Haskell Intro

Dhananjay and Rémi

November ?, 2016


Haskell

Haskell - Let's get our hands dirty

- Introduction on basic features
- Practice
- Some mind-bending exercises to get up-to-speed

Install the env

Install haskell platform, including everything you need!
https://www.haskell.org/platform/
Then create a file named `code.hs` with content:

```
module Main 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.

*Main> main -- Calls function `main` from loaded module
Hello World

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

*Main> :info [1, 2]
data [] a = [] | a : [a]      -- Defined in ‘GHC.Types’
instance Eq a => Eq [a]       -- Defined in ‘GHC.Classes’
...
```

Use `:type` and `:info` on `main`!

Types

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

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

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

Functions

Functions are declared with an (optional) type signature:

```
zip :: [a] -> [b] -> [(a, b)]
```

And one or more body with *pattern matching*:

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

Or just:

```
zip list1 list2 = ...
-- Deal with different patterns in the body
-- of the function directly (we will see how later)
```

Types

Lists

`List` is the most ubiquitous data structure in Haskell. It's implemented as a *linked list* of elements with *same type* `::` `[a]`.

```
*Main> :info []
data [] a = [] | a : [a] -- Defined in ‘GHC.Types’
--   ^  ^   ^       ^
--   |  |   |       constructor 2 (cons)
--   |  |   constructor 1 (empty list)
--   |  type parameter
--   type name
```

List is either empty or one element with the rest of the list.

```
*Main> []
*Main> [1]
*Main> [1, 2]
*Main> 1 : [2]
*Main> 1 : 2 : []
*Main> [1..2] -- Syntactic sugar for range
```

Pattern matching

In Haskell you can pattern-match on data types

Let's play!

Add the following lines in your file to create a module and hide the standard functions:

```
module Session1 where

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

Functions on list

You have to deal with at least two cases:

- The list is empty
- The list has at least one element

```
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`
```

Length

Implement the `length` function:

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

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

Map

Implement the function `map` with following type:

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

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

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
```

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

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

Is there another possible implementation?

Filter

Implement the `filter` function with following signature:

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

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

Sum of odd elements

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

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

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

Reverse

Implement the `reverse` function with the following signature:

```
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,]
```