Introductory Haskell, polyconf'15, Alexander Ulrich, Matthias Fischmann

Basic data types

```
> 42
> 2.3
> "foo"
> 'x'
> True
> False
> (42, "foo")
```

Basic Syntax

```
-- A function named 'f' with one parameter
f x = x * 2

-- A function named 'g' with two parameters
g x y = x + y

-- Function application is syntactically lightweight.
> g 42 (f 23)
```

More forms of expressions:

```
-- let-bindings
> let x = 5 in x + 42
-- conditionals
> if x == 5 then [42] else []
> \x y -> x + y
```

The last one is an *anonymous function* or lambda expression. Functions are first-class values - regular values that you can pass around like any other.

Inductive list definition, list construction examples

Lists are an important data structure in Haskell (as in most other functional languages).

Lists are defined *inductively*. You might know the schema from various Lisp or ML dialects.

```
-- The empty list
> []

-- Construct a new list from a value (head) and a list (tail): cons
> 2 : []

-- Constructing lists with more elements
> 1 : (2 : [])

-- Without parentheses
> 1 : 2 : []

-- Syntactic sugar for list construction
> [1,2,3]
> 42 : [1,2,3]
> [42,19] ++ [1,2,3]
```

Syntax `[e_1, ..., e_n]` is transformed into an equivalent expression using the cons `(:)` operator.

Functions on lists

Task: Given a value and a list, check wether the list contains the value.

Implementation: *Deconstruct* the list according to its inductive definition by *pattern matching*. A list is either empty or a cons combination of a value and some other list. Treat the cases separately!

```
import Prelude hiding (elem)

-- Any value is certainly not an element of the empty list
elem x [] = False

-- Check wether the head value equals 'x' or if 'x' occurs
-- in the tail of the list.
elem x (y : ys) = x == y || elem x ys

> elem 5 [6,9,42]
> elem 9 [6,9,42]
```

- Two equations, two cases for the underlying data type: case-wise function definition. Very common pattern for function definitions
- A *recursive* call deals with the tail of the list (if necessary)
- **boolean or: short-cut evaluation**
- In Haskell, all bindings are mutually recursive by default

Truth values

Truth values are just a data type with two constructors:

```
> True
> False
```

Define boolean functions by pattern-matching on those constructors. We define our own version of boolean that behaves exactly like the Haskell one. The `or`-operator is present in the standard library, but we define our own version.

Haskell allows to define our own infix operators.

```
import Prelude hiding ((||))

True  || y = True
False || y = y

> True  || False
> False || False
```

Side note: An infix operator can be used as a regular function:

```
> (||) True False
```

Additionally, any function with two arguments can be used as an infix operator by enclosing it in backticks.

```
> 5 `elem` [1,2,3,5]
```

Lazy Evaluation

Short-cut behaviour of boolean operators is not a special hack in Haskell.

Lazy evaluation: function arguments are evaluated only when they are actually required. Usually, *required* means that they are matched on.

```
-- A special value that raises an exception when it is evaluated
> undefined

-- Only need to evaluate the first argument to True to give the result
> True || undefined

-- Need to evaluate the second argument.
> False || undefined
```

Lazy evaluation is neat:

```
> let allIntegers = [1..]
-- Show evaluation status of binding. Thunks are marked with an underscore
> :print allIntegers
> :t take
> take 10 allIntegers
> :print allIntegers
> :t sum
> sum (take 10 allIntegers)
> sum allIntegers
```

We can work with non-terminating computations:

```
ones = 1 : twos

twos = 2 : ones
```

Type Inference

Haskell is a strong and statically typed language. The compiler checks if every expression is type-correct.

But: We have not seen any type signatures so far. How is this consistent with static typing?

Answer: The compiler can infer the type of an expression from the types of its sub-expressions. *Type inference*

Example:

```
not True  = False
not False = True
```

From the code, we can conclude that:

- It's a function (it has an argument)
- It takes a truth value
- It produces a truth value

```
not :: Bool -> Bool
```

The compiler checks wether we have given a correct type for the function.

We can also ask the compiler for the type of some expression or function:

```
> :t not
> :t not True
```

What is the type of the "or" function?

```
> True || False
```

It takes two bools and produces a bool:

```
(||) :: Bool -> Bool -> Bool

-- Actually: A function that is applied to a bool and gives us another function.
(||) :: Bool -> (Bool -> Bool)
```

Let's return to our `elem` function. We can give its type as follows:

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

-- But actually, it's this type:
elem :: Int -> ([Int] -> Bool)
```

What is the type of `elem 5`?

Functions with more than one parameter can be *partially applied*. Partial application specializes (or fixes) a function on some parameters.

```
containsFive :: [Int] -> Bool
containsFive = elem 5

> containsFive [1,2,3]
> containsFive [1..10]
```

(Parametric) Polymorphism

Let's define a function that appends two lists of integers.

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

-- Case 1: first argument is the empty list
[] ++ ys     = ys

-- Case 2: non-empty first argument.
(x:xs) ++ ys = x : (xs ++ ys)
```

Do we use the fact that we append lists of *integers* specifically? Wouldn't the code for appending lists of strings look exactly the same (except in the type signature)?

==> Remove type signature

```
["foo", "bar"] ++ ["baz"]
```

What, then, is the type of `(++)`? Let's ask the compiler. For any program, Haskell infers not only *some* type, but *the most general type*.

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

Read this as: For *any element type* `a`, the function takes two lists of `a` and produces a new list of `a`. Crucially though, both arguments must have *the same* element type.

`a` is a type variable which can be *instantiated* to any type.

Parametric polymorphism: A function behaves the same, regardless of the type. Parametric polymorphism is a powerful way to write abstract and generic code.

Another polymorphic function

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

Note also that this is a higher-order function. The Haskell eco system is full of powerful and generic abstractions like this.

```
> map (elem 5) [[1..10], [2..4], []]
> map (+ 1) [1,2,3,4,5]
```

Type Classes: Ad-hoc overloading

If a type contains a type variable, the function behaves the same for all types. In other words: we can not assume anything about the type. In a lot of cases, we do not want to handle all types, but only those that support certain operations.

What is the most general type of our `elem` function? More than just integers. But can it handle *all* types?

```
:t elem
elem :: Eq a => a -> [a] -> Bool
```

`elem` can handle all types of list elements that support equality.

Types that support equality are in the *type class* `Eq`. The type of `elem` contains a *class constraint*.

Read: For all types `a` that support equality...

Type class: a collection of types that support common functionality.

The comparison operator is actually a *method* of the type class `Eq`.

```
> :t (==)
-- Accepts any type that is an instance of Eq

> :info Eq
```

Haskell brings a number of standard type classes. Defining your own type classes in a Haskell program is very common.

```
> :i Ord
> :i Show
> show 3
> show "4"
> import Data.Aeson
> :info ToJSON
> :info FromJSON
```

Class and Instance Definitions

Assume you are unhappy with the fact that you can't write:

```
if 0 then ... else ...
```

because `0` is an `Int`, not a boolean.

This is arguably a bad idea in the first place, and you really want to just have booleans in conditions.

But if you must do it, type classes can help you.

```
class Boolsy a where
    boolsy :: a -> Bool

if boolsy 0 then ...
if boolsy "" then ...
if boolsy False then ...
```

```
instance Boolsy Bool where
    boolsy True  = True
    boolsy False = False

instance Boolsy Int where
    boolsy 0 = False
    boolsy _ = True

if' :: Boolsy c => c -> a -> a -> a
if' c thn els = if boolsy c then thn els
```

Data types

Haskell supports a powerful mechanism to define new data types: *algebraic data types* or *sums of products*.

Simple example:

```
data Maybe a = Just a | Nothing

> Just 3
> Nothing
> [Just 3, Just False]
> [Just 3, Nothing, Just 5]

data Either a b = Left a | Right b

> [Left 3, Right False]
```

(We'll get back to Maybe in the Monad part of this tutorial.)

Let's define a data type for binary trees with node labels of *some type*. The type gets a *type parameter*.

```
data Tree a = Leaf
            | Node (Tree a) a (Tree a)
```

We have two constructors. A `Leaf` is a tree, as well as a `Node`, applied to three parameters. Note that the type is recursive: A tree contains two child trees (if it is a `Node`).

This is an easy example of a generic data structure.

```
-- The 'Leaf' constructor is a valid tree for /any/ element type.
> :t Leaf
> :t Node Leaf "foo" Leaf
```

Side note: What is the type of `Node`?

```
> :t Node
```

Data constructors are just functions and can be treated like any other function.

Functions over data types usually follow the structure of inductively defined types by pattern matching and recursion (just as in the case of lists).

```
sumTree :: Tree Int -> Int
sumTree Leaf           = 0
sumTree (Node t1 x t2) = sumTree t1 + x + sumTree t2

t :: Tree Int
t = Node
       (Node Leaf 3 Leaf)
       91
       (Node
           (Node Leaf 16 (Node Leaf 21 Leaf))
           24
           Leaf)

sumTree t
```

We can interpret binary trees as *search trees*. Write a function that checks wether a value is a member of a given search tree.

```
elemTree :: Ord a => a -> Tree a -> Bool
elemTree _ Leaf = False
elemTree a (Node t1 x t2) | a == x    = True
                          | a < x     = elemTree a t1
                          | otherwise = elemTree a t2
```

Equations of a function can be further qualified with *guards* that test a boolean condition. A combination of `if`-conditionals would work as well, but would be considerably less elegant.

We can build an associative map based on binary search trees.

```
-- A type alias
type Map a b = Tree (a, b)
```

A lookup function on such a map is easily defined:

```
-- incorrect. Need Ord constraint on a
-- lookupMap :: a -> Map a b -> b
lookupMap :: Ord a => a -> Map a b -> b
lookupMap x Leaf = ???
```

We have to account for the case that the key we search is not present in the map. Other languages would likely "model" this case by returning a NULL reference. NULL values lead to a large number of well-known (and very expensive) problems (Tony Hoare: "my billion-dollar mistake"). Luckily, the following isn't valid Haskell code.

```
lookupMap x Leaf = NULL
```

`lookupMap` returns either a value of type `b` or nothing at all. Haskell models this explicitly in a simple data type.

```
> :info Maybe
```

Maybe is defined in the Haskell standard library.

When calling a function that *maybe* returns a value, the type system forces us to explicitly handle the error case (`Nothing`) by pattern matching on the `Maybe` value.

We can now define the `lookupMap` function:

```
lookupMap :: Ord a => a -> Tree (a, b) -> Maybe b
lookupMap _ Leaf = Nothing
lookupMap x (Node t1 (a,b) t2)
    | a == x    = Just b
    | x < a     = lookupMap x t1
    | otherwise = lookupMap x t2
```

Side note: Note the nested pattern in the second equation.

```
t2 :: Map Int String
t2 = Node
       (Node Leaf (3, "drei") Leaf)
       (91, "einundneunzig")
       (Node
           (Node Leaf (16, "sechzehn") (Node Leaf (21, "einundzwanzig") Leaf))
           (24, "vierundzwanzig")
           Leaf)

> lookupMap 5 t2
> lookupMap 6 t2
```

Purity and Effects

See well-typed QuickIntro (slide 41..end).

Purity/referential transparency is nice because:

- control over effects through types
- testing
- code analysis

Holes

`mapMaybe` is actually defined in the standard library, but we will define it ourselves.

In Haskell, the expressive type system allows to specify interesting properties of programs. But furthermore, we can also let us be *guided by types* during development.

```
mapMaybe :: (a -> Maybe b) -> [a] -> [b]
-- C-u M-t for equation.
```

An expression that begins with an underscore is called a *hole*. The Haskell compiler gives us information about the type of the expression that we should fill the hole with.