" Empty)
  In an equation for ‘it’: it = Node True (Node "foo" Empty)
```

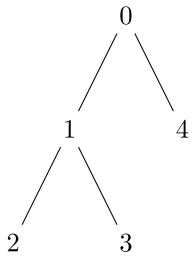
5.3.1 Example: Growing a Tree

Just like a list, we can also represent a binary tree:

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

Our tree contains nodes, which contain a value of type `a` and two child trees, and empty trees.

In case you're not familiar with binary trees, they're a data structure that's often used as the basis for other data structures (`Data.Map` is based on trees!). Binary trees are often drawn as (upside-down) pictures, like this:



The highest node in the tree is called the *root* (`0` in this case), and the nodes with no children are called `leaves` (`2`, `3` and `4` in this case). We can define this tree using our `Tree` type like this:

```
example :: Tree Int
example = (Node 0 (Node 1 (Node 2 Empty Empty)
                          (Node 3 Empty Empty))
                  (Node 4 Empty Empty))
```

The height of a binary tree is length of the longest path from the root to a leaf. In Haskell terms, it's how many nested levels of `Node` constructors you need to build the tree. The height of our example tree is 3. Here's a function that computes the height of a tree:

```
treeHeight :: Tree a -> Int
treeHeight Empty = 0
```

```
treeHeight (Node _ l r) = 1 + max (treeHeight l) (treeHeight r)
```

```
treeHeight Empty ==> 0
treeHeight (Node 2 Empty Empty)
  ==> 1 + max (treeHeight Empty) (treeHeight Empty)
  ==> 1 + max 0 0
  ==> 1
treeHeight (Node 1 Empty (Node 2 Empty Empty))
  ==> 1 + max (treeHeight Empty) (treeHeight (Node 2 Empty Empty))
  ==> 1 + max 0 1
  ==> 2
treeHeight (Node 0 (Node 1 Empty (Node 2 Empty Empty)) Empty)
  ==> 1 + max (treeHeight (Node 1 Empty (Node 2 Empty Empty)))
         (treeHeight Empty)
  ==> 1 + max 2 0
  ==> 3
```

In case you're familiar with *binary search trees*, here are the definitions of the lookup and insert opertions for a binary search tree. If you don't know what I'm talking about, you don't need to understand this.

```
lookup :: Int -> Tree Int -> Bool
lookup x Empty = False
lookup x (Node y l r)
  | x < y = lookup x l
  | x > y = lookup x r
  | otherwise = True

insert :: Int -> Tree Int -> Tree Int
insert x Empty = Node x Empty Empty
insert x (Node y l r)
  | x < y = Node y (insert x l) r
  | x > y = Node y l (insert x r)
  | otherwise = Node y l r
```

5.4 Record Syntax

If some fields need to be accessed often, it can be convenient to have helper functions for reading those fields. For instance, the type `Person` might have multiple fields:

```
data Person = MkPerson String Int String String String deriving Show
```

A list of persons might look like the following:

```
people :: [Person]
people = [ MkPerson "Jane Doe" 21 "Houston" "Texas" "Engineer"
         , MkPerson "Maija Meikäläinen" 35 "Rovaniemi" "Finland"
         "Engineer"
         , MkPerson "Mauno Mutikainen" 27 "Turku" "Finland"
         "Mathematician"
         ]
```

Suppose that we need to find all engineers from Finland:

```
query :: [Person] -> [Person]
query [] = []
query (MkPerson name age town state profession):xs
  | state == "Finland" && profession == "Engineer" =
      (MkPerson name age town state profession) : query xs
  | otherwise = query xs
```

Thus,

```
query people ==> [MkPerson "Maija Meikäläinen" 35 "Rovaniemi"
         "Finland" "Engineer"]
```

Note that the types of the fields give little information on what is the intended content in those fields. We need to remember in all places in the code that `town` goes before `state` and not vice versa.

Haskell has a feature called *record syntax* that is helpful in these kinds of cases. The datatype `Person` can be defined as a record:

```
data Person = MkPerson { name :: String, age :: Int, town :: String,
         state :: String, profession :: String}
  deriving Show
```

We can still define values of `Person` normally, but the `Show` instance prints the field names for us:

```
Prelude> MkPerson "Jane Doe" 21 "Houston" "Texas" "Engineer"
```

```
MkPerson {name = "Jane Doe", age = 21, town = "Houston", state =
         "Texas", profession = "Engineer"}
```

However, we can also define values using record syntax. Note how the fields don't need to be in any specific order now that they have names.

```
Prelude> MkPerson {name = "Jane Doe", town = "Houston", profession =
         "Engineer", state = "Texas", age = 21}
MkPerson {name = "Jane Doe", age = 21, town = "Houston", state =
         "Texas", profession = "Engineer"}
```

Most importantly, We get *accessor functions* for the fields for free:

```
Prelude> :t profession
profession :: Person -> String
Prelude> profession (MkPerson "Jane Doe" 21 "Houston" "Texas"
         "Engineer")
"Engineer"
```

We can now rewrite the query function using these accessor functions:

```
query :: [Person] -> [Person]
query []     = []
query (x:xs)
  | state x == "Finland" && profession x == "Engineer" =
      x : query xs
  | otherwise = query xs
```

You'll probably agree that the code looks more pleasant now.

5.5 Algebraic Datatypes: Summary

- Types are defined like this

```
data TypeName = ConstructorName FieldType FieldType2 |
         AnotherConstructor FieldType3 | OneMoreCons
```

- ... or like this if we're using type variables

```
data TypeName variable = Cons1 variable Type1 | Cons2 Type2 variable
```

- You can have one or more constructors
- Each constructor can have zero or more fields
- Constructors start with upper case, type variables with lower case
- Values are handled with pattern matching:

```
foo (ConstructorName a b) = a+b
foo (AnotherConstructor _) = 0
foo OneMoreCons = 7
```

- Constructors are just functions:

```
ConstructorName :: FieldType -> FieldType2 -> TypeName
Cons1 :: a -> Type1 -> TypeName a
```

- You can also define datatypes using record syntax:

```
data TypeName = Constructor { field1 :: Field1Type, field2 ::
         Field2Type }
```

This gives you accessor functions like `field1 :: TypeName -> Field1Type` for free.

5.6 Sidenote: Other Ways of Defining Types

In addition to the `data` keyword, there are two additional ways of defining types in Haskell.

The `newtype` keyword works like `data`, but you can only have a single constructor with a single field. It's sometimes wise to use `newtype` for performance resons, but we'll get back to those in part 2.

The `type` keyword introduces a *type alias*. Type aliases don't affect type checking, they just offer a shorthand for writing types. For example the familiar `String` type is an alias for `[Char]`:

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

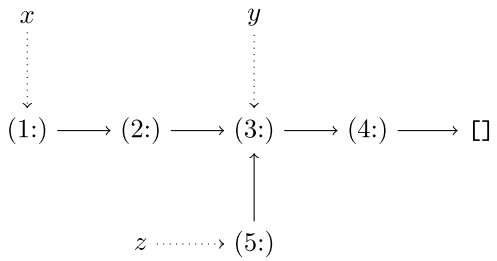
This means that whenever the compiler reads `String`, it just immediately replaces it with `[Char]`. Type aliases seem useful, but they can easily make reading type errors harder.

5.7 How Do Algebraic Datatypes Work?

Remember how lists were represented in memory as linked lists? Let's look in more detail at what algebraic datatypes look like in memory.

Haskell data forms *directed graphs* in memory. Every constructor is a node, every field is an edge. Names (of variables) are pointers into this graph. Different names can *share* parts of the structure. Here's an example with lists. Note how the last two elements of `x` are shared with `y` and `z`.

```
let x = [1,2,3,4]
    y = drop 2 x
    z = 5:y
```

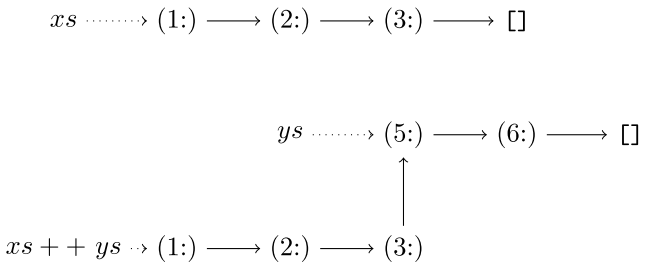


What happens when you make a new version of a datastructure is called *path copying*. Since Haskell data is immutable, the changed parts of the datastructure get copied, while the unchanged parts can be shared between the old and new versions.

Consider the definition of ++:

```
[]     ++ ys = ys
(x:xs) ++ ys = x:(xs ++ ys)
```

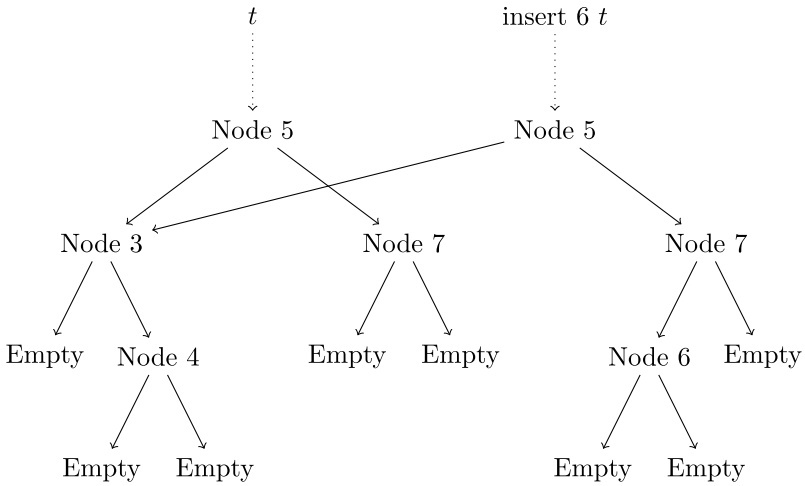
We are making a copy of the first argument while we walk it. For every `:` constructor in the first input list, we are creating a new `:` constructor in the output list. The second argument can be shared. It is not used at all in the recursion. Visually:



One more way to think about it is this: we want to change the `tail` pointer of the list element `(3:)`. That means we need to make a new `(3:)`. However the `(2:)` points to the `(3:)` so we need a new copy of the `(2:)` as well. Likewise for `(1:)`.

The graphs that we get when working with lists are fairly simple. As a more involved example, here is what happens in memory when we run the binary tree insertion example from earlier in this lecture.

```
insert :: Int -> Tree Int -> Tree Int
insert x Empty = Node x Empty Empty
insert x (Node y l r)
  | x < y = Node y (insert x l) r
  | x > y = Node y l (insert x r)
  | otherwise = Node y l r
```



Note how the old and the new tree share the subtree with 3 and 4 since it wasn't changed, but the node 7 that was "changed" and all nodes above it get copied.

5.8 Quiz

Why can't we `map Nothing`?

a. Because `Nothing` doesn't take arguments
b. Because `Nothing` returns nothing

c. Because `Nothing` is a constructor.

If we define `data Boing = Frick String Boing (Int -> Bool)`, what is the type of `Frick`?

a. `Boing`
b. `String -> Boing -> Int -> Bool -> Boing`
c. `String -> Boing -> (Int -> Bool) -> Boing`

If we define `data ThreeLists a b c = ThreeLists [a] [b] [c]`, what is the type of the constructor `ThreeLists`?

a. `[a] -> [b] -> [c] -> ThreeLists`
b. `a -> b -> c -> ThreeLists a b c`
c. `[a] -> [b] -> [c] -> ThreeLists a b c`
d. `[a] -> [b] -> [c] -> ThreeLists [a] [b] [c]`

If we define `data TwoLists a b = TwoList {aList :: [a], bList :: [b]}`, what is the type of the function `aList`?

a. `aList` is not a function, it is a field
b. `TwoLists a b -> [a]`
c. `[a] -> TwoLists a b`
d. `[a]`

5.9 Exercises

- Set5a: using and defining algebraic datatypes
- Set5b: playing with binary trees

6 Lecture 6: Working Class Hero

We've seen class constraints like `Eq a =>` in types. We know how to use existing classes with existing types. But how do we use existing classes with our own types? How can we define our own classes?

Here's how to make your own type a member of the `Eq` class:

```haskell
data Color = Black | White

instance Eq Color where
  Black == Black  = True
```

```
  White == White  = True
  _     == _      = False
```

A class instance is an `instance` block that contains definitions for the functions in that class. Here we define how `==` works on `Color`.

6.1 Syntax of Classes and Instances

A type class is defined using `class` syntax. The functions in the class are given types. Here's a class `Size` that contains one function, `size`:

```
class Size a where
  size :: a -> Int
```

Instances of a class are defined with `instance` syntax we've just seen. Here is how we make `Int` and `[a]` members of the `Size` class:

```
instance Size Int where
  size x = abs x

instance Size [a] where
  size xs = length xs
```

Our class `Size` behaves just like existing type classes. We can use `size` anywhere where a function can be used, and Haskell can infer types with `Size` constraints for us:

```
Prelude> :t size
size :: Size a => a -> Int
Prelude> size [True,False]
2
Prelude> sizeBoth a b = [size a, size b]
Prelude> :t sizeBoth
sizeBoth :: (Size a1, Size a2) => a1 -> a2 -> [Int]
```

A class can contain multiple functions, and even constants. Here we define a new version of the `Size` class with more content.

```
class Size a where
  empty :: a
  size :: a -> Int
```

```
  sameSize :: a -> a -> Bool

instance Size (Maybe a) where
  empty = Nothing

  size Nothing = 0
  size (Just a) = 1

  sameSize x y = size x == size y

instance Size [a] where
  empty = []
  size xs = length xs
  sameSize x y = size x == size y
```

6.2 Sidenote: Restrictions on Instances

The Haskell 2010 allows only a very specific type of class instance. Let's look at some instances that aren't allowed. The examples use the `Size` class:

```
class Size a where
  size :: a -> Int
```

We saw a `Size [a]` instance above. Why not define an instance just for lists of booleans?

```
instance Size [Bool] where
  size bs = length (filter id bs)   -- count Trues
```

```
error:
    • Illegal instance declaration for ‘Size [Bool]’
        (All instance types must be of the form (T a1 ... an)
         where a1 ... an are *distinct type variables*,
         and each type variable appears at most once in the instance he
         Use FlexibleInstances if you want to disable this.)
    • In the instance declaration for ‘Size [Bool]’
```

Dang. As the error tries to tell us, we can only define instances where all type parameters are *different* type variables. That is, we can define `instance Size (Either a b)` but we can't define:

- `instance Size (Either String a)` – since `String` is not a type variable
- `instance Size (Either a a)` – since the type variables aren't different
- `instance Size [[a]]` – since `[a]` is not a type variable

Why is this? This rule guarantees that it's simple for the compiler to look up the correct type class instance. It can just look at what the *top-level type constructor* is, and then pick an instance.

The GHC Haskell implementation has many extensions to the type class system – we'll get back to some of them in part 2 of this course.

6.3 Default Implementations

Did you notice how in the previous example we gave `sameSize` the same definition in both instances? This is a very common occurrence, and it's why Haskell classes can have *default implementations*. As a first example, here's an `Example` type class for giving example values of types.

```
class Example a where
  example :: a           -- the main example for the type `a`
  examples :: [a]        -- a short list of examples
  examples = [example]   -- ...defaulting to just the main example

instance Example Int where
  example = 1
  examples = [0,1,2]

instance Example Bool where
  example = True
```

Here's how `Example` works. Note how the default implementation of `examples` got used in the `Bool` case but not in the `Int` case. Also note the need for explicit type signatures to tell GHCi which instance we're interested in. Without them, we would get an "Ambiguous type variable" error.

```
Prelude> example :: Bool
True
Prelude> example :: Int
1
Prelude> examples :: [Bool]
[True]
Prelude> examples :: [Int]
[0,1,2]
```

The standard type classes use lots of default implementations to make implementing the classes easy. Here is the standard definitions for `Eq` (formatted for readability).

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

  (/=) ::  a -> a -> Bool
  x /= y  = not (x == y)
```

Note how both operations have a default implementation in terms of the other. This means we could define an `Eq` instance with no content at all, but the resulting functions would just recurse forever. In practice, we want to define at least one of `==` and `/=`.

When there are lots of default implementations, it can be hard to know which functions you need to implement yourself. For this reason class documentation usually mentions the *minimal complete definition*. For `Eq`, the docs say "Minimal complete definition: either `==` or `/=`."

Let's look at `Ord` next. `Ord` has 7 operations, all with default implementations in terms of each other. By the way, note the quirky way of defining multiple type signatures at once. It's okay, it's a feature of Haskell, this is how `Ord` is defined in the standard. (We'll get back to what the `(Eq a) =>` part means soon.)

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

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

  x <= y  = compare x y /= GT
  x <  y  = compare x y == LT
  x >= y  = compare x y /= LT
  x >  y  = compare x y == GT

  max x y | x <= y    =  y
          | otherwise =  x
  min x y | x <= y    =  x
          | otherwise =  y
```

With this definition it's really hard to know what the minimal complete definition is. Luckily the docs tell us "Minimal complete definition: either compare or <=."

As a final word on default implementations, if there is never a need to override the default definition, the function can be moved out of the class for simplicity. Consider a class like `Combine` below:

```
class Combine a where
  combine :: a -> a -> a
  combine3 :: a -> a -> a -> a
  combine3 x y z = combine x (combine y z)
```

It's hard to think of a case where `combine3` would be given any other definition, so why not move it out of the class:

```
class Combine a where
  combine :: a -> a -> a

combine3 :: Combine a => a -> a -> a -> a
combine3 x y z = combine x (combine y z)
```

As an example, here are the `Eq` and `Ord` instances for a simple pair type. Note how the definition uses the minimal complete definition rules by only defining == and <=.

```
data IntPair = IntPair Int Int
  deriving Show

instance Eq IntPair where
  IntPair a1 a2 == IntPair b1 b2  =  a1==b1 && a2==b2

instance Ord IntPair where
  IntPair a1 a2 <= IntPair b1 b2
     | a1<b1     = True
     | a1>b1     = False
     | otherwise = a2<=b2
```

```
*Main> (IntPair 1 2) < (IntPair 2 3)
True
*Main> (IntPair 1 2) > (IntPair 2 3)
False
*Main> compare (IntPair 1 2) (IntPair 2 3)
LT
```

```
*Main Data.List> sort [IntPair 1 1,IntPair 1 4,IntPair 2 1,IntPair 2
         2]
[IntPair 1 1,IntPair 1 4,IntPair 2 1,IntPair 2 2]
```

6.4 Useful Stuff

6.4.1 Deriving

As we've seen many times already, `deriving` is a way to get automatically generated class instances. The `Read` and `Show` classes should pretty much always be derived to get the standard behaviour. The derived instance for `Eq` is typically what you want. It requires constructors and fields to match.

The derived `Ord` instance might not be what you want. It orders constructors left-to-right, and then compares fields inside constructors left-to-right. An example:

```
data Person = Dead | Alive String Int
  deriving (Show, Eq, Ord)
```

```
Prelude> Dead < Alive "Bob" 35                  -- constructors are
         ordered left-to-right
True
Prelude> Alive "Barbara" 35 < Alive "Clive" 17  -- names are
         compared before ages
True
Prelude> Alive "Clive" 17 < Alive "Clive" 30    -- finally, ages
         are compared if names match
True
```

6.4.2 Asking GHCi About Classes

You can use the `:info` command in GHCi to get the contents and instances of a class. These days the info even includes the minimal complete definition (see the MINIMAL pragma). For example:

```
Prelude> :info Num
class Num a where
  (+) :: a -> a -> a
  (-) :: a -> a -> a
  (*) :: a -> a -> a
  negate :: a -> a
```

```
  abs :: a -> a
  signum :: a -> a
  fromInteger :: Integer -> a
  {-# MINIMAL (+), (*), abs, signum, fromInteger, (negate | (-)) #-}
    -- Defined in ‘GHC.Num’
instance Num Word -- Defined in ‘GHC.Num’
instance Num Integer -- Defined in ‘GHC.Num’
instance Num Int -- Defined in ‘GHC.Num’
instance Num Float -- Defined in ‘GHC.Float’
instance Num Double -- Defined in ‘GHC.Float’
```

6.5 Hierarchies

Both classes and instances can form *hierarchies*. This means that a class or instance depends on another class or instance.

6.5.1 Instance Hierarchies

Let’s try to define an `Eq` instance for a simple pair type:

```
data Pair a = MakePair a a
  deriving Show

instance Eq (Pair a) where
  (MakePair x y) == (MakePair a b)   =   x==a && y==b
```

```
error:
    • No instance for (Eq a) arising from a use of ‘==’
      Possible fix: add (Eq a) to the context of the instance
         declaration
    • In the first argument of ‘(&&)’, namely ‘x == a’
      In the expression: x == a && y == b
      In an equation for ‘==’:
          (MakePair x y) == (MakePair a b) = x == a && y == b
```

The compiler is trying to tell us that our `Eq (Pair a)` instance needs an `Eq a` instance to work. How can we compare pairs of values of type `a` if we can’t compare values of type `a`? To solve this we need to add a type constraint to the instance declaration, just like we’ve added type constraints to function definitions.

```
instance Eq a => Eq (Pair a) where
```

```
  (MakePair x y) == (MakePair a b)    =    x==a && y==b
```

Now we can compare pairs, as long as the element type is comparable. However, we can’ t compare, say, pairs of functions, since functions don’ t have an Eq instance.

```
Prelude> MakePair 1 1 == MakePair 1 1
True
Prelude> MakePair reverse reverse == MakePair reverse reverse

<interactive>:15:1: error:
    • No instance for (Eq ([a0] -> [a0])) arising from a use of ‘==’
        (maybe you haven't applied a function to enough arguments?)
    • In the expression:
        MakePair reverse reverse == MakePair reverse reverse
      In an equation for ‘it’:
          it = MakePair reverse reverse == MakePair reverse reverse
```

Let’ s continue with another example. Here’ s a simple type class and an instance

```
class Check a where
  check :: a -> Bool

instance Check Int where
  check x = x > 0
```

Now we can write a function that checks a list. We use the standard library function `and :: [Bool] -> Bool` that checks if a list is all `True`s.

```
checkAll :: Check a => [a] -> Bool
checkAll xs = and (map check xs)
```

In order to turn this into a `Check [a]` instance, we need to add a constraint to the instance declaration. Our `Check [a]` is based on the `Check a` instance.

```
instance Check a => Check [a] where
  check xs = and (map check xs)
```

This means that our `Check [a]` instance is only valid when there is a corresponding `Check a` instance. For example, if we try to invoke a `Check [Bool]` instance, we get an error about the missing `Check Bool` instance:

```
Prelude> check [True,False]

<interactive>:1:1: error:
    • No instance for (Check Bool) arising from a use of ‘check’
    • In the expression: check [True, False]
      In an equation for ‘it’: it = check [True, False]
```

Also, if we try to define `Check [a]` instance without the constraint, we get an error (with a pretty good suggestion!)

```
    • No instance for (Check a) arising from a use of ‘check’
      Possible fix:
        add (Check a) to the context of the instance declaration
```

If you think about it, this instance hierarchy allows us to circumvent the limitation that we can’ t make a `Check [Int]` instance.

Finally, sometimes multiple constraints are needed. Consider for example the `Eq` instance for `Either`:

```
instance (Eq a, Eq b) => Eq (Either a b) where
  Left x  == Left y   =  x==y
  Right x == Right y  =  x==y
  _       == _        =  False
```

6.5.2 Class Hierarchy

Respectively, a class can depend on another class. This is useful for instance when you want to use functions from another class in your default implementations:

```
class Size a where
  size :: a -> Int

class Size a => SizeBoth a where
  sizeBoth :: a -> a -> Int
  sizeBoth x y = size x + size y
```

In cases like this we say `SizeBoth` is a *subclass* of `Size`. Note again the confusion with object oriented programming. Examples of subclasses in the standard library include:

```
class Eq a => Ord a where
  ...
class Num a => Fractional a where
  ...
```

Another way to look at subclasses is that if you have a `class Main a => Sub a`, you must provide an `instance Main MyType` in order to be able to declare `instance Sub MyType`.

6.6 Quiz

What are the functions in the `Eq` class?

a. `(==), (/=)`
b. `(==)`
c. `(==), (<), (>)`

For which of the following classes can we get automatic instances with `deriving`?

a. `Num`
b. `Ord`
c. `Size`

Which of the following instance declarations is legal?

a. `instance Eq Maybe`
b. `instance Eq (a,a)`
c. `instance Eq (Maybe Int)`
d. `instance Eq (a,b)`

Given the following definition of the class `BitOperations`

```
class BitOperations a where
  bitNot :: a -> a
  bitNot x = bitNand x x
  bitAnd :: a -> a -> a
  bitAnd x y = bitNot (bitOr (bitNot x) (bitNot y))
  bitOr :: a -> a -> a
  bitOr x y = bitNot (bitAnd (bitNot x) (bitNot y))
  bitNand :: a -> a -> a
  bitNand x y = bitNot (bitAnd x y)
```

which set of operations is *not* a minimal complete definition of `BitOperations`?

a. `bitNand, bitAnd`
b. `bitAnd, bitOr`
c. `bitAnd, bitNot`
d. `bitNot, bitOr`

The declaration `instance Num a => Eq (Pair a)` tells me that

a. All instances of `Num` are instances of `Eq`
b. `Pair a` is an instance of `Eq` if `a` is an instance of `Num`
c. The instance `Eq (Pair a)` inherits the instance `Num a`

The declaration `class Num a => Fractional a` tells me that

a. All instances of `Fractional` must be instances of `Num`
b. All instances of `Num` must be instances of `Fractional`
c. If I define an instance for `Fractional`, I also get an instance for `Num`
d. If I define an instance for `Num`, I also get an instance for `Fractional`

6.7 Exercises

- Set6: defining classes and instances

7 Lecture 7: New Constellations

This lecture offers an introduction to *design patterns* for *typed functional programming*. These patterns are both useful when writing Haskell programs, and offer a nice arena for practicing skills from the previous lectures.

7.1 Modeling with Boxes

Sometimes you don't need a new type, but instead can just reuse a standard type. For example, repesenting car register plate numbers with `String`. However, if your code is full of `Strings`, it can be easy to accidentally mix up e.g. a car's model and registration in a function like
`registerCar :: String -> String -> CarRegistry -> CarRegistry`.

For situations like this it's common to create a new type that just contains a `String` (a "boxed" string):

```
data Plate = Plate String
```

```
  deriving (Show, Eq)
```

We can now give `registerCar` a slightly nicer type, `String -> Plate -> CarRegistry -> CarRegistry`. Additionally, we can restrict the operations that are possible on `Plates` to a subset of those that are possible on strings. For example, there is no need to combine the register plate numbers of two cars. Thus we don't need to offer a function `concatPlates :: Plate -> Plate -> Plate`. We can also define a *smart constructor* for `Plate` that checks that the register number is in the correct format:

```
parsePlate :: String -> Maybe Plate
parsePlate string
  | correctPlateNumber string  =  Just (Plate string)
  | otherwise                  =  Nothing
```

Here's another example: representing money. If we just store money as `Int`s, the compiler won't protect us from mistakes like multiplying money with money. If instead we implement our own `Money` type that wraps `Int`, we get type safety. Additionally, we can encapsulate the fact that money is represented as an integer amount of cents.

```
data Money = Money Int
  deriving Show

renderMoney :: Money -> String
renderMoney (Money cents) = show (fromIntegral cents / 100)

(+!) :: Money -> Money -> Money
(Money a) +! (Money b) = Money (a+b)

scale :: Money -> Double -> Money
scale (Money a) x = Money (round (fromIntegral a * x))

addVat :: Money -> Money
addVat m = m +! scale m 0.24



renderMoney (Money 100 +! Money 150)
  ==> "2.5"

scale (Money 299) 0.24
  ==> Money 72
```

```
addVat (Money 299)
  ==> Money 371
```

Note! If you're familiar with Object-Oriented Programming, this is a bit like encapsulation.

7.2 Modeling with Cases

Haskell's algebraic datatypes are really powerful at modeling things based on *cases*. It's often useful to think of types as defining the set of possible cases, and functions *handling* those cases (often via pattern matching). Let's look at two examples.

Since it's so easy to define custom types in Haskell, it's quite convenient to use more descriptive types instead of booleans or strings. Consider a list of persons. In some other language if you wanted to sort the persons into ascending order by name you might use a call like `sortPersons(persons, "name", true)`. In Haskell you can do this instead:

```
data Person = Person {name :: String, age :: Int}
  deriving Show

data SortOrder = Ascending | Descending
data SortField = Name | Age

sortByField :: SortField -> [Person] -> [Person]
sortByField Name ps = sortBy (comparing name) ps
sortByField Age ps = sortBy (comparing age) ps

sortPersons :: SortField -> SortOrder -> [Person] -> [Person]
sortPersons field Ascending ps = sortByField field ps
sortPersons field Descending ps = reverse (sortByField field ps)

persons = [Person "Fridolf" 73, Person "Greta" 60, Person "Hans" 65]
```

```
sortPersons Name Ascending persons
  ==> [Person {name = "Fridolf", age = 73},Person {name = "Greta",
         age = 60},Person {name = "Hans", age = 65}]
sortPersons Age Descending persons
  ==> [Person {name = "Fridolf", age = 73},Person {name = "Hans",
         age = 65},Person {name = "Greta", age = 60}]
```

Note how you can't accidentally typo the field name (unlike with strings), and how you don't need to remember whether `true` refers to ascending or descending

order.

Let's move on to the next example. Many Haskell functions don't work with empty lists (consider `head []`). If you're writing code that needs to track whether lists are possibly empty or guaranteed to not be empty, you can use the `NonEmpty` type from the Data.List.NonEmpty module.

Consider the definition of `NonEmpty`:

```
data NonEmpty a = a :| [a]
```

Here the type represents a *lack of* cases. The type `NonEmpty a` will always consist of a value of type `a`, and some further `a`s, collected in a list. Here are some example values of `NonEmpty Int`:

```
1 :| [2,3,4]
1 :| []
```

By the way, this is also an example of an *infix constructor*. We've already met another infix constructor earlier, the list constructor `(:)`. Any operator that begins with a colon (the `:` character) can be used as an infix constructor. We can pattern match on `(:|)` just like on `(:)`, as you'll see in the examples below.

Here are the functions that convert between normal lists and nonempty lists. Note how we can't have a function `[a] -> NonEmpty a`, but must instead use `Maybe` to represent the possibility that the list was, indeed, empty. Note also how `toList` has only one equation, we can't have a `toList []` situation due to the type `NonEmpty`.

```
nonEmpty :: [a] -> Maybe (NonEmpty a)
nonEmpty [] = Nothing
nonEmpty (x:xs) = Just (x :| xs)

toList :: NonEmpty a -> [a]
toList (x :| xs) = x : xs
```

```
nonEmpty [1,2,3]     ==>  Just (1 :| [2,3])
nonEmpty [1]         ==>  Just (1 :| [])
nonEmpty []          ==>  Nothing
toList (1 :| [2,3])  ==>  [1,2,3]
```

Here are `head` and `last` implemented for `NonEmpty`:

```
neHead (x :| _) = x
neLast (x :| []) = x
neLast (_ :| xs) = last xs
```

```
neHead (1:|[2,3])  ==>  1
neLast (1:|[2,3])  ==>  3
```

By the way, these functions are available as `Data.List.NonEmpty.head` and `Data.List.NonEmpty.last` along with many other useful functions.

In summary, if you write types that represent all possible cases for your values, and then write functions that handle those cases, your code will be simple and correct.

7.3 Monoids

A pattern that comes up surprisingly often in functional programming is the *monoid* (not to be confused with a *monad*!). Explanations of monoids are often very mathematical, but the idea is simple: combining things.

7.3.1 Associative Operations

Many functions and operators we use are *associative*. This is just a fancy way of saying they don't need parentheses. For example, all of these expressions have the value 16 because addition is associative:

```
(1 +  3) + (5 + 7)
 1 + (3  + (5 + 7))
 1  + 3  +  5 + 7
```

Examples of associative operations are easy to come by in Haskell. For example the `++` operator for catenating lists is associative: it doesn't matter whether you do `([1] ++ [2,3]) ++ [4]` or `[1] ++ ([2,3] ++ [4])` – the result is `[1,2,3,4]`.

Another great example is the function composition operator. Both `(head . tail) . tail` and `head . (tail . tail)` compute the third element of a list.

However not all operators are associative. The most familiar examples are subtraction and exponentiation. `(1-2)-3` is `-4` but `1-(2-3)` is `2`. Similarly, `(2^3)^2` is `64` while `2^(3^2)` is 512. One needs to be careful with parentheses when using operators that are not associative.

Another operator that's not associative is the list constructor, `(:)`. This time the reason is even more fundamental: while `True:(False:[])` is ok, `(True:False):[]` does not even type! In order for an operation to be associative, it needs to take two arguments of the same type.

In addition to operators, functions can also be associative. The syntax looks a bit different, but a function `f` is associative if these are the same:

```
f x (f y z)
f (f x y) z
```

Two widely-used associative functions are the `min` and `max` functions:

```
min 2 (min 1 3) ==> 1
min (min 2 1) 3 ==> 1

max 2 (max 1 3) ==> 3
max (max 2 1) 3 ==> 3
```

7.3.2 Semigroups

Mathematically speaking, an associative function (or operator) forms a *semigroup*. Haskell has a type class `Semigroup` (defined in the module `Data.Semigroup`) that can be used when a type has one clear associative operation.

```
class Semigroup a where
  -- An associative operation.
  (<>) :: a -> a -> a
```

Lists are an instance of `Semigroup` with `(++)` as `(<>)`:

```
[1] <> [2,3] <> [4]  ==>  [1,2,3,4]
```

Types that have multiple different associative operators usually aren't made an instance of Semigroup. An example is `Int`, which has many associative functions like `+`, `*` and `max`. Instead, the Haskell standard library uses boxing (see earlier in this lecture). Here are the definitions for `Sum` and `Product`:

```haskell
data Sum a = Sum a
instance Num a => Semigroup (Sum a) where
  Sum a <> Sum b  =  Sum (a+b)

data Product a = Product a
instance Num a => Semigroup (Product a) where
  Product a <> Product b   =  Product (a*b)
```

By the way, this is another benefit of boxing things: being able to declare different type class instances!

Note how the `Num a` constraint lets us use `Num` operations like `+` and `*` on the contained type `a`. We can have values like `Sum "abc" :: Sum String`, but they won't have a `Semigroup` instance!

Similarly, we have box types `Min` and `Max`. Let's play around in GHCi a bit:

```haskell
Prelude> import Data.Semigroup
Prelude Data.Semigroup> Product (2::Int) <> Product 3 <> Product 1
Product {getProduct = 6}
Prelude Data.Semigroup> Sum 3 <> Sum 5 <> Sum 7
Sum {getSum = 15}
Prelude Data.Semigroup> Product 2 <> Product 3 <> Product 1
Product {getProduct = 6}
Prelude Data.Semigroup> Min 4 <> Min 3 <> Min 5
Min {getMin = 3}
Prelude Data.Semigroup> Max 4 <> Max 3 <> Max 5
Max {getMax = 5}
```

7.3.3 Finally, Monoids

If we listen to the mathematicians for a moment again, a *monoid* is a semigroup with a *neutral element*. A neutral element is a zero: an element that does nothing when combined with other elements. Here are some examples:

```haskell
-- 0 is the neutral element of (+)
3 + 0        ==>  3
0 + 3        ==>  3

-- 1 is the neutral element of (*)
1 * 5        ==>  5
5 * 1        ==>  5

-- [] is the neutral element of (++)
```

```
[] ++ [1,2]  ==>  [1,2]
[1,2] ++ []  ==>  [1,2]
```

The Haskell type class `Monoid` (from the module `Data.Monoid`) represents monoids.

```
class Semigroup a => Monoid a where
  -- The neutral element
  mempty :: a
```

Here are the `Monoid` instances corresponding to our three examples of neutral elements:

```
instance Num a => Monoid (Sum a) where
  mempty = Sum 0

instance Num a => Monoid (Product a) where
  mempty = Product 1

instance Monoid [] where
  mempty = []
```

So, what is a monoid for a programmer? A type forms a monoid if there's a way of combining two elements of the type together so that parenthesis don't matter, and there's a also an "empty element" that can be combined with things without changing them. When thought of like this, monoids come up in programming quite often!

7.3.4 Why?

What use is this `Monoid` class? Can't we just write `1 + 2` instead of `Sum 1 <> Sum 2`? We can, yes, but some library functions work on all `Monoid` types.

The reason we want both a neutral element and an associative binary operator is that those are the exact two things we need in order to *reduce* or *fold* multiple elements into one value. This is the job of:

```
mconcat :: Monoid a => [a] -> a
```

Sidenote: one way to define `mconcat` is `foldr (<>) mempty`. Do you remember `foldr`?

Let's look at why we need the properties of `Monoid` to implement `mconcat`. Firstly, we need `mempty` to handle empty lists:

```
mconcat [] :: Sum Int          ==>  Sum 0
```

Secondly, we need associativity to be able to reduce a list `[x,y,z]` to a unique value. If `<>` were not associative, we would have two possible values for `mconcat [x,y,z]`, namely `(x<>y)<>z` and `x<>(y<>z)`.

The most useful `Monoid` function is `foldMap`:

```
foldMap :: (Foldable t, Monoid m) => (a -> m) -> t a -> m
```

That type signature looks scary, but concrete cases are simpler:

```
foldMap Max [1::Int,4,2]  ==>  Max 4
foldMap Product [1::Int,4,2]  ==>  Product 8
-- We need the ::Int to avoid an "Ambiguous type variable" error
         when printing the result
```

Let's break down that type. We know that an example of a `Foldable t => t a` type is `[a]`, so we can rewrite the type as

```
foldMap' :: Monoid m => (a -> m) -> [a] -> m
```

We can build this function out of functions we already know:

```
foldMap' f xs = mconcat (map f xs)
```

Oh, by the way, thanks to the `(Monoid a, Monoid b) => Monoid (a,b)` instance we can even compute the maximum and product in one pass:

```
foldMap (\x -> (Max x, Product x)) [1::Int,4,2]  ==>  (Max 4,
         Product 8)
```

Note, you don't need to use monoids in your own code, but you'll eventually bump into them when using Haskell libraries so it's good to know what they are.

7.3.5 How?

Due to various historical and performance reasons, the definition of the `Monoid` and `Semigroup` classes aren't just

```
class Semigroup a where
  (<>) :: a -> a -> a
class Semigroup a => Monoid a where
  mempty :: a
```

Although you can mostly pretend they are. The actual definitions are:

```
class Semigroup a where
  -- | An associative operation.
  (<>) :: a -> a -> a

  -- Combine elements of a nonempty list with <>
  sconcat :: NonEmpty a -> a
  sconcat as = ... -- default implementation omitted

  -- Combine a value with itself using <>, n times
  stimes :: Integral b => b -> a -> a
  stimes n x = ... -- default implementation omitted
```

```
class Semigroup a => Monoid a where
  mempty  :: a

  mappend :: a -> a -> a
  mappend = (<>)

  -- Combine elements of a list with <>
  mconcat :: [a] -> a
  mconcat = ... -- default implementation omitted
```

As you can see, all the operations except `<>` and `mempty` have default definitions, so a normal `Monoid` instance declaration looks just like this:

```
instance Semigroup MyType where
  x <> y = ...

instance Monoid MyType where
  mempty = ...
```

7.4 Open and Closed Abstractions

A question novice Haskell programmers often ask (or at least should ask!) is: when should I use type classes? This section offers one answer.

Let's look at a concrete example. A vehicle can be either a car or an airplane. We can model this with algebraic datatypes (as we've seen earlier in this chapter), but also with type classes. Here's the datatype version:

```
data Vehicle = Car String | Airplane String

sound :: Vehicle -> String
sound (Car _) = "brum brum"
sound (Airplane _) = "zooooom"
```

Here's the class version. Note how each case gets its own datatype, which are collected together in a type class.

```
data Car = Car String
data Airplane = Airplane String

class VehicleClass a where
  sound :: a -> String

instance VehicleClass Car where
  sound (Car _) = "brum brum"

instance VehicleClass Airplane where
  sound (Airplane _) = "zooooom"
```

What is the difference between these solutions? The data-based solution is *closed*, meaning the set of cases is fixed and we can handle all of them in one place. The class-based solution is *open*, meaning we can add new cases, even in other modules.

An open abstraction is nice when we want extensibility. In the class-based solution, another module could define a bike:

```
data Bike = Bike String

instance VehicleClass Bike where
  sound (Bike _) = "whirrr"
```

A closed abstraction is good when we want to know that we've handled all cases, consider for example the function `canCollide` which checks whether two vehicles can collide:

```
canCollide :: Vehicle -> Vehicle -> Bool
canCollide (Car _)      (Car _)      = True
canCollide (Airplane _) (Airplane _) = True
canCollide _            _            = False
```

This would be very hard to implement reliably in the class-based solution. Consider for example how collision checks between `Bikes` and `Cars` would get handled.

7.5 Modeling with Languages

Sometimes it's useful to implement a mini programming language for describing parts of your software. The fancy term for these is an *Embedded Domain-Specific Language (EDSL)*. Haskell is well suited to modeling and interpreting languages. The expressions of the language are represented using (often recursive) algebraic data types. The language can be *interpreted* (that is, evaluated or run) by a recursive function.

Here's an example of a language for describing price computations for products in a web shop.

```
data Discount = DiscountPercent Int          -- A percentage discount
              | DiscountConstant Int         -- A constant discount
              | MinimumPrice Int             -- Set a minimum price
              | ForCustomer String Discount -- Discounts can be
         conditional
              | Many [Discount]              -- Apply a number of
         discounts in row
```

The language is interpreted by the function `applyDiscount` that takes a customer name, a price, a discount, and returns a price.

```
applyDiscount :: String -> Int -> Discount -> Int
applyDiscount _        price (DiscountPercent percent) = price -
         (price * percent) `div` 100
applyDiscount _        price (DiscountConstant discount) = price -
         discount
applyDiscount _        price (MinimumPrice minPrice) = max price
         minPrice
applyDiscount customer price (ForCustomer target discount)
    | customer == target  = applyDiscount customer price discount
```

```
    | otherwise           = price
applyDiscount customer price (Many discounts) = go price discounts
  where go p [] = p
        go p (d:ds) = go (applyDiscount customer p d) ds
```

Here we apply a discount chain of -50%, -$30 with a minimum price of $35:

```
applyDiscount "Bob" 120 (DiscountPercent 50)
  ==> 60
applyDiscount "Bob" 60 (DiscountConstant 30)
  ==> 30
applyDiscount "Bob" 30 (MinimumPrice 35)
  ==> 35
applyDiscount "Bob" 120 (Many [DiscountPercent 50, DiscountConstant
         30, MinimumPrice 35])
  ==> 35
```

Here we have different discounts for Ssarah and Yvonne:

```
applyDiscount "Yvonne" 100 (Many [ForCustomer "Yvonne"
         (DiscountConstant 10), ForCustomer "Ssarah"
         (DiscountConstant 20)])
  ==> 90
applyDiscount "Ssarah" 100 (Many [ForCustomer "Yvonne"
         (DiscountConstant 10), ForCustomer "Ssarah"
         (DiscountConstant 20)])
  ==> 80
```

As you can see, even a simple `Discount` type can generate complex behaviours because it is self-referential (recursive). Using `Discount` we are able to represent the discount logic of our webshop as *data* instead of writing code.

There are multiple reasons for representing logic as data instead of code. Unlike code, data can easily be stored in a file or database, or even transmitted over the network. We can also use the same data for multiple purposes, for exaple we could visualize the discount chains in an administration user interface.

7.6 Exercises

- Set7

8 Lecture 8: The Aftertaste

8.1 A Taste of IO

This course has been centered around pure functional programming. We've done lots of arithmetic, reversed lists, worked with binary trees, but so far we haven't been able to affect the world outside our GHCi.

Things like reading input, writing to a file, or talking over the network are *side effects*. Side effects can't be represented with pure functional code. A function like

```
readInputFromTheUser :: String -> String
```

can't be pure, because if it were, `readInputFromUser "What is your name?"` would always have to return the same result. However, representing side-effects and impurity in a pure language *is* possible. There are many ways of doing it, and the Haskell way is to use *Monads*.

Monads are reputedly difficult to understand. That is probably because they are so abstract. I think it's best to focus on practical and concrete cases first. Here's a taste of the `IO` Monad, which you can use for all sorts of side effects in Haskell.

Let's start!

```
Prelude> :t getLine
getLine :: IO String
Prelude> line <- getLine



another line



Prelude> :t line
line :: String
Prelude> line
"another line"
Prelude> reverse line
"enil rehtona"
```

What we've seen here is the *IO action* `getLine`. It has type `IO String`. This means that GHCi can *execute* the action to produce a value of type `String`. When we enter

`line <- getLine` into GHCi, we mean:

> Execute the IO action `getLine`, and give the result the name `line`.

After we've received `line`, it's a pure `String` value and we can work with it normally.

Some IO actions take parameters. For example `putStrLn :: String -> IO ()` takes a `String` and returns an `IO` action that prints that string. The `()` type is a special type that only has one value, `()`. In this case `IO ()` means that this IO always produces the same empty value `()`. You can run IO actions by just

```
Prelude> :t putStrLn
putStrLn :: String -> IO ()
Prelude> :t putStrLn "hello"
putStrLn "hello" :: IO ()
Prelude> val <- putStrLn "hello"
hello
Prelude> val
()
```

If you don't need the return value of an IO action, you can run it *in GHCi* without the `<-`:

```
Prelude> putStrLn "hello"
hello
```

You can build your own IO actions by combining other actions with *do-notation*. A `do` block lists IO actions that are executed in order.

```
printTwoThings :: IO ()
printTwoThings = do
  putStrLn "Hello!"
  putStrLn "How are you?"

greet :: IO ()
greet = do
  putStrLn "What's your name?"
  name <- getLine
  putStrLn ("Hello, " ++ name)
```

```
Prelude> printTwoThings
Hello!
```

```
How are you?
Prelude> greet
What's your name?
Seraphim
Hello, Seraphim
```

8.1.1 What About Purity?

It feels as if we can just do side effects where ever we want with these IO actions. However, it's important to remember the distinction between *defining* an IO action and *executing* it.

Let's try to print while mapping over a list

```
printAndIncrement :: Int -> Int
printAndIncrement x = x+1
  where action = putStrLn "got a number!"
```

```
Prelude> map printAndIncrement [1,2,3]
[2,3,4]
```

This didn't print anything, because even though we defined our `action`, it wasn't given to GHCi for execution. Because `printAndIncrement` returns an `Int`, it can't return an action. Ok, let's try another approach:

```
Prelude> length (map putStrLn ["string1","string2"])
2
```

That didn't print anything either! Let's see why:

```
Prelude> :t map putStrLn ["string1","string2"]
map putStrLn ["string1","string2"] :: [IO ()]
Prelude> :t length (map putStrLn ["string1","string2"])
length (map putStrLn ["string1","string2"]) :: Int
```

We generated a list of IO actions and computed the value of the list. Defining IO actions is pure, it's *running* them that causes side effects. Since the type of our expression was `Int`, no `IO` actions could land in GHCi and be executed.

If we instead return an IO action, it does get run:

```
Prelude> :t head (map putStrLn ["string1","string2"])
head (map putStrLn ["string1","string2"]) :: IO ()
Prelude> head (map putStrLn ["string1","string2"])
string1
```

Here too, the code that produces the action `putStrLn "string1"` is pure, it's only after the IO action is executed by GHCi that we see the printed string. And as you can see, the other IO action, `putStrLn "string2"`, never got run.

If this feels complicated, don't worry. We'll get back to this on part 2 of the course.

8.1.2 What About Haskell Programs?

We know that GHCi can run IO actions. What about actual Haskell programs? The way Haskell programs work is that the IO action called `main` gets executed when the program is run. Recall our example program from Lecture 1.

```
module Gold where

-- The golden ratio
phi :: Double
phi = (sqrt 5 + 1) / 2

polynomial :: Double -> Double
polynomial x = x^2 - x - 1

f x = polynomial (polynomial x)

main = do
  print (polynomial phi)
  print (f phi)
```

Here we see some pure code and a `main` IO action that prints two things (`print` is just `putStrLn` combined with `show`).

We can place this code in a file called `Gold.hs`, compile it into an executable, and run it:

```
$ ghc -main-is Gold Gold.hs
[1 of 1] Compiling Gold             ( Gold.hs, Gold.o )
Linking Gold ...
$ ./Gold
```

```
0.0
-1.0
```

8.2 Summary

So far, we've learned about Haskell's syntax and types, quite a bit of functional programming and about some language features like type classes.

We've also seen some type-oriented programming, and even gotten a taste of I/O in Haskell.

Now you know how to write a real computer program in Haskell, but there's still much to learn.

8.3 What Next?

Part 2 of the course is now out! Part 2 will cover topics like Monads, IO and how Haskell works under the hood. We'll also get to do some real world programming with networks and databases. Oh and testing in Haskell is also covered.

If you don't feel like jumping into part 2 right now, here are some other Haskell resources you should be able to follow now:

- Real World Haskell - free e-book on advanced topics
- What I Wish I Knew When Learning Haskell
- The Haskell Wikibook
- The Haskell Website has links to talks and presentations
- Haskell mini-patterns handbook - more stuff like Lecture 7

I also recommend working on some programming problems in Haskell, like the ones from:

- Advent of Code – nice varying puzzles, start off easy and get harder
- Sphere Online Judge – algorithm problems, good variety of difficulties
- Project Euler – mathematical programming puzzles

You can also keep extending your final project and maybe generate some cool art in Haskell.

In any case – thank you so much for tagging along, and we hope you have a great rest of the year!

8.4 Final Project: Graphics

Open up the exercise file `Set8.hs` and follow the instructions there. Have fun!

8.5 Acknowledgements

This course was made possible by Nitor who donated hours and hours of Joel's working time for this project. Thank you! Check out our open positions if you're interested in working somewhere that values continuous learning.

Thanks to the whole Haskell Mooc team, especially

- John Lång for help on the material
- Antti Laaksonen for setting up the course and helping with arrangements