Logic - CM0845
Introduction to Haskell

Diego Alejandro Montoya-Zapata

EAFIT University

Semester 2016-1

What is Haskell?

Haskell is a **purely functional** programming language. That means that every function in Haskell is also a function in the mathematical sense.

Example

```
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

Functions

Example

```
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

Functions

Example

```
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

What is the type of this function?

Functions

Example

```
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

What is the type of this function?

```
factorial :: Int -> Int
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

Functions

Example

```
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

What is the type of this function?

```
factorial :: Int -> Int
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

But -1 is an Integer, so...

Functions

A solution for this bug:

```
factorial :: Int -> Int
factorial n
  | n == 0 = 1
  | n > 0 = n * factorial (n - 1)
  | otherwise = error "factorial: n < 0"
```

Functions

A solution for this bug:

```
factorial :: Int -> Int
factorial n
  | n == 0 = 1
  | n > 0 = n * factorial (n - 1)
  | otherwise = error "factorial: n < 0"
```

There are more than you believe!

Google for “The evolution of a Haskell programmer”.

Lists

Inductive definition Haskell has a built-in syntax for lists, where a list is either:

- the empty list, written **[ ]**, or
- an element **x** and a list **xs**, written (**x** : **xs**).

Lists

Example - Pattern matching on lists

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

Lists

Example - Pattern matching on lists

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

What if one wanted to get the length of a list of Booleans?

Lists

Example - Pattern matching on lists

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

What if one wanted to get the length of a list of Booleans?

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

Lists

Example - Pattern matching on lists

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

What if one wanted to get the length of a list of Booleans?

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

Take it easy, there's another solution!

Parametric Polymorphism

Example - Basic functions

```
-- Returns the length of a finite list as an Int.
length :: [a] -> Int

-- Appends two lists.
(++) :: [a] -> [a] -> [a]

-- Extracts the first element of a list.
head :: [a] -> a

-- Extracts the last element of a list.
last :: [a] -> a
```

Example - Basic functions

```
-- Extracts the elements after the head of a list.
tail :: [a] -> [a]

-- Returns all the elements of a list except
-- the last one.
init :: [a] -> [a]

-- Testes if a list is empty.
null :: [a] -> Bool
```

Lazy

Haskell wont't execute functions or calculate things until necessary.

Example

```
foo :: Int -> Bool -- Non-terminating function.
foo n = foo (n + 1)

bar :: Int -> Bool
bar n = True || foo n

bar' :: Int -> Bool
bar' n = foo n || True
```

Lazy

Haskell wont't execute functions or calculate things until necessary.

Example

```
foo :: Int -> Bool -- Non-terminating function.
foo n = foo (n + 1)

bar :: Int -> Bool
bar n = True || foo n

bar' :: Int -> Bool
bar' n = foo n || True
```

Try to calculate `bar 3`.

Lazy

Haskell wont't execute functions or calculate things until necessary.

Example

```
foo :: Int -> Bool -- Non-terminating function.
foo n = foo (n + 1)

bar :: Int -> Bool
bar n = True || foo n

bar' :: Int -> Bool
bar' n = foo n || True
```

Try to calculate `bar 3`.
Try to calculate `bar' 3`.

High-Order Functions and Currying

Every function in Haskell officially only takes one parameter. So how can we define a function that takes more than a parameter?

```
-- Takes two things that can be ordered and
returns the greater one.
max :: (Ord a) => a -> a -> a
```

Example

- `max 2 3`

High-Order Functions and Currying

Every function in Haskell officially only takes one parameter. So how can we define a function that takes more than a parameter?

```
-- Takes two things that can be ordered and
returns the greater one.
max :: (Ord a) => a -> a -> a
```

Example

- `max 2 3`
- `(max 2) 3`

High-Order Functions and Currying

Every function in Haskell officially only takes one parameter. So how can we define a function that takes more than a parameter?

```
-- Takes two things that can be ordered and
returns the greater one.
max :: (Ord a) => a -> a -> a
```

Example

- `max 2 3`
- `(max 2) 3`

Haskell functions can take functions as parameters and return functions as return values!

High-Order Functions

Example

```
-- map f xs is the list obtained by applying f
-- to each element of xs.
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x : xs) = f x : map f xs
```

Which is the value of `map (*2) [1, 2, 4]`?

High-Order Functions

Example

```
-- map f xs is the list obtained by applying f
-- to each element of xs.
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x : xs) = f x : map f xs
```

Which is the value of `map (*2) [1, 2, 4]`?

```
GHCi> map (*2) [1, 2, 4]
[2,4,8]
```

Example

```
-- foldr, applied to a binary operator, a starting
-- value and a list, reduces the list using th
-- binary operator, from right to left (see also
-- foldl):
-- foldr f z [x1, x2, ..., xn] ==
-- x1 `f` (x2 `f` ... (xn `f` z)...)

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

Example

```
-- foldr, applied to a binary operator, a starting
-- value and a list, reduces the list using th
-- binary operator, from right to left (see also
-- foldl):
-- foldr f z [x1, x2, ..., xn] ==
-- x1 `f` (x2 `f` ... (xn `f` z)...)

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

```
GHCi> foldr (*) 1 [1..5]
120
```

Creating Types - Algebraic Data Types

Example

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

Functions by pattern-matching

```
(||) :: Bool -> Bool -> Bool
True || _ = True
False || x = x

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

Creating Types - Algebraic Data Types

Example

```
-- Recursive data type.
data Nat = Zero | Succ Nat
```

Functions by pattern-matching

```
(+) :: Nat -> Nat -> Nat
Zero + n = n
(Succ m) + n = Succ (m + n)
```

Example

```
-- Polymorphic data type.
data List a = Nil | Cons a (List a)
```

Some Links

- **Real-World Applications**

  See http://www.haskell.org/haskellwiki/Haskell_in_industry.

- **Nice Tutorial**

  See http://learnyouahaskell.com.

- **Downloading**

  See https://www.haskell.org/downloads.