Chapter 6

Functional Programming

Background

- In Erlang you've already seen some basic concepts of functional programming
- Now we'll take a closer look at this paradigm using a purely functional programming language: Haskell
  - Was/is developed by a committee of experts combining the best features of functional programming languages
  - Named after the logician Haskell Curry
- Basically, Haskell is a statically and strongly typed, compiled, functional language

Functional Style Revisited

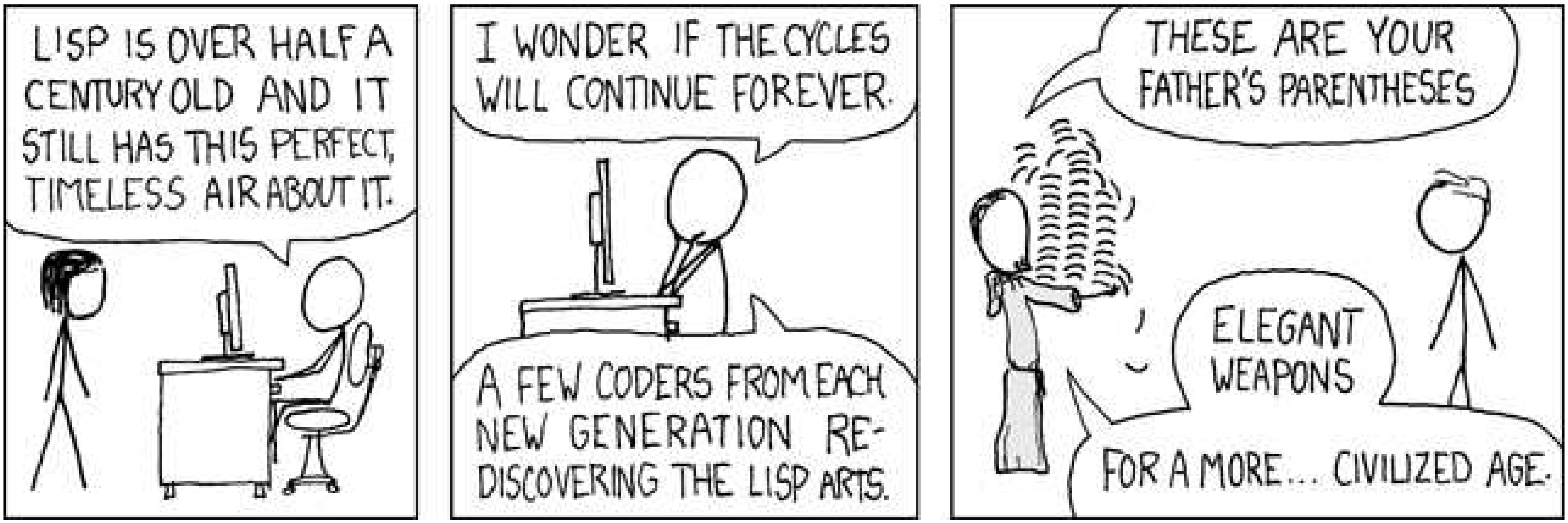
- Like in Erlang, in Haskell
  - functions return the same output, given the same input
  - functions do not have side effects, i.e., they do not modify program state
  - a variable can only be assigned (matched) a value once
- This is called *referential transparency*

Referential Transparency

- What are the advantages of referential transparency?
  - Allows a compiler to figure out a program's behavior more easily
  - Allows a programmer to show correctness of their code more easily
    - Helps in building correct programs by putting together smaller, correct functions
  - Allows Haskell to do *lazy evaluation*: it will not compute anything until the result is actually needed
    - For example, an infinite data structure is not a problem (as long as you don't try to access all of it)

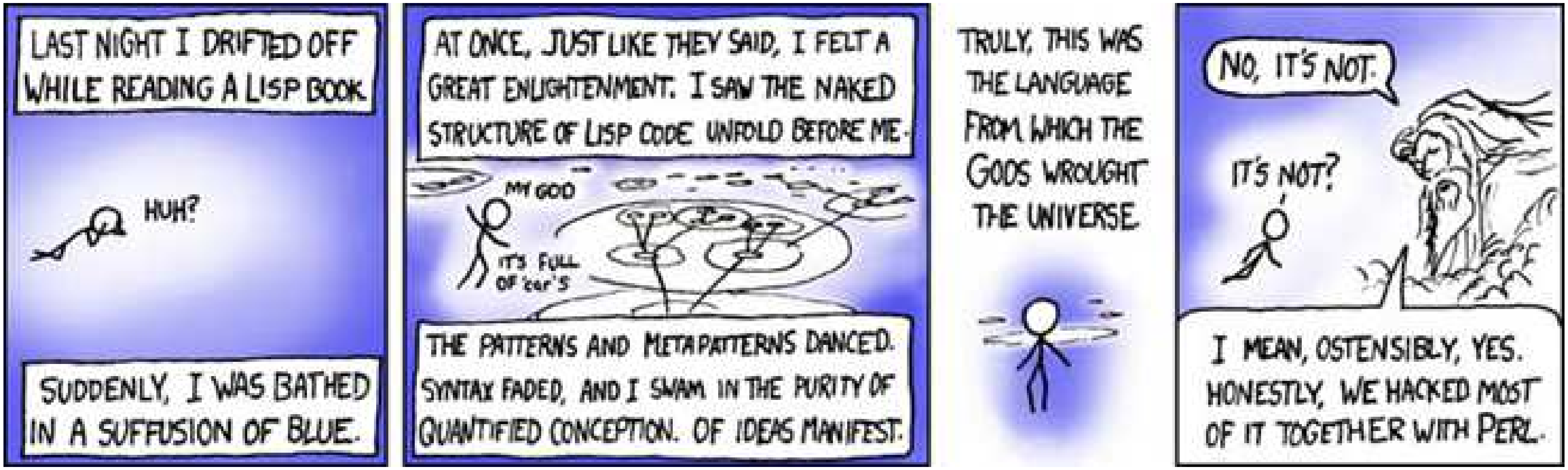
What Do the “Experts” Say?

- Functional programming is considered an elegant style of programming



What Do the “Experts” Say? (2)

- It is considered to be a bit academic, though



What Do the “Experts” Say? (3)

- However, the functional style of programming is applied in practice
  - There are users in the financial industry (mainly for building complex models)
    - More details are provided here:

      `http://www.haskell.org/haskellwiki/Haskell_in_industry`
  - Unreal Engine 3 has taken functional programming concepts on board, e.g. see here:

    `http://graphics.cs.williams.edu/archive/SweeneyHPG2009/TimHPG2009.pdf`
    - Purists would disagree, as the engine is written in C++, but functional concepts are applied

First Steps

- Although Haskell is usually compiled, there is also an interactive interpreter
- Typing some stuff into the interpreter will already tell us something about Haskell

```
> 5 + "string"
...some lengthy error message...
```

- As Haskell is strongly typed, it doesn't like you to mix types

First Steps (2)

- It can infer types, though:

```
> 5 + 3.6
8.6
```

- “Variables” in Haskell are lower-case

```
> a = 5
<interactive>:1:3: parse error on input '='
```

- When doing so in the shell, you have to use `let` (which controls the scope):

```
> a = 5
> a
5
```

First Steps (3)

- When calling functions, parameters are not enclosed in parentheses, you just list them:

```
> min 8 12
8
```

- Nevertheless, parentheses are used to indicate precedence

```
> max (min 8 12) (min 3 7)
8
```

Writing Your Own Functions

- When defining a function of your own, you have to provide the following (don't forget the `let` in the shell):
  - The name of the function
  - A list of parameters
  - The symbol =
  - The actual definition of the function

```
> let doubleMe x = x + x
> doubleMe 8
16
```

Bootstrapping Complex Functions

- If you want to double two numbers and add them, you could start from scratch:

```
doubleUs x y = x * 2 + y * 2
```

- However, it is good (functional) programming style to re-use correct code:

```
doubleUs x y = (doubleMe x) + (doubleMe y)
```

Conditionals

- Conditionals are functions in Haskell, so they always have to return something:

```
doubleSmallNumber x = if x > 100 then x else x*2
```

- Writing statements spanning more than one line in the shell can be a bit of a pain:

```
> :{
| let { doubleSmallNumber x = if x > 100
| ;then x
| ;else x*2}
| :}
```

- After this quick introduction, we are going to be using (compiled) modules

Lists

- Haskell also supports lists with the standard square bracket notation

```
> let numberlist = [1,2,3,4,5]
```

- The syntax for getting the head and the tail of a list is a bit different:

```
> let a:b = numberlist
> a
1
> b
[2,3,4,5]
```

- This can also be used to construct new lists:

```
> 10:[11,12]
[10,11,12]
```

Lists (2)

- Alternatively, you can call the functions head or `tail`

```
> head numberlist
1
> tail numberlist
[2,3,4,5]
```

- There are also functions to take or drop the first n elements of a list:

```
> take 3 numberlist
[1,2,3]
> drop 3 numberlist
[4,5]
```

Ranges

- Similar to Ruby you can create lists of numbers in a certain range

```
> [1..10]
[1,2,3,4,5,6,7,8,9,10]
```

- You can also skip some numbers or count backwards:

```
> [2,4..20]
[2,4,6,8,10,12,14,16,18,20]
[10,7..1]
[10,7,4,1]
```

- Or create an infinite list

```
> let xxx = [1..]
> take 5 xxx
[1,2,3,4,5]
```

List Comprehensions

- *Set comprehension* is a mathematical way of defining specific sets, given a more general set
- For example, the first ten even natural numbers can be defined by:

$$S_{even10} = \{2x | x \in \mathbb{N}, x \leq 10\}$$

- Set comprehensions are usually described by
  - an output function (here $2x$)
  - a variable (here x)
  - an input set (here $\mathbb{N}$)
  - a predicate (here $x \leq 10$)

List Comprehensions (2)

- In Haskell this concept can be applied to lists
- Allows you to generate lists too complex for ranges
- For example, out of the first five odd natural numbers, we want those whose square is not equal to 25

```
[ x | x <- [1,3..9], (x*x) /= 25 ]
```

- $< -$ stands for $\in$ (or is interpreted as “drawn from”)
- The above list comprehension will output

```
[1,3,7,9]
```

Tuples

- Haskell also knows tuples (like Erlang)
- However, (unlike Erlang) it uses round brackets:

  ```
  (1,"one","uno")
  ```

- Unlike lists, tuples can combine different data types in the same tuple

Haskell's Type System

- After mentioning types a few times now, it's time to have a closer look
- The `:t` command gives you the type of an expression

```
> :t 'a'
'a' :: Char
> :t True
True :: Bool
> :t "hello!"
"hello!" :: [Char]
> :t (True,'a')
(True,'a') :: (Bool, Char)
> :t 4==5
4==5 :: Bool
```

Haskell's Type System (2)

- All the major built-in types found in other languages are also available in Haskell
- You can also find out the types of functions:

```
> :t doubleMe
doubleMe :: Integer -> Integer
> :t doubleUs
doubleUs :: Integer -> Integer -> Integer
```

- The last type is the return type

Type Variables

- Let's look at more subtle typing issues
- For example, what is the type of the function `head`?
- It can be applied to lists of different types

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

- Types start with an upper-case letter, the answer includes a lower-case letter
- a is a type variable, i.e., a can be of any type

Type Classes

- What is the type of the comparison operator?

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

- Again, we have a type variable, but this time with a restriction
- The symbol =$>$ is called a *type constraint*
- Any type that is comparable should be a member of the *type class* Eq
- Haskell supports a couple of type classes, e.g. `Ord` for types that have ordering or `Num` for types that have numerical values

Type Classes (2)

- Type classes are similar to interfaces
- They tell you what kind of functions a type supports
- For example:
  - types belonging to the type class `Num` support all the standard mathematical operators: +, -, *, /, . . .
  - `Show` converts values to strings
  - `Read` is the opposite: takes a string and converts it to a value

Modules

- Let's start with some proper programming and define and compile code in modules
- The code below shows a complete module (module names start with an upper-case letter)

```
module MyModule (
doubleMe
) where

doubleMe :: Integer -> Integer
doubleMe x = x + x
```

- Note that we also define the type of the function `doubleMe`

Modules (2)

- You can load the file `MyModule.hs` straight into the interpreter:

```
> :l MyModule
[1 of 1] Compiling MyModule
         ( MyModule.hs, interpreted )
Ok, modules loaded: MyModule.
*MyModule>
```

- The code will then be interpreted
- You can also compile it using the OS command ghc and then load the compiled version with `:l` as above

Modules (3)

- If you want to re-use code from a module in another module, you can import it

```
module YAM (
doubleUs
) where

import MyModule

doubleUs :: Integer -> Integer -> Integer
doubleUs x y = (doubleMe x) + (doubleMe y)
```

Functions

- Now that we have modules, let's write slightly more sophisticated functions
- Haskell does pattern matching like Erlang
  - Goes from top to bottom, taking the first match

```
module MyMath (
factorial
) where

factorial :: Integer -> Integer
factorial 0 = 1
factorial x = x * factorial (x -1)
```

Functions (2)

- If you need to match in a different or very particular order, you have to use *guards*

```
module MyMath (
factorial
) where

factorial :: Integer -> Integer
factorial x
    | x > 1 = x * factorial (x -1)
    | otherwise = 1
```

- The vertical bar | on the left indicates the scope
- A guard is a Boolean value followed by = and the definition of the function

Functions (3)

- Next we are going to unleash more of the power of Haskell
- We are writing a function computing Fibonacci numbers using lazy evaluation

```
module Fibonacci (
lazyFib,
fib
) where

lazyFib :: Integer -> Integer -> [Integer]
lazyFib x y = x:(lazyFib y (x + y))

fib :: Int -> Integer
fib x = head(drop (x-1) (lazyFib 1 1))
```

Functions (4)

- `lazyFib` generates an infinite sequence of Fibonacci numbers

```
> lazyFib 1 1
[1,1,2,3,5,8,13,21,34,55,89,144,...
```

- Due to lazy evaluation, we never actually generate the whole list
- `fib` takes the first `x` elements of this list, drops all but the last one, and then takes the head of this singleton list

```
> fib 4
3
```

Functions (5)

- Combining lots of functions to get a result is a common pattern in functional languages
- This is called *function composition*
- As this is very common, Haskell has a shortcut notation
- Instead of writing

```
f(g(h(i(j(k(l(m(n(o(x))))))))))
```

you can write

```
f.g.h.i.j.k.l.m.n.o x
```

Functions (6)

- So our Fibonacci code could be rewritten into

```
module Fibonacci (
lazyFib,
fib
) where

lazyFib :: Integer -> Integer -> [Integer]
lazyFib x y = x:(lazyFib y (x + y))

fib :: Int -> Integer
fib x = (head.drop (x-1)) (lazyFib 1 1)
```

Higher-Order Functions

- Haskell (as functional language) supports higher-order functions, i.e., functions that can take functions as parameters or return functions
- Let's start off with anonymous functions, called *lambda* in Haskell
- The syntax is

  ```
  (\parameter_1,...,parameter_n -> function body)
  ```

- For example, just returning the input parameter would be

  ```
  > (\x -> x) "mirror, mirror on the wall"
  "mirror, mirror on the wall"
  ```

Higher-Order Functions (2)

- Haskell also knows the usual list functions, such as `map, foldl, foldr, filter`

```
> map (\x -> x * x) [1,2,3]
[1,4,9]
> foldl (\x sum -> sum + x) 0 [1..10]
55
```

Curried Functions

- Every function in Haskell takes a **single** parameter
- We've already defined functions with multiple input parameters, so how does this work?
- Haskell uses the concept of *curried functions*
- Functions are applied partially, one parameter at a time
- Let's have a look at an example

Curried Functions (2)

- When Haskell computes the maximum of two numbers, what is really going on behind the scenes?
- `max 4 5` is evaluated in two steps:
  - Haskell creates a function that takes one parameter and returns either 4 or the parameter (depending on which one is larger)
  - Then 5 is passed as a parameter to this function
- So we are actually computing

```
(max 4) 5
```

Curried Functions (3)

- Let's have a look at the type of `max`

```
> :t max
max :: Ord a => a -> a -> a
```

- What this really says is the following:
  - `max` takes a as an input parameter and returns a function taking a as a parameter and returning an a
  - So it could be written as `a -> (a -> a)`

Curried Functions (4)

- What are the advantages of curried functions?
  - We can create new functions on the fly, already partially evaluating a function in a different context
  - It makes formal proofs about programs simpler, because all functions are treated in the same way
  - There are some techniques used in Haskell where currying becomes important

User-Defined Types

- In Haskell you can declare your own data types
- The simplest version is just a finite list of values, an *enumeration*

```
data Verdict = Guilty | Innocent
```

- A variable of type `Verdict` will have a single value, either `Guilty` or `Innocent`

Enumerated Types

- In the following, `Suit` and `Rank` are *type constructors*

```
module Cards where

data Suit = Spades | Clubs | Hearts | Diamonds
data Rank = Ace | Ten | King | Queen | Jack
```

- Loading this module and then trying to use one of these values leads to an error message

```
> :l Cards
...
*Cards> Spades

<interactive>:1:1:
    No instance for (Show Suit)
    ...
```

Enumerated Types (2)

- Haskell tells us that it does not know how to show values of these types
- That is because `Suit` is not a member of the type class `Show`
- For simple data types, Haskell can derive instances of type classes automatically

```
data Suit = Spades | Clubs | Hearts | Diamonds
     deriving (Show)
data Rank = Ace | Ten | King | Queen | Jack
     deriving (Show)

> Clubs
Clubs
> Ten
Ten
```

Composite Types

- When building more complex types, we can uses a type *alias*

```
data Suit = Spades | Clubs | Hearts | Diamonds
     deriving (Show)
data Rank = Ace | Ten | King | Queen | Jack
     deriving (Show)
type Card = (Rank,Suit)
type Hand = [Card]
```

- An alternative way is to introduce a *type constructor*

```
data Card = Crd (Rank,Suit) deriving (Show)

> let card1 = Crd(Ten,Hearts)
> card1
Crd (Ten,Hearts)
```

Composite Types (2)

- If we want to know the value of a card, we could write a function taking a `Rank` and returning an `Int`

```
value :: Rank -> Int
value Ace = 11
value Ten = 10
value King = 4
value Queen = 3
value Jack = 2
```

- Applying this function:

```
> let card1 = (Ace,Spades)
...
> (r,s) = card1
> value r
11
```

Functions and Polymorphism

- If we want to write a function that reverses a list of cards, we could do the following

```
backwards :: Hand -> Hand
backwards [] = []
backwards (h:t) = backwards t ++ [h]
```

- However, that would restrict the function to lists containing items of type `Hand`
- If we want it to work with general lists, we can introduce any type

```
backwards :: [a] -> [a]
backwards [] = []
backwards (h:t) = backwards t ++ [h]
```

Polymorphic Types

- User-defined types can also be made polymorphic
- For example, you need a type that stores a list of pairs
  - However, the pairs should be able to contain any type

```
data ListOfPairs a = LoP [(a,a)]
     deriving (Show)

> let list1 = LoP[(1,2),(2,3),(3,4)]
> list1
LoP [(1,2),(2,3),(3,4)]
```

Polymorphic Types (2)

- If you need the pairs to store different kinds of types, you can use different type variables

```
data AdvListOfPairs a b = ALoP [(a,b)]
     deriving (Show)

> let list2 = ALoP[(1,'a'),(2,'b')]
> list2
ALoP [(1,'a'),(2,'b')]
> let list3 = ALoP[(1,2),(2,3),(3,4)]
> list3
ALoP [(1,2),(2,3),(3,4)]
```

Recursive Types

- You can have recursive types in Haskell
- Let's look at an example: defining a polymorphic tree structure

```
data Tree a = Nil | Node a (Tree a) (Tree a)
     deriving (Show)

let tree1 = Nil
> tree1
Nil
> let tree2 = Node 'a' (Node 'b' Nil Nil)
                      (Node 'c' Nil Nil)
> tree2
Node 'a' (Node 'b' Nil Nil) (Node 'c' Nil Nil)
```

Recursive Types (2)

- Operating on recursive types often needs recursive functions as well
- If we want to determine the depth of a tree, we could do it like this:

```
depth :: Tree a -> Int
depth Nil = 0
depth (Node a left right) =
      1 + max (depth left) (depth right)
```

- The first case is straightforward: an empty tree has depth 0
- The second case traverses the tree recursively and adds one to depth of the deeper subtree

Type Classes Revisited

- We are now going to have another look at type classes
- So far we've automatically made some of our types instances of existing type classes (with the keyword `deriving`)
- We will now
  - make a type instance of a type class explicitly (Haskell may not always be able to derive this automatically)
  - create our own type class

Type Classes Revisited (2)

- Let's build a simple type called `TrafficLight`

```
data TrafficLight = Red | Yellow | Green
```

- We want this type to be comparable, i.e., be an instance of type class `Eq`
- Type class `Eq` is defined as follows:

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

    x == y = not (x /= y)
    x /= y = not (x == y)
```

Type Classes Revisited (3)

- So to be an instance of `Eq`, our type needs to implement the function `(==)` or `(/=)`
  - The last two lines mean that Haskell can figure out the definition of the other function
- All that is left to do is to declare `TrafficLight` an instance of `Eq` and declare `(==)` or `(/=)`

```
instance Eq TrafficLight where
    Red == Red = True
    Green == Green = True
    Yellow == Yellow = True
    _ == _ = False
```

User-Defined Type Classes

- Let’s build our own type class
- In other languages, you can use lots of different values for conditionals
  - For example, in JavaScript, 0 and "" evaluates to false, any other integer and non-empty string to true
- To introduce this behavior into Haskell, we write a YesNo type class that takes a value and returns a Boolean value

```
class YesNo a where
    yesno :: a -> Bool
```

User-Defined Type Classes (2)

- Next, we'll make `Int/Integer` an instance of our new type class

```
instance YesNo Int where
    yesno 0 = False
    yesno _ = True

instance YesNo Integer where
    yesno 0 = False
    yesno _ = True

> yesno 4
True
> yesno 0
False
```

Functor Type Class

- The functor type class is a type class including all the types you can apply a `map` operator to
  - For example, lists are an instance of this type class
- How is this class defined?

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

- What does this signature mean?
- `f` is a type constructor, i.e., a constructor that takes a type parameter to create a new type
- For example, a list is a type that takes a type parameter
  - A concrete value always has to be a list of some type, e.g., a list of strings, it cannot be just a generic list

Functor Type Class (2)

- So a functor takes
  - a function from a type a to a type b
  - and a type with type parameter a

  and returns
  - a type with type parameter b
- For example, for a list of type a and a function `a -> b`
  - you get as return value a list of type b
  - And that's exactly what a `map` operator does on a list
- Put more simply: give me a function `a -> b` and a box with as in it and I'll give you a box with bs in it

Functor Type Class (3)

- List is already an instance of the type class `Functor`
- Let's see if we can make `Tree` an instance

```
instance Functor Tree where
    fmap f Nil = Nil
    fmap f (Node x left right) =
        Node (f x) (fmap f left) (fmap f right)
```

- Doing a map on an empty tree is straightforward, it is going to return an empty tree
- For any other tree, we have to recursively go down the left and right subtrees

Functor Type Class (4)

- Now we can run a map (more specifically an fmap) on our tree

```
> let tree1 = Node 1 (Node 2 Nil Nil)
                     (Node 3 Nil Nil)
> fmap (+2) tree1
Node 3 (Node 4 Nil Nil) (Node 5 Nil Nil)
> fmap (show) tree1
Node "1" (Node "2" Nil Nil) (Node "3" Nil Nil)
```

Input/Output

- We have avoided any input/output (IO) functions so far
  - For the other languages we always started out with a small “Hello world!” program
- Adding IO to a purely functional language poses some problems
- We need at least two IO operations: one for input and one for output
  - Input does not need an input parameter, and may return different values
  - Output does not return a value, but clearly has side effects: it changes the state of the output device
- Neither of them is a function

Input/Output (2)

- Haskell needs to interact with the “real world” somehow
- It separates the functional (“pure”) parts of a program from the non-functional (“impure”) parts
- Haskell has `getLine` to read a string and `putStrLn` to print a string
- What are the types of `getLine` and `putStrLn`?

```
> :t getLine
getLine :: IO String
> :t putStrLn
putStrLn :: String -> IO ()
```

IO Actions

- Haskell has the concept of an *IO action*
- An IO action is something that
  - when performed, will carry out an action with some side effect (impure)
  - contains a return value inside of it (pure)
- For our input and output operations this means
  - `getLine` does some “dirty” stuff in `IO`, but part of the result is a “clean” data type: a string
  - `putStrLn` gets a string as input parameter and returns an IO action with no result (empty tuple) in it

IO Actions (2)

- How do we actually execute IO actions?
- You assign it to `main`, which will run it for you

```
> let main = putStrLn "Hello World!"
> main
Hello World!
```

- With the help of `main` you can also compile stand-alone programs

```
module Main where
main = putStrLn "Hello World!"
```

- You can put the above in a file `helloworld.hs` and run it through ghc to get an executable

IO Actions (3)

- Running a single IO action would not lead to very exciting programs
- Haskell allows you to “glue” together IO actions:

```
main = do
    putStrLn "Hi there, what's your name?"
    name <- getLine
    putStrLn ("Hello " ++ name ++ "!")
```

- The lines in a do-block work similar to an imperative execution
- `<-` extracts the “pure” part (the string) from `getLine`’s `IO String` value

IO Actions (4)

- The `IO` action carries along the baggage of the impure context
  - So you don't have to worry about it
- If you want to do a "pure" assignment in the context of `IO`, you have to use `let`

```
module Main where

import Data.Char

main = do
    putStrLn "What's your name?"
    name <- getLine
    let bigName = map toUpper name
    putStrLn ("Hi " ++ bigName ++ "!")
```

IO Actions (5)

- An IO action can also be executed directly in the interactive shell

  ```
  > putStrLn "Hi!"
  Hi!
  ```

  so there's no need to go via `main` in the shell
- That means, in the shell we are in an `IO` environment
- Consequently, we have to use `let` to do "pure" stuff

IO Actions (6)

- In summary, a do block
  - introduces a sequence of statements
  - and executes these statements in order
- A statement can be one of the following:
  - an action
  - a `<-`, binding the (“pure”) result of an action
  - a `let`, expressing “pure” definitions

Monads

- The principle used for IO actions can be generalized and not only applied to IO
- Haskell uses the concept of a *monad* to handle "impurity"
  - For example, for IO, non-determinism, and exceptions
- We are going to introduce the general principle a bit later
- First, we are going to look at another example where Haskell meets the messy "real world"

Handling Errors

- Sometimes things go wrong, i.e., a function is not able to return a value
- For example, if we call `head` on an empty list, we get an error
- We don't necessarily want the program to just stop working and output an error in a case like that
- However, a function always has to return a value
- So we have to be able to handle the concept of failure (which is "impure" in Haskell's eyes)

Handling Errors (2)

- Haskell offers the type constructor `Maybe` that has a type parameter:

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

- Now we can “wrap” the result of a function call inside of a `Maybe`
- If the function call was successful, we hand it to the type constructor `Just`
- Otherwise, it becomes `Nothing`

Handling Errors (3)

- Let's write an alternative version of head that can cope with empty lists

```
safeHead :: [a] -> Maybe a
safeHead [] = Nothing
safeHead (x:xs) = Just x

> safeHead [1,2,3]
Just 1
> safeHead []
Nothing
```

- However, this comes at a price: we've introduced impurity into our function

Handling Errors (4)

- We cannot just take the result of `safeHead` and use it in other pure functions:

  ```
  doubleMe (safeHead [1,2,3])
  ```

  will raise an error
- `Maybe` is an instance of `Functor`
  - Quick reminder: a functor can be seen as content “wrapped” in a box
  - Haskell does not allow the concept of failure to escape its impure box
- So we have to get inside of the box

Handling Errors (5)

- `fmap` gets us on the inside of `Maybe`

```
> fmap doubleMe (safeHead [1,2,3])
Just 2
> fmap doubleMe (safeHead [])
Nothing
```

- If there is `Nothing` inside, `fmap` will not even apply the function, but return `Nothing`
- Just as a sidenote: IO is also a functor

```
main = do
    putStrLn "What's your name?"
    name <- fmap reverse getLine
    putStrLn ("!" ++ name ++ " iH")
```

Beefing Up Functors

- What happens if we want to use a function that takes two parameters with a functor?
- For example, we have two values `Just 2` and `Just 5` and want to multiply them:

  `(Just 2) * (Just 5)`

  does not work, as `*` expects two numerical values, not two values wrapped in `Maybe`

  - Again: pure function, impure parameters. . .
- We could push `*` into one of the functors

```
> :t fmap (*) (Just 2)
fmap (*) (Just 2) :: Num a => Maybe (a -> a)
```

Beefing Up Functors (2)

- That means, we now have function wrapped in a `Just`
- We could also rewrite the above as

```
Just (*2)
```

- This is a partially evaluated function (remember currying?)
- Now we have problem, though:
  - How do we get this into the other functor box?
- `fmap` only takes ordinary functions and maps them over a functor, so the following does not work:

```
fmap (Just(*2)) (Just 5)
```

Beefing Up Functors (3)

- So, what do we do? Rewrite all our multi-parameter functions for functors?
- No, it's not that bad, there is a type class called `Applicative`, which has two important functions

```
pure :: a -> f a
(<*>) :: f (a -> b) -> f a -> f b
```

- The second function is exactly what we are looking for, remember:

```
>:t Just (*2)
Just (*2) :: Num a => Maybe (a -> a)
> :t Just 5
Just 5 :: Num a => Maybe a
```

Applicative Functors

- Guess what, `Maybe` is an instance of `Applicative`, so we can use it right out of the box
  - Well, we have to import the module `Control.Applicative` first. . .

```
> import Control.Applicative
> (Just (*2)) <*> (Just 5)
Just 10
```

Success!

- This also works for values of `Nothing`

```
> (Just (*2)) <*> Nothing
Nothing
> Nothing <*> (Just 5)
Nothing
```

Applicative Functors (2)

- So what does `pure` do?
- It "wraps" a pure value into an impure context
- We cannot combine pure and impure values in the same computation
- With applicative functors we wrap the pure value into a (default) impure context

  ```
  (Just (*2)) <*> 5
  ```
  does not work

  ```
  (Just (*2)) <*> (pure 5)
  ```
  does work

Applicative Functors (3)

- This does not stop at two parameters, with applicative functors we can chain any number of functors:

```
pure f <*> x <*> y <*> z <*> ...
```

- So, for example we define a function summing up three numbers

```
sum3 x y z = x + y + z
```

and then use it in a functor context

```
> pure sum3 <*> Just 4 <*> Just 9 <*> Just 2
Just 15
> pure sum3 <*> Just 4 <*> Nothing <*> Just 2
Nothing
```

Monads

- We will now introduce the concept of monads with the help of an example
- Let's assume x persons want to divide up y things:

```
divideUp :: Int -> Int -> Int
divideUp x y = div y x
```

is not going to work, as the following shouldn't work but fail:

```
> divideUp 5 12
2
```

Second Try

- We could give back `Maybe Int`, so if the function fails, we return `Nothing`

```
divideAmong :: Int -> Int -> Maybe Int
divideAmong x y =
    if mod y x /= 0 then
        Nothing
    else
        Just (div y x)

> divideAmong 5 12
Nothing
```

- So far, so good

Further Divisions

- What happens if we want to divide up one lot among further persons, i.e.:

  ```
  divideAmong 3 (divideAmong 2 12)
  ```

- This is not going to work, as `divideAmong` expects pure Ints
  - The number of things and persons don't fail, we're sure about them
- Let's try using an applicative functor:

  ```
  > pure (divideAmong) <*> Just 2 <*> Just 12
  Just (Just 6)
  ```

  Nope, this adds yet another layer...

Further Divisions (2)

- Is implementing this manually the only option left?

```
divideAmongTwice :: Int -> Int -> Int -> Maybe Int
divideAmongTwice x y z =
    if mod y x /= 0 then
        Nothing
    else
        if mod (div y x) z /= 0 then
            Nothing
        else
            Just (div (div y x) z)
```

- Keeping track of every step that can fail is very awkward

Monads, Finally

- Monads can help out here
- They are a type class having two important functions:

```
return :: m -> m a
(>>=) :: m a -> (a -> m b) -> m b
```

- The first function, `return`, works like `pure` for applicative functors
  - Meaning it wraps a pure value into an impure context
- The second one, called bind, allows us to apply a function such as our `divideAmong` to an impure context
  - And it returns its result in an impure context

Monads, Finally (2)

- Now we can chain together calls of the function

```
> divideAmong 2 120 >>= divideAmong 3 >>= divideAmong 5
Just 4
```

- And Haskell will keep track of any failures on the way for us

```
> divideAmong 5 12 >>= divideAmong 3 >>= divideAmong 5
Nothing
> divideAmong 6 12 >>= divideAmong 3 >>= divideAmong 5
Nothing
```

Do Notation

- Monads are so important in Haskell that they have their own special notation, the do notation
- This notation allows you to chain together monadic function calls in a seemingly imperative way

```
routine :: Maybe Int
routine = do
    x <- divideAmong 2 120
    y <- divideAmong 3 x
    divideAmong 4 y
```

- The statements are executed line by line
- With `<-` we bind a monadic `Maybe` value to a variable
- The result of the final execution is the result of `routine`

IO is a Monad

- Yes, you have seen this notation before in the context of IO
- And, yes, this means that IO is a monad!
- It doesn't end there:
  - There are monads for representing state
  - For dealing with indeterminism
  - Even lists can be interpreted as monads
- There are lots of other things to say about monads
  - All instance of monads need to follow certain laws (instances of (applicative) functors as well)
- But we are going to stop here

Mathematical Foundation

- The concepts used in Haskell did not just fall from the sky
- They are rooted in mathematical theory, *category theory* to be more specific
- In category theory, mathematicians try to capture the underlying properties of mathematical concepts
- Expressed in simplified terms, it is like finding and defining “type classes” for mathematical structures

Summary

- Strengths of Haskell
  - The type system (strong/static) will prevent you from making a lot of mistakes
    - Nevertheless, it is quite flexible when it comes to extending it with user-defined types
  - Haskell offers a lot in terms of expressiveness, you can write very concise code
  - It is easier to show the correctness of your programs, due to the pure functional style
  - It does lazy evaluation, which gives you an additional tool for writing programs efficiently

Summary (2)

- Weaknesses of Haskell
  - The pure functional paradigm also has a price: dealing with messy real-world situations such as IO and state is not easy
  - Haskell has a steep learning curve, it takes a while to learn how to wield the power of Haskell
  - This may also explain the fact that the Haskell community is relatively small