Haskell Intro

Dhananjay and Rémi

November 9, 2016

Haskell

Haskell - Let's get our hands dirty

- Introduction on basic features
  - Functions
  - Basic types
  - Lists
  - Pattern-matching
- Practice
- Some mind-bending exercises to get up-to-speed

Install the env

Install stack, including everything you need!
https://docs.haskellstack.org/en/stable/README/
Then create a file named `code.hs` with content:

```haskell
module Session1 where

main = putStrLn "Hello World!"
```

Run `stack runhaskell code.hs`. You're good to go!

REPL

Use **stack ghci** to run the REPL in the directory where your code is:

```
Prelude> :load code -- Loads your module
[1 of 1] Compiling Session1         ( code.hs, interpreted )
Ok, modules loaded: Session1.
```

REPL

Use **stack ghci** to run the REPL in the directory where your code is:

```
Prelude> :load code -- Loads your module
[1 of 1] Compiling Session1         ( code.hs, interpreted )
Ok, modules loaded: Session1.

*Session1> main -- Calls `main` from loaded module
Hello World
```

REPL

Use **stack ghci** to run the REPL in the directory where your code is:

```
Prelude> :load code -- Loads your module
[1 of 1] Compiling Session1         ( code.hs, interpreted )
Ok, modules loaded: Session1.

*Session1> main -- Calls `main` from loaded module
Hello World

*Session1> :type [1, 2] -- Displays type of an expression
[1, 2] :: Num t => [t]

*Session1> :info main
main :: IO ()    -- Defined at code.hs:18:1
```

Types

Type annotations are of the form `expression :: type`. Examples of types:

Types

Type annotations are of the form `expression :: type`. Examples of types:

```
42 :: Int
42.0 :: Float
'a' :: Char
[1, 2] :: [Int] -- List of Ints
"Hello" :: [Char] -- :'(
length :: [a] -> Int
```

Types

Type annotations are of the form `expression :: type`. Examples of types:

```
42 :: Int
42.0 :: Float
'a' :: Char
[1, 2] :: [Int] -- List of Ints
"Hello" :: [Char] -- :'(
length :: [a] -> Int
```

Use `:type` in `ghci` to query the type of more complex expressions. Like: `(^)`, `(++)`, `map`, `(:)`

Lists

`List` is the most ubiquitous data structure in Haskell.

- *Linked list* of elements.
- With *same type* `:: [a]` (not heterogeneous)

Lists

`List` is the most ubiquitous data structure in Haskell.

- *Linked list* of elements.
- With *same type* `:: [a]` (not heterogeneous)

A list can be **empty**:

```
*Session1> []
```

Lists

`List` is the most ubiquitous data structure in Haskell.

- *Linked list* of elements.
- With *same type* `:: [a]` (not heterogeneous)

A list can be **empty**:

```
*Session1> []
```

Or a **head** and a **tail**:

```
*Session1> 42 : []
*Session1> 1 : 2 : 3 : []
```

Lists

`List` is the most ubiquitous data structure in Haskell.

- *Linked list* of elements.
- With *same type* `:: [a]` (not heterogeneous)

A list can be **empty**:

```
*Session1> []
```

Or a **head** and a **tail**:

```
*Session1> 42 : []
*Session1> 1 : 2 : 3 : []
```

Syntactic sugar:

```
*Session1> [1]
*Session1> [1, 2]
*Session1> [1..5] -- [1,2,3,4,5]
```

Functions

Functions are declared with an (optional) **type signature**:

```haskell
factorial :: Int -> Int
-- ^         ^   ^      ^
-- |         |   |      result is an Int
-- |         |   first argument is an Int
-- |         type of `factorial`
-- name of function
```

Functions

Functions are declared with an (optional) **type signature**:

```
factorial :: Int -> Int
-- ^         ^   ^       ^
-- |         |   |       result is an Int
-- |         |   first argument is an Int
-- |         type of `factorial`
-- name of function
```

And one or more body with **pattern matching**:

```
factorial 0 = 1
factorial n = n * factorial (n - 1)
--          ^       ^
--          arg 1   recursive call
```

Functions on lists

You have to deal with at least two cases:

- The list is **empty**
- The list has a **head** and a **tail**

Functions on lists

You have to deal with at least two cases:

- The list is **empty**
- The list has a **head** and a **tail**

```haskell
listFunction :: [a] -> ?
listFunction [] = -- Empty case
listFunction (x:xs) = -- Other cases
```

Functions on lists

You have to deal with at least two cases:

- The list is **empty**
- The list has a **head** and a **tail**

```
listFunction :: [a] -> ?
listFunction [] = -- Empty case
listFunction (x:xs) = -- Other cases
```

You can also match “one element” or “two elements”:

```
listFunction [a] = -- One element
listFunction [a, b] = -- two elements
listFunction [a, 42] = -- second element must be `42`
```

Example - Zip

The signature:

```
  zip :: [a] -> [b] -> [(a, b)]
-- ^    ^   ^                 ^
-- |    |   |                 result is a list of tuples
-- |    |   first argument is a list of `a`
-- |    type of `zip`
-- name of function
```

Example - Zip

The signature:

```
  zip :: [a] -> [b] -> [(a, b)]
-- ^    ^  ^                ^
-- |    |  |                result is a list of tuples
-- |    |  first argument is a list of `a`
-- |    type of `zip`
-- name of function
```

Different cases:

```
zip [] _ = []
zip _ [] = []
zip (x1:xs1) (x2:xs2) = (x1, x2) : zip xs1 xs2
--      ^         ^          ^    ^         ^
--      arg 1     arg2       tuple  cons    recursive call
```

Let's play!

Add the following in `code.hs` file.
It hides some standard functions:

```
module Session1 where

import Prelude hiding
    ( concat
    , filter
    , foldl
    , foldr
    , length
    , map
    , product
    , reverse
    , sum
    )
```

Length

Implement the `length` function:

```haskell
length :: [a] -> Int

-- Example:
-- length [] == 0
-- length [1, 2, 3, 4, 5] == 5
```

Double list

Double all elements of a list:

```
doubleList :: [Int] -> Int

-- Examples:
-- doubleList [] == []
-- doubleList [1, 2, 3] == [2, 4, 6]
```

How can we **generalize** this pattern?

Map

Implement the function `map` with following type:

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

-- Example:
-- map (1+) [1, 2, 3, 4] == [2, 3, 4, 5]
-- map (*2) [1, 2, 3, 4] == [2, 4, 6, 8]
```

Sum/Product/Fold

Implement a `sum` function which takes a list of numbers and return the sum of all of them:

```
sum :: Num a => [a] -> a
```

Sum/Product/Fold

Implement a `sum` function which takes a list of numbers and return the sum of all of them:

```
sum :: Num a => [a] -> a
```

Implement the `product` function which does the multiplication of all the elements of a list:

```
product :: Num a => [a] -> a
```

Sum/Product/Fold

Implement a `sum` function which takes a list of numbers and return the sum of all of them:

```haskell
sum :: Num a => [a] -> a
```

Implement the `product` function which does the multiplication of all the elements of a list:

```haskell
product :: Num a => [a] -> a
```

Can you identify a pattern here? This is `fold`. Implement the `fold` function:

```haskell
fold :: (b -> a -> b) -> b -> [a] -> b
```

Is there another possible implementation?

Filter

Implement the `filter` function with following signature:

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

-- Example:
-- filter even [1, 2, 3, 4, 5, 6, 7] == [2, 4, 6]
```

Hint:

```haskell
if test then expression1 else expression2
```

Hint2: `if then else` is an expression in Haskell!

Sum of odd elements

Given a list of integers, output the sum of the odd numbers:

```haskell
sumOdds :: [Int] -> Int

-- Example:
-- sumOdds [1, 2, 3, 4] == 1 + 3 == 4
```

Reverse

Implement the `reverse` function with the following signature:

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

-- Example:
-- reverse [1, 2, 3] == [3, 2, 1]
```

Concat

Implement the `concat` function with the following signature:

```
concat :: [a] -> [a] -> [a]

-- Examples:
-- concat [1, 2, 3] [4, 5] == [1, 2, 3, 4, 5]
-- concat [] [4, 5] == [4, 5]
-- concat [1, 2, 3] [] == [1, 2, 3,]
```

Palindrom

Check if a list if a *palindrom*:

```haskell
palindrom :: Ord a => [a] -> Bool

-- Examples:
-- palindrom "otto" == True
-- palindrom [1, 2, 3] == False
-- palindrom [1, 2, 3, 2, 1] == True
```

Run-length encoding (RLE)

Compress a string with `RLE`:

```
rle :: [Char] -> [(Int, Char)]
-- rle :: String -> [(Int, Char)]

-- Examples:
-- rle "" == []
-- rle "abba" == [(1, 'a'), (2, 'b'), (1, 'a')]
```

Merge sort

Implement a merge sort on lists:

```
mergeSort :: Ord a => [a] -> [a]

-- Examples:
-- mergeSort [] == []
-- mergeSort [1, 2, 4, 3] == [1, 2, 3, 4]
-- mergeSort [3, 2, 1] == [1, 2, 3]
-- mergeSort "Hello" == "Hello"
```

Hint implement two helper functions:

```
split :: [a] -> ([a], [a])
merge :: Ord a => [a] -> [a] -> [a]
```