Part 2: Datatypes and Functions

Introduction to Haskell

Andres Löh

Well-Typed
The Haskell Consultants

Goals

- Learn how to define functions
- How types help you while programming
- Syntax of Haskell
- How to define and use datatypes
- Overview of base types and datatypes

Structure of a Haskell program

- Haskell programs comprise one or more **modules**. One module per file. Main module is always called `Main`.
- Modules consist of **declarations**. Declarations introduce datatypes, functions and constants, type classes and instances.
- We will focus on functions and constants first, then datatypes. Later type classes and instances.

Declaring new functions and constants

```
length :: [a] -> Int
length []       = 0
length (x : xs) = 1 + length xs
```

Well-Typed

Declaring new functions and constants

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

- The name being introduced.

Declaring new functions and constants

```
length :: [a] -> Int
length []       = 0
length (x : xs) = 1 + length xs
```

- The name being introduced.
- Type signature (optional, but recommended).

Declaring new functions and constants

```
length :: [a] -> Int
length []       = 0
length (x : xs) = 1 + length xs
```

- The name being introduced.
- Type signature (optional, but recommended).
- One or more equations defining the function.

Declaring new functions and constants

```
length :: [a] -> Int
length []       = 0
length (x : xs) = 1 + length xs
```

- The name being introduced.
- Type signature (optional, but recommended).
- One or more equations defining the function.
- The `=` symbol separates the **left hand sides** from the **right hand sides**.

Declaring new functions and constants

```
length :: [a] -> Int
length []       = 0
length (x : xs) = 1 + length xs
```

- The name being introduced.
- Type signature (optional, but recommended).
- One or more equations defining the function.
- The `=` symbol separates the **left hand sides** from the **right hand sides**.
- Cases are distinguished by **patterns**.

Declaring new functions and constants

```
length :: [a] -> Int
length []       = 0
length (x : xs) = 1 + length xs
```

- The name being introduced.
- Type signature (optional, but recommended).
- One or more equations defining the function.
- The `=` symbol separates the **left hand sides** from the **right hand sides**.
- Cases are distinguished by **patterns**.
- On the right hand side, we have **expressions**.

Declarations, patterns, expressions

Informally:

- A (function or constant) **declaration** binds a (new) identifier to an expression.
- A **pattern** occurs as an argument to an identifier on the left hand side of a declaration. It introduces names that are available on the right hand side. Patterns can be matched against actual function arguments. Matches can fail or succeed.
- **Expressions** occur on the right hand side of a function definition. Expressions can be evaluated.

Types

Every expression must have a **type** in Haskell – otherwise it will be rejected by the compiler:

- Haskell types can be inferred. There's usually no need for type annotations.
- Use `:t` in GHCi to obtain the inferred type of an expression.
- Type annotations (using `::`) are optional. But if they're given, their correctness is checked.

How to define a function?

There are two main design principles for defining functions:

- by (systematic) pattern matching and recursion,

How to define a function?

There are two main design principles for defining functions:

- by (systematic) pattern matching and recursion,
- by applying a higher-order function (such as composition, `map` , `foldr` , ...) and thereby reducing the problem to smaller subproblems.

In both cases, thinking about the types first **helps** you!

How to define a function?

There are two main design principles for defining functions:

- **by (systematic) pattern matching and recursion,**
- by applying a higher-order function (such as composition, `map` , `foldr` , ...) and thereby reducing the problem to smaller subproblems.

In both cases, thinking about the types first **helps** you!

We will focus on the pattern matching approach first.

Functions on lists

Most functions operate on **structured data**.

Lists are a simple data structure, so they're ideal for learning.

Functions on lists

Most functions operate on **structured data**.

Lists are a simple data structure, so they're ideal for learning.

Recall from the Quick Tour

Lists are defined **inductively**:

- The empty list `[]` is a list.
- Given a single element `y` and a list `ys`, we can construct a new list `y : ys` (pronounced `y` **cons** `ys`).

Functions on lists

Most functions operate on **structured data**.

Lists are a simple data structure, so they're ideal for learning.

Recall from the Quick Tour

Lists are defined **inductively**:

- The empty list `[]` is a list.
- Given a single element `y` and a list `ys`, we can construct a new list `y : ys` (pronounced `y` **cons** `ys`).

We call `[]` and `(:)` the **constructors** of the list datatype.

Well-Typed

Another look at `elem`

Let's try to implement `elem` once more, systematically.

```
elem :: Int -> [Int] -> Bool
```

We **start** with the type.

Do we want to restrict ourselves to `Int` lists? No!

Another look at `elem`

Let's try to implement `elem` once more, systematically.

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

Let's make as few assumptions as possible.

In order to split up the programming problem, let's take a look at the input list . . .

Another look at `elem`

Let's try to implement `elem` once more, systematically.

```
elem :: a -> [a] -> Bool
elem x []         = ...
elem x (y : ys)   = ...
```

There are two cases, one per constructor of the list datatype.

Let's see if we can solve the simple case for `[]` .

Another look at `elem`

Let's try to implement `elem` once more, systematically.

```
elem :: a -> [a] -> Bool
elem x []       = False
elem x (y : ys) = ...
```

Now to the cons-case.

The `ys` is a shorter list – the most natural way to define functions on recursive datatypes is to use recursive functions.

Another look at `elem`

Let's try to implement `elem` once more, systematically.

```
elem :: a -> [a] -> Bool
elem x []       = False
elem x (y : ys) = ...  elem ys  ...
```

Let's try to complete the second case making use of the recursive call.

Another look at `elem`

Let's try to implement `elem` once more, systematically.

```
elem :: a -> [a] -> Bool
elem x []        = False
elem x (y : ys) = x == y || elem x ys
```

Done?

Another look at `elem`

Let's try to implement `elem` once more, systematically.

```
elem :: a -> [a] -> Bool
elem x []       = False
elem x (y : ys) = x == y || elem x ys
```

Oh, we actually need equality on the elements. That seems to be a sensible requirement for `elem`, so let's refine the type ...

Another look at `elem`

Let's try to implement `elem` once more, systematically.

```
elem :: Eq a => a -> [a] -> Bool
elem x []       = False
elem x (y : ys) = x == y || elem x ys
```

Now we're really done.

Another look at `elem`

Let's try to implement `elem` once more, systematically.

```
elem :: Eq a => a -> [a] -> Bool
elem x []       = False
elem x (y : ys) = x == y || elem x ys
```

The systematic development we've just seen generalizes to most functions on lists and most functions on other structured datatypes.

Mapping over a list

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

Start with the type. A function is like any other argument.

Mapping over a list

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

Introduce cases based on the list constructors.

Mapping over a list

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

Solve the simple case first.

Mapping over a list

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

Keep recursion in mind.

Mapping over a list

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

As `f` is a function, we can apply it.

Take a final look.

Everything looks fine.

Another example: `drop` elements from a list

The call `drop n xs` should remove the first `n` elements from `xs`.

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

Another example: `drop` elements from a list

The call `drop n xs` should remove the first `n` elements from `xs`.

```
drop :: Int -> [a] -> [a]
drop n []       = ...
drop n (x : xs) = ...
```

What do we actually want to do if we want to drop `3` elements of an empty list?

Another example: `drop` elements from a list

The call `drop n xs` should remove the first `n` elements from `xs`.

```
drop :: Int -> [a] -> [a]
drop n []       = []
drop n (x : xs) = ...
```

We take a simple approach.

Another example: `drop` elements from a list

The call `drop n xs` should remove the first `n` elements from `xs`.

```
drop :: Int -> [a] -> [a]
drop n []       = []
drop n (x : xs) = ... drop ... xs ...
```

Wait, but what we want to do depends on `n`?

We have multiple options here . . .

Another example: `drop` elements from a list

The call `drop n xs` should remove the first `n` elements from `xs`.

```
drop :: Int -> [a] -> [a]
drop n []       = []
drop 0 (x : xs) = ... drop ... xs ...
drop n (x : xs) = ... drop ... xs ...
```

We can pattern match on an `Int` too …

If cases **overlap**, the first matching case applies.

Another example: `drop` elements from a list

The call `drop n xs` should remove the first `n` elements from `xs` .

```
drop :: Int -> [a] -> [a]
drop n []       = []
drop 0 (x : xs) = x : xs
drop n (x : xs) = ... drop ... xs ...
```

Sometimes, we don’t need to recurse – even though we could.

Another example: `drop` elements from a list

The call `drop n xs` should remove the first `n` elements from `xs`.

```
drop :: Int -> [a] -> [a]
drop n []       = []
drop 0 (x : xs) = []
drop n (x : xs) = drop (n - 1) xs
```

Done.

But what happens with negative numbers as arguments?

Another example: `drop` elements from a list

The call `drop n xs` should remove the first `n` elements from `xs`.

```
drop :: Int -> [a] -> [a]
drop n []       = []
drop n (x : xs) =
  if n <= 0
    then x : xs
    else drop (n - 1) xs
```

We can use `if` - `then` - `else`.

We can include negative numbers now.

If an equation spans multiple lines, the subsequent lines must be indented with respect to the first.

Another example: `drop` elements from a list

The call `drop n xs` should remove the first `n` elements from `xs`.

```
drop :: Int -> [a] -> [a]
drop n []        = []
drop n (x : xs)
   | n <= 0      = x : xs
   | otherwise = drop (n - 1) xs
```

Yet another option: use so-called **guards**.

Can only appear directly after the pattern match. Boolean conditions are tried in order, `otherwise` is just a constant that is defined to be `True`.

Exercise – define the following functions

Append two lists:

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

Hint: Only pattern match on the **first** list (i.e., don't distinguish more cases than needed).

Reverse a list:

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

Hint: Follow the standard pattern, and make use of `(++)` that you have just defined.

Excursion: infix operators

Haskell allows you to create your own operators from a given set of symbols:

- names are either completely symbolic or completely alphanumeric;
- symbolic names are by default used infix, but can be used in prefix notation by surrounding them in parentheses (Example: `(+) 2 3` evaluates to `5`);
- alphanumeric names are by default used prefix, but can be used in infix notation by surrounding them in backquotes (Example: ``8 `mod` 3`` evaluates to `2`);
- you can define the associativity and priority of infix operators by using `infix` , `infixl` , and `infixr` declarations;
- by using `:i` or `:info` in GHCi, you can obtain information about the priority of infix operators.

Next: `filter` ing a list

We want to traverse a list and keep all elements that have a certain property.

Question

How to best express a property of an element?

Next: `filter` ing a list

We want to traverse a list and keep all elements that have a certain property.

Question

How to best express a property of an element?

As a **function** from the element to a `Bool` .

Next: `filter` ing a list

We want to traverse a list and keep all elements that have a certain property.

Question

How to best express a property of an element?

As a **function** from the element to a `Bool` .

Recall from the Quick Tour: a `Bool` is a another datatype with two **constructors**, called `True` and `False` .

Defining `filter`

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

We can now write down the type.

Defining `filter`

```
filter :: (a -> Bool) -> [a] -> [a]
filter p []       = ...
filter p (x : xs) = ... filter ... xs ...
```

Defining `filter`

```
filter :: (a -> Bool) -> [a] -> [a]
filter p []       = []
filter p (x : xs) = ... filter ... xs ...
```

It depends on the outcome of `p x` what we want to do!

We have several options here.

Defining `filter`

```
filter :: (a -> Bool) -> [a] -> [a]
filter p []       = []
filter p (x : xs) = if p x then x : filter p xs
                           else     filter p xs
```

We can use the built-in `if` - `then` - `else` construct.

Note that `Bool` is a type like any other. No need to write `p x == True`. Plain `p x` is simpler and equivalent.

Defining `filter`

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

We can also use **guards** – each guard is tried in order.

Note that

```
otherwise :: Bool
otherwise = True
```

Using `filter`

There are some useful predicates:

```
even, odd         :: Integral a => (a -> Bool)
isUpper, isDigit :: Char -> Bool
```

We can also define our own:

```
positiveInt :: Int -> Bool
positiveInt n = n > 0

palindrome :: [Char] -> Bool
palindrome xs = reverse xs == xs
```

Note that `String` is a (type) synonym for `[Char]`.

Try using `filter` with these predicates.

Excursion: anonymous functions

In practice, functions such as `filter` are often used with **lambda expressions** or **anonymous functions**:

```
filter (\ n -> n > 10 && even n) [1 .. 100]
```

Excursion: anonymous functions

In practice, functions such as `filter` are often used with **lambda expressions** or **anonymous functions**:

```
filter (\ n -> n > 10 && even n) [1 .. 100]
```

A lambda expression is a way to define a function without giving it a name:

```
myPredicate n = n > 10 && even n
```

is just different syntax for

```
myPredicate = \ n -> n > 10 && even n
```

Excursion: operator sections

Partially applied infix operators have yet again special syntax:

```
\ n -> n > 10
```

can be abbreviated to

```
(> 10)
```

Similarly, we can write `(1 +)` or `("Hello " ++)` or `(`div` 5)` .

So it's possible to say

```
filter (> 10) [1 . . 100]
```

Well-Typed

Tuples

Lists vs. tuples

It's time to talk about a new (family of) datatypes: tuples.

- lists are a datatype that collects an arbitrary number of elements; all elements must be of the same type.
- tuples are a family of datatypes that collect a fixed number of elements; each element can have a different type.

Let's look at examples.

Example tuples

Pairing a `Bool` and a `String`:

```
example :: (Bool, String)
example = (True, "yes, it's true")
```

Example tuples

Pairing a `Bool` and a `String`:

```
example :: (Bool, String)
example = (True, "yes, it's true")
```

- The type states it is a pair, and also states the type of each component.

Example tuples

Pairing a `Bool` and a `String`:

```
example :: (Bool, String)
example = (True, "yes, it's true")
```

- The type states it is a pair, and also states the type of each component.
- The corresponding expression has similar syntax, and constructs a pair out of two components.

Example tuples

Pairing a `Bool` and a `String` :

```
example :: (Bool, String)
example = (True, "yes, it's true")
```

- The type states it is a pair, and also states the type of each component.
- The corresponding expression has similar syntax, and constructs a pair out of two components.

Here's a triple:

```
triple :: ([a] -> [a], [b] -> Int, Char)
triple = (reverse, length, 'x')
```

General structure of tuples

For each `n >= 2` , there's a type of `n` -tuples.

- Each of these is a different type.
- Unlike lists (or `Bool`), each tuple type has a single **constructor**, conveniently written using parentheses and commas, with the arguments interspersed (it can also be used in prefix notation, actually).
- We can use pattern matching to extract the components of a tuple (but we do not need to distinguish several cases).

General structure of tuples

For each `n >= 2`, there's a type of `n`-tuples.

- Each of these is a different type.
- Unlike lists (or `Bool`), each tuple type has a single **constructor**, conveniently written using parentheses and commas, with the arguments interspersed (it can also be used in prefix notation, actually).
- We can use pattern matching to extract the components of a tuple (but we do not need to distinguish several cases).

Remarks:

- There are no 1-tuples. Haskell treats `(Int)` and `Int` as the same type.
- The unit type `()` can be seen as a 0-tuple.

Example: selecting the first component of a pair

```
fst :: ...
```

What is the type here?

```
fst :: (Int, Char) -> Int
```

is certainly too specific . . .

Example: selecting the first component of a pair

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

We can accept arbitrary component types. The two components can have different types. But the result type matches the type of the first component.

Now let's apply pattern matching on the input.

Example: selecting the first component of a pair

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

Now `x` is the component of type `a`, and `y` the component of type `b`.

It's nearly trivial to finish the definition.

Example: selecting the first component of a pair

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

And we are done.

Zipping two lists

Sometimes, we have two lists of equal length and want to combine them element by element:

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

We start with the type.

It's a function over (two) lists, so let's apply the standard principle to the first list.

Zipping two lists

Sometimes, we have two lists of equal length and want to combine them element by element:

```
zip :: [a] -> [b] -> [(a, b)]
zip []       ys      = ...
zip (x : xs) ys      = ... zip xs ...
```

We actually need to look at the second list, too. So let's just match on that one as well.

Zipping two lists

Sometimes, we have two lists of equal length and want to combine them element by element:

```
zip :: [a] -> [b] -> [(a, b)]
zip []       []       = ...
zip []       (y : ys) = ...
zip (x : xs) []       = ...
zip (x : xs) (y : ys) = ... zip xs ys ...
```

The first case is easy: if both lists are empty, we return the empty list.

Zipping two lists

Sometimes, we have two lists of equal length and want to combine them element by element:

```
zip :: [a] -> [b] -> [(a, b)]
zip []       []       = []
zip []       (y : ys) = ...
zip (x : xs) []       = ...
zip (x : xs) (y : ys) = ... zip xs ys ...
```

In the final case, we can produce the first element of the resulting list and recurse.

Zipping two lists

Sometimes, we have two lists of equal length and want to combine them element by element:

```
zip :: [a] -> [b] -> [(a, b)]
zip []       []       = []
zip []       (y : ys) = ...
zip (x : xs) []       = ...
zip (x : xs) (y : ys) = (x, y) : zip xs ys
```

In the other two cases, there's a bit of flexibility:

- We could fail, yielding a partial function.
- But we can also just agree to return the shorter list.

Zipping two lists

Sometimes, we have two lists of equal length and want to combine them element by element:

```
zip :: [a] -> [b] -> [(a, b)]
zip []       []       = []
zip []       (y : ys) = []
zip (x : xs) []       = []
zip (x : xs) (y : ys) = (x, y) : zip xs ys
```

This definition has the advantage that we can use an infinite list as one argument:

```
zip [1 ..] listOfNames
```

Zipping two lists

Sometimes, we have two lists of equal length and want to combine them element by element:

```
zip :: [a] -> [b] -> [(a, b)]
zip (x : xs) (y : ys) = (x, y) : zip xs ys
zip xs       ys       = []
```

We can actually collapse the first three cases into one, but now the order of patterns matters.

Simple variables match everything.

Zipping two lists

Sometimes, we have two lists of equal length and want to combine them element by element:

```
zip :: [a] -> [b] -> [(a, b)]
zip (x : xs) (y : ys) = (x, y) : zip xs ys
zip _        _        = []
```

Pattern variables that are not used on the right hand side can be replaced by underscores.

Association lists

A list of pairs serves as a primitive way to associate keys with values.

```
numbers :: [(Int, String)]
numbers = [(1, "one"), (5, "five"), (42, "forty-two")]
```

Association lists

A list of pairs serves as a primitive way to associate keys with values.

```
numbers :: [(Int, String)]
numbers = [(1, "one"), (5, "five"), (42, "forty-two")]
```

Let's try to write a `lookup` function that obtains the value associated with a particular key …

Defining `lookup` – “bad” version

```
lookup :: key -> [(key, val)] -> val
```

A first approximation of the type.

Let’s analyze the input list.

Defining `lookup` – "bad" version

```
lookup :: key -> [(key, val)] -> val
lookup x []       = ...
lookup x (y : ys) = ... lookup ... ys ...
```

What `val` can we return if we reach the empty list and haven't found our key?

Defining `lookup` – “bad” version

```
lookup :: key -> [(key, val)] -> val
lookup x []       = error "lookup: unknown key"
lookup x (y : ys) = ... lookup ... ys ...
```

A bad solution is to trigger a run-time exception. We’ll improve on that shortly.

For the other case, we have to look at the first pair …

Defining `lookup` – “bad” version

```
lookup :: key -> [(key, val)] -> val
lookup x []             = error "lookup: unknown key"
lookup x ((k, v) : ys) = ... lookup ... ys ...
```

Now we have to compare `x` and `k`. Let’s use guards.

Defining `lookup` – “bad” version

```
lookup :: key -> [(key, val)] -> val
lookup x []                 = error "lookup: unknown key"
lookup x ((k, v) : ys)
   | x == k                 = ... lookup ... ys ...
   | otherwise              = ... lookup ... ys ...
```

If we found the key, we can immediately return the value. (So what will happen if the key occurs multiple times?)

Defining `lookup` – "bad" version

```
lookup :: key -> [(key, val)] -> val
lookup x []                = error "lookup: unknown key"
lookup x ((k, v) : ys)
   | x == k                = v
   | otherwise             = ... lookup ... ys ...
```

In the remaining case, we simply recurse.

Defining `lookup` – “bad” version

```
lookup :: key -> [(key, val)] -> val
lookup x []                = error "lookup: unknown key"
lookup x ((k, v) : ys)
   | x == k                = v
   | otherwise             = lookup x ys
```

Let’s take a final look. Oh, we need equality on the key type . . .

Defining `lookup` – “bad” version

```
lookup :: Eq key => key -> [(key, val)] -> val
lookup x []                = error "lookup: unknown key"
lookup x ((k, v) : ys)
   | x == k                = v
   | otherwise             = lookup x ys
```

Now we’re done, apart from the ugly call to `error` .

Excursion: about `error` and `undefined`

A call to `error` (as well as a function call for which no pattern match succeeds) causes a **run-time exception**.

Excursion: about `error` and `undefined`

A call to `error` (as well as a function call for which no pattern match succeeds) causes a **run-time exception**.

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

Note that these are polymorphic in the result type. This means they can be used in any context, because they abort normal control flow.

Excursion: total and partial functions

- A function that can trigger a run-time exception or that may loop is called a **partial** function.

Excursion: total and partial functions

- A function that can trigger a run-time exception or that may loop is called a **partial** function.
- Writing and using partial functions is discouraged – always try to cover all cases and make your functions **total**.

Excursion: total and partial functions

- A function that can trigger a run-time exception or that may loop is called a **partial** function.
- Writing and using partial functions is discouraged – always try to cover all cases and make your functions **total**.
- However, `undefined` and `error` can be useful tools while incrementally developing a program.

Maybe

How to fix `lookup`

We need a disciplined way to express failure without crashing.

How to fix `lookup`

We need a disciplined way to express failure without crashing.

Idea

Let's use a different result type for `lookup`, containing one additional value called `Nothing` to express failure.

The `Maybe` datatype

Given a type `a`, the type `Maybe a` contains all the values of type `a` plus one additional value:

- the term `Nothing` is a value of type `Maybe a`,
- if `x` is of type `a`, then `Just x` is of type `Maybe a`.

The `Maybe` datatype

Given a type `a`, the type `Maybe a` contains all the values of type `a` plus one additional value:

- the term `Nothing` is a value of type `Maybe a`,
- if `x` is of type `a`, then `Just x` is of type `Maybe a`.

There are two shapes / constructors of the `Maybe` datatype: `Nothing` and `Just`.

Defining `lookup` – "good" version

```
lookup :: Eq key => key -> [(key, val)] -> val
lookup x []                = error "lookup: unknown key"
lookup x ((k, v) : ys)
   | x == k                = v
   | otherwise             = lookup x ys
```

This is the version we had before.

Defining `lookup` – "good" version

```
lookup :: Eq key => key -> [(key, val)] -> Maybe val
lookup x []                 = error "lookup: unknown key"
lookup x ((k, v) : ys)
   | x == k                 = v
   | otherwise              = lookup x ys
```

We are now adapting the result type.

This requires changes in the rest of the function.

Defining `lookup` – “good” version

```
lookup :: Eq key => key -> [(key, val)] -> Maybe val
lookup x []                = error "lookup: unknown key"
lookup x ((k, v) : ys)
   | x == k                = v
   | otherwise             = lookup x ys
```

The first of the right hand sides can be improved now. The other is no longer type correct.

Defining `lookup` – "good" version

```
lookup :: Eq key => key -> [(key, val)] -> Maybe val
lookup x []                 = Nothing
lookup x ((k, v) : ys)
   | x == k                 = v   -- still wrong
   | otherwise              = lookup x ys
```

Instead of using `error`, we can now return `Nothing`.

Defining `lookup` – "good" version

```
lookup :: Eq key => key -> [(key, val)] -> Maybe val
lookup x []                = Nothing
lookup x ((k, v) : ys)
   | x == k                = Just v
   | otherwise             = lookup x ys
```

To inject `v` into the `Maybe` type, we use `Just`.

Defining `lookup` – "good" version

```haskell
lookup :: Eq key => key -> [(key, val)] -> Maybe val
lookup x []                 = Nothing
lookup x ((k, v) : ys)
   | x == k                 = Just v
   | otherwise              = lookup x ys
```

Done. This version of `lookup` is **total**.

It does not crash, but the type tells the user that `Nothing` may be returned, and forces the caller to deal with it.

Handling exceptions with `Maybe`

Given a default value, we can always recover a value from a `Maybe` :

```
fromMaybe :: a -> Maybe a -> a
fromMaybe def x        = ...
```

We **pattern match** on the `Maybe` .

Two constructors, `Nothing` and `Just` .

Handling exceptions with `Maybe`

Given a default value, we can always recover a value from a `Maybe` :

```
fromMaybe :: a -> Maybe a -> a
fromMaybe def Nothing  = ...
fromMaybe def (Just x) = ...
```

We use the default value in the `Nothing` case, and the wrapped value in the other.

Handling exceptions with `Maybe`

Given a default value, we can always recover a value from a `Maybe` :

```
fromMaybe :: a -> Maybe a -> a
fromMaybe def Nothing  = def
fromMaybe def (Just x) = x
```

Done.

Combining `Maybe` computations

We can provide a "backup" computation for a possibly failing computation.

```
(<|>) :: Maybe a -> Maybe a -> Maybe a
x        <|> y = ...
```

We pattern match on the first input.

Combining `Maybe` computations

We can provide a "backup" computation for a possibly failing computation.

```
(<|>) :: Maybe a -> Maybe a -> Maybe a
Nothing <|> y = ...
Just x  <|> y = ...
```

We take the second computation if the first fails, otherwise ignore it.

Combining `Maybe` computations

We can provide a "backup" computation for a possibly failing computation.

```
(<|>) :: Maybe a -> Maybe a -> Maybe a
Nothing <|> y = y
Just x  <|> y = Just x
```

Done.

Note the similarity with `(||)` on Booleans (from the Quick Tour).

A variant of `filter` for `Maybe`

```
mapMaybe :: (a -> Maybe b) -> [a] -> [b]
mapMaybe p [] = []
mapMaybe p (x : xs) = ... mapMaybe p xs ...
```

We get to this point, but now we have to inspect the result of `p x` .

We could define a helper function, but we can also use a **`case`** expression . . .

A variant of `filter` for `Maybe`

```
mapMaybe :: (a -> Maybe b) -> [a] -> [b]
mapMaybe p [] = []
mapMaybe p (x : xs) =
  case p x of
    Nothing -> ... mapMaybe p xs ...
    Just y  -> ... mapMaybe p xs ...
```

With `case`, we can pattern match on the result of an expression.

Note that the left hand sides are separated from the right hand sides with an arrow (`->`) and not an equality sign (`=`).

A variant of `filter` for `Maybe`

```
mapMaybe :: (a -> Maybe b) -> [a] -> [b]
mapMaybe p [] = []
mapMaybe p (x : xs) =
  case p x of
    Nothing -> mapMaybe p xs
    Just y  -> y : mapMaybe p xs
```

We can complete the definition similarly to `filter` .

Everything looks fine now.

Why `Maybe` ?

- By using `Maybe` in a result, we can express explicitly that the function can fail.
- The caller has to address the potential failure.
- By using `Maybe` in an argument, we can express that an argument is optional.
- The function writer has to say what to do if `Nothing` is passed.
- Only `Maybe` types have `Nothing`. This is different from `null` in other languages. There are no “null pointer exceptions” in Haskell.

Datatypes

Parameterized types, type constructors

- In Haskell, many types are parameterized by others.
- From existing types, we can make new types.
- Parameterized types are often called **type constructors**.

Parameterized types, type constructors

- In Haskell, many types are parameterized by others.
- From existing types, we can make new types.
- Parameterized types are often called **type constructors**.

Examples:

“List” is a type constructor.

Given any type `a` , there is also a type `[a]` .

“Pair” is a type constructor.

Given any types `a` and `b` , there is also a type `(a, b)` .

“Function” is a type constructor.

Given any types `a` and `b` , there is also a type `a -> b` .

Building types with type constructors

There are no limits to composing type constructors:

```
([Int -> Bool -> Int], [[Double]] -> (Bool, Int))
```

Building types with type constructors

There are no limits to composing type constructors:

```
([Int -> Bool -> Int], [[Double]] -> (Bool, Int))
```

There are arbitrarily many type constructors, because we can define our own!

Defining data types

New datatypes with the `data` construct

```
data Weekday = Mo | Tu | We | Th | Fr | Sa | Su
```

This is an **enumeration** type. There are 7 **constructors**:

```
Mo, Tu, We, Th, Fr, Sa, Su :: Weekday
```

New datatypes with the `data` construct

```
data Weekday = Mo | Tu | We | Th | Fr | Sa | Su
```

This is an **enumeration** type. There are 7 **constructors**:

```
Mo, Tu, We, Th, Fr, Sa, Su :: Weekday
```

```
data Date = D Int Int Int  -- year, month, day
```

This is a **record** type. It has a single constructor:

```
D :: Int -> Int -> Int -> Date
```

(Data) Constructors

```
Mo    :: Weekday
D     :: Int -> Int -> Int -> Date
False :: Bool
(:)   :: a -> [a] -> [a]
(,)   :: a -> b -> (a, b)
Just  :: a -> Maybe a
```

- Constructors are constants or functions that can be used to construct terms on the right hand side of a declaration.
- They have types targetting the datatype they belong to.
- Constructors determine the shape of values – they are not reduced, but evaluate to themselves.
- We can pattern-match on constructors (and not on ordinary constants or functions).

Datatypes yield programming patterns

From the datatype definition, we can read off the standard design principle for functions over the datatype:

- For each constructor, make a case.
- Use the arguments of the constructor on the right hand side.
- Whenever the datatype is recursive, consider making the function recursive.

Booleans

```haskell
data Bool = False | True
```

An enumeration type, like `Weekday` .

Two constructors, no recursion.

Booleans

```haskell
data Bool = False | True
```

An enumeration type, like `Weekday` .

Two constructors, no recursion.

Example functions:

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

(&&) :: Bool -> Bool -> Bool
(&&) True True = True
(&&) _    _    = False
```

Tuples

```
data (a, b)    = (a, b)
data (a, b, c) = (a, b, c)
```

Parameterized. One constructor each. No recursion. Built-in syntax.

Tuples

```
data (a, b)    = (a, b)
data (a, b, c) = (a, b, c)
```

Parameterized. One constructor each. No recursion. Built-in syntax.

We could define our own, with less convenient syntax:

```
data Pair   a b   = MakePair a b
data Triple a b c = MakeTriple a b c
```

Tuples

```
data (a, b)    = (a, b)
data (a, b, c) = (a, b, c)
```

Parameterized. One constructor each. No recursion. Built-in syntax.

We could define our own, with less convenient syntax:

```
data Pair   a b   = MakePair a b
data Triple a b c = MakeTriple a b c
```

Example functions:

```
secondOfThree :: (a, b, c) -> b
secondOfThree (x, y, z) = y

secondOfThree' :: Triple a b c -> b
secondOfThree' (MakeTriple x y z) = y
```

Maybe

```haskell
data Maybe a = Nothing | Just a
```

Parameterized. Two constructors. No recursion.

Maybe

```haskell
data Maybe a = Nothing | Just a
```

Parameterized. Two constructors. No recursion.

Example function:

```haskell
fromMaybe :: a -> Maybe a -> a
fromMaybe def Nothing  = def
fromMaybe def (Just x) = x
```

Lists

```
data [a] = [] | a : [a]
```

Parameterized. Two constructors. Recursive. Built-in syntax.

We could define our own, with less convenient syntax.

```
data List a = Nil | Cons a (List a)
```

We have seen lots of example functions following the standard design principle.

Well-Typed

The `data` construct

The syntax of the `data` construct:

```
data Type arg₁...argₘ = Con₁ ty₁ ... tyₙ
                      | Con₂ ...
                      | ...
```

Introduces the new datatype `Type` and the data constructors `Con₁`, `Con₂`, `...`.

The `data` construct

The syntax of the `data` construct:

```
data Type arg₁...argₘ = Con₁ ty₁ ... tyₙ
                      | Con₂ ...
                      | ...
```

Introduces the new datatype `Type` and the data constructors `Con₁`, `Con₂`, `...`.

Types of constructors are determined by the data declaration:

```
Con₁ :: ty₁ -> ... -> tyₙ -> Type arg₁ ... argₘ
```

The `data` construct

The syntax of the `data` construct:

```
data Type arg₁...argₘ = Con₁ ty₁ ... tyₙ
                      | Con₂ ...
                      | ...
```

Introduces the new datatype `Type` and the data constructors $\texttt{Con}_1$, $\texttt{Con}_2$, `...`.

Types of constructors are determined by the data declaration:

```
Con₁ :: ty₁ -> ... -> tyₙ -> Type arg₁ ... argₘ
```

Type and constructor names must start with an uppercase letter; symbolic infix constructors must start with a colon (`:`). Lists and tuples support additional built-in syntax that cannot be used for other datatypes.

Applying the design principle

Also for new datatypes, always keep in mind that by looking at the datatype, you obtain a design principle for functions over that type:

```
data Weekday = Mo | Tu | We | Th | Fr | Sa | Su
```

Applying the design principle

Also for new datatypes, always keep in mind that by looking at the datatype, you obtain a design principle for functions over that type:

```
data Weekday = Mo | Tu | We | Th | Fr | Sa | Su
```

```
isWeekend :: Weekday -> Bool
isWeekend Mo = False
isWeekend Tu = False
isWeekend We = False
isWeekend Th = False
isWeekend Fr = False
isWeekend Sa = True
isWeekend Su = True
```

Applying the design principle

Also for new datatypes, always keep in mind that by looking at the datatype, you obtain a design principle for functions over that type:

```haskell
data Weekday = Mo | Tu | We | Th | Fr | Sa | Su
```

```haskell
isWeekend :: Weekday -> Bool
isWeekend Sa = True
isWeekend Su = True
isWeekend _  = False
```

Collapsing cases – the order of cases then matters!

Another example

```haskell
data Date = D Int Int Int  -- year, month, day
```

One constructor. No recursion.

Example function:

```haskell
valid :: Date -> Bool
valid (D y m d) =
  m >= 1 && m <= 12 && d >= 1 && d <= 31
```

Of course, this is not an optimal definition.

Type synonyms with `type`

Often, for datatypes with a single constructor, the constructor is named the same as the datatype itself:

```
data Date = Date Int Int Int
```

Type synonyms with `type`

Often, for datatypes with a single constructor, the constructor is named the same as the datatype itself:

```
data Date = Date Int Int Int
```

It’s often better to give more meaningful names to types without creating a completely new type:

```
type Year  = Int
type Month = Int
type Day   = Int

data Date = Date Year Month Day
```

Type synonyms with `type`

Often, for datatypes with a single constructor, the constructor is named the same as the datatype itself:

```
data Date = Date Int Int Int
```

It's often better to give more meaningful names to types without creating a completely new type:

```
type Year  = Int
type Month = Int
type Day   = Int

data Date = Date Year Month Day
```

Note that `type` introduces type **synonyms**. For example, `2 :: Year` and `2 :: Int`. No conversion function is required.

Renamed types with `newtype`

We could also define:

```haskell
data Year = Year Int
```

Now `Year` and `Int` are **different**:

```haskell
Year :: Int -> Year   -- the constructor
```

To extract the `Int` from a year, we can use **pattern matching**:

```haskell
fromYear :: Year -> Int
fromYear (Year n) = n
```

Renamed types with `newtype`

We could also define:

```haskell
data Year = Year Int
```

Now `Year` and `Int` are **different**:

```haskell
Year :: Int -> Year  -- the constructor
```

To extract the `Int` from a year, we can use **pattern matching**:

```haskell
fromYear :: Year -> Int
fromYear (Year n) = n
```

For the case of a single-constructor, single-argument datatype (i.e., a **renamed** type), there’s a more efficient construct:

```haskell
newtype Year = Year Int
```

Lessons

- Let the types guide you.
- Use pattern matching to get at components of values, and to distinguish cases.
- Try to follow the recursive structure of types (lists, trees).
- Cover all cases.
- Use precise types, such as `Maybe` , rather than causing uncontrolled errors.