Introduction to Functional Programming with Haskell - Part 2

Norton Jenkins

August 14, 2013

Table of Contents

Data in Haskell

- Data has a different Type

Data in Haskell

- Data has a different Type
- Int and Char are the most common.

Data in Haskell

- Data has a different Type
- Int and Char are the most common.
- Data can be abstracted into Tupples and lists

Data in Haskell

- Data has a different Type
- Int and Char are the most common.
- Data can be abstracted into Tupples and lists
- Typeclasses are interfaces for similar data.

Working with lists

- Lists collect data of the same type.

Working with lists

- Lists collect data of the same type.
- Comprehensions allow you to filter lists to your liking.

Working with lists

- Lists collect data of the same type.
- Comprehensions allow you to filter lists to your liking.
- Mapping over lists returns a list of data with a function applied to it.

Working with lists

- Lists collect data of the same type.
- Comprehensions allow you to filter lists to your liking.
- Mapping over lists returns a list of data with a function applied to it.
- The notation [Type] creates a list. [Bool] is a list of booleans.

Working with Tupples

- Tupples “types” are determined by the number of elemnts they have, and what types are in what location.
- (Int,Char) is a valid tupple.

Pattern matching

- Any data can be pattern matched.

Pattern matching

- Any data can be pattern matched.
- Asks “Does the data look like this?”

Pattern matching

- Any data can be pattern matched.
- Asks “Does the data look like this?”
- The list [1,2] is syntax sugar for 1:2

Pattern matching

- Any data can be pattern matched.
- Asks “Does the data look like this?”
- The list [1,2] is syntax sugar for 1:2
- The pattern (x:xs:[]) matches 1 is then bound to x, 2 to c, and they are indeed appended to an empty list, so the match completes.

Pattern matching

- Any data can be pattern matched.
- Asks "Does the data look like this?"
- The list [1,2] is syntax sugar for 1:2
- The pattern (x:xs:[]) matches 1 is then bound to x, 2 to c, and they are indeed appended to an empty list, so the match completes.
- (a,b) = (1,2). You now have two variables. a that is 1, b that is 2.

Pattern matching

- Any data can be pattern matched.
- Asks "Does the data look like this?"
- The list [1,2] is syntax sugar for 1:2
- The pattern (x:xs:[]) matches 1 is then bound to x, 2 to c, and they are indeed appended to an empty list, so the match completes.
- (a,b) = (1,2). You now have two variables. a that is 1, b that is 2.
- (a:b:_) = [1,2]. This will bind variables the same as above.

Pattern Matching by value

- Take this function: isThisNumberZero x = if x == 0 then True else False

Pattern Matching by value

- Take this function: isThisNumberZero x = if x == 0 then True else False

This can be written as:

```
isThisNumber0 0 = True
isThisNumber0 _ = False
```

- If x is 0, it returns True. If it is ABSOLUTELY ANYTHING ELSE (special underscrore sugar) it returns False.

Pattern Matching outside of Functions

- Everything in Haskell returns a value.

Pattern Matching outside of Functions

- Everything in Haskell returns a value.
- Inline pattern matching is no exception. Use the **case** keyword.

Pattern Matching outside of Functions

- Everything in Haskell returns a value.
- Inline pattern matching is no exception. Use the **case** keyword.

```
let y = [1,2] in let x = case y of
                           []     -> 0
                           [x]    -> 1
                           (x:xs) -> 2
```

The above code makes a variable y equal to [1,2]. It's pointless to check, but if y is empty, x becomes 0. If it has one element [x] it becomes 1. Otherwise, it will become 2 as long as it has two elements.