NAMAL COLLEGE

UNIVERSITY of BRADFORD

Symbolic & Declarative Computing / Artificial Intelligence (COS5012-B)

November 08, 2016 Week 7 Lab1

LAB MANUAL: FUNCTION COMPOSITION + SOME OTHER HOFs

In this tutorial you will learn about

- Function Composition
- Function Application
- Iterate
- takeWhile
- dropWhile

Lambda Tribute

- Lambda calculus is due to the logician Alonzo Church (1903–1995).
- The lambda calculus is a theory of functions, that was designed before computers existed.
- Lambda expressions finally came to Java in 2014, only about 55 years after they came to functional programming.

Warm up

1. Can you guess the name of the higher order function

```
a. ? f xs = [ f x | x <- xs ]
b. ? pred xs = [ x | x <- xs, pred x ]
```

2. Write a definition of `zipWith` using above notation.
3. Implement `factorial` using `fold`.
4. Write a function `reverseString` to reverse a string.

Function Composition (.)

Composition is a binary operator represented by an infix full stop: `(f.g) x` is equivalent to `f (g x)`. The type of the section `(.)` is `(a -> b) -> (c -> a) -> c -> b`. Function composition is useful for many reasons. One of them is that `f (g ( h (i (j (k x)))))`, say, can be written as `(f . g . h . i . j . k ) x`.

Example

Find the sum of the cubes of all the numbers divisible by 7 in a list xs of integers.

```
answer :: [Int] -> Int
answer xs = sum (map cube (filter by7 xs))


cube :: Int -> Int
```

```
cube x = x * x * x


by7 :: Int -> Bool
by7 x = x `mod` 7 == 0
```

- Using function composition the above code can be written as

```
answer' :: [Int] -> Int
answer' xs = (sum . map cube . filter by7) xs
```

The composition operator `(.)` can be defined as below

```
(.) :: (b -> c) -> (a -> b) -> a -> c
f . g = \x -> f (g x)
```

Exercise

- Write the function composition equivalent of possqr

```
sqr :: Int -> Int
sqr x = x * x
pos :: Int -> Bool
pos x = x > 0
possqr :: Int -> Bool
possqr x = pos (sqr x)
```

- Write a function using function composition to compute the length of head of a list of strings.

Function Application ($)

The `($)` function is called a function application. It makes functions right associative.

```
Hugs> not odd 4  -- It results in an error
Hugs> not $ odd 4  -- This is equal to not (odd 4)
True
```

Higher Order Function: Iterate

`iterate` produces a list by applying a function an increasing number of times to a value:

`iterate f x = [x ,f x, f (f x), f (f (f x)), · · ·]`

```
Hugs> iterate (*2) 1  -- What happened?
Hugs> take 10 (iterate (*2) 1)
[1,2,4,8,16,32,64,128,256,512]
Hugs> take 10 (iterate (*2) 10)
```

```
?
```

Exercise

- Write a statement using iterate to print numbers from `0..9`
- Create a list of first 40 even numbers using `iterate`
- Define a Haskell variable **`dollars`** that is the infinite list of amounts of money you have every year, assuming you start with $100 and get paid 5% interest, compounded yearly. (Ignore inflation, deflation, taxes, bailouts, the possibility of total economic collapse, and other such details.) So **`dollars`** should be equal to `[100.0, 105.0, 110.25, ...]`.
- Write down the definition of higher order function **`iterate`**.

Higher Order Functions: dropWhile and takeWhile

```
Hugs> takeWhile (\x -> x /= 5) [2,4,5,8,2,5]
[2,4]
Hugs> dropWhile (\x -> x /= 5) [2,4,5,8,2,5]
[5,8,2,5]
```

Haskell Prelude

File uploaded on LMS.