Class 1: Some Standard Prelude Functions*, v1.5

Jacek Malec (but see Introduction)

March 23, 2018

1 Introduction

This document is a derivative of many sources, most notably (in no particular order):

1. "Haskell" on WikiBooks: `http://en.wikibooks.org/wiki/Haskell/`;
2. Bryan O'Sullivan, John Goerzen and Don Stewart, "Real World Haskell", O'Reilly, 2009;
3. Simon Thompson, "The Craft of Functional Programming", 2nd ed., (formally our course book), Addison-Wesley, 1999;
4. Bernie Pope, "A tour of the Haskell Prelude", 2001;
5. Manuel M. T. Chakravarty, course materials COMP1011 at University of New South Wales;
6. Materials of TDA555 from Chalmers, `http://www.cse.chalmers.se/edu/course/TDA555/`.

2 Exercises

2.1 Basics

2.1.1 The Maximum Function (TDA555)

Write definition of a function `maxi x y` computing the maximum of `x` and `y`. Do not use the function `max` in your definition, of course.

2.1.2 Sum of squares (TDA555)

Define a function that computes the sum of the squares of the numbers from 1 to n.

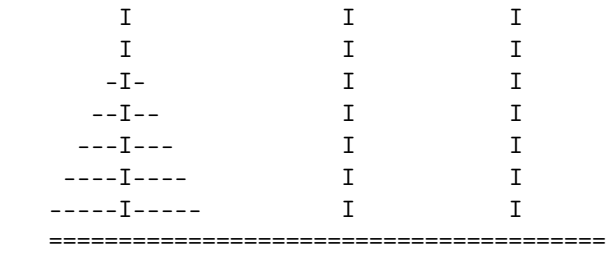
```
-- sumsq n returns 1*1 + 2*2 + ... + n*n
```

Use recursion.
Now use mapping instead.

*Intended for EDAF/N40 course, after first lectures.

2.1.3 The Towers of Hanoi (TDA555)

The Towers of Hanoi is an ancient puzzle, consisting of a collection of rings of different sizes, and three posts mounted on a base. At the beginning all the rings are on the left-most post as shown, and the goal is to move them all to the rightmost post, by moving one ring at a time from one post to another. But, at no time may a larger ring be placed on top of a smaller one!



Can you find a strategy for solving the puzzle based on recursion? That is, if you already know how to move n-1 rings from one post to another, can you find a way to move n rings?

If you try out your strategy, you will quickly discover that quite a large number of moves are needed to solve the puzzle with, say, five rings. Can you define a Haskell function

```
hanoi n
```

which computes the number of moves needed to move n rings from one post to another using your strategy? How many moves would be needed to solve the puzzle with ten rings? Legend has it that the original version of the puzzle has 32 rings[1], and is being solved at this very moment by Bhuddist monks in a monastery. When the puzzle is complete, the world will end.

2.1.4 Factors (TDA555)

A prime number p has only two factors, 1 and p itself. A composite number has more than two factors. Define a function

```
smallestFactor n
```

which returns the smallest factor of n larger than one. For example,

```
smallestFactor 14 == 2
smallestFactor 15 == 3
```

Hint: write `smallestFactor` using an auxiliary function `nextFactor k n` which returns the smallest factor of `n` larger than `k`. You can define `smallestFactor` using `nextFactor`, and `nextFactor` by recursion.

Now define

```
numFactors n
```

which computes the number of factors of `n` in the range `1..n`, possibly except factor 1.

[1] Wikipedia claims it has 64.

2.1.5 Defining Types (TDA555)

Define a data type `Month` to represent months, and a function

```
daysInMonth :: Month -> Integer -> Integer
```

which computes the number of days in a month, given also the year. (You can ignore leap centuries and the like: just assume that every fourth year is a leap year).

Define a data type `Date`, containing a year, month, and day, and a function

```
validDate :: Date -> Bool
```

that returns `True` if the day in the date lies between 1 and the number of days in the month.

2.2 Lists

2.2.1 Multiplying List Elements (TDA555)

Define a function

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

which multiplies together all the elements of a list. (Think: what should its value be for the empty list?). For example

```
Main> multiply [1,2,3,4,5]
120
```

(This is actually a standard function, called `product`).

2.2.2 Substitution (Chakravarty)

Define a function substituting elements in a list by another element. E.g.

```
Main> substitute 'e' 'i' "eigenvalue"
"iiginvalui"
```

2.2.3 Avoiding duplicates (TDA555)

In many situations, lists should not contain duplicate elements. For example, a pack of cards should not contain the same card twice. Define a function

```
duplicates :: Eq a => [a] -> Bool
```

which returns `True` if its argument contains duplicate elements.

```
Main> duplicates [1,2,3,4,5]
False
Main> duplicates [1,2,3,2]
True
```

Hint: the standard function `elem`, which tests whether an element occurs in a list, is helpful here.

One way to ensure a list contains no duplicates is to start with a list that might contain duplicate elements, and remove them. Define a function

```
removeDuplicates :: Eq a => [a] -> [a]
```

which returns a list containing the same elements as its argument, but without duplicates. Test it using the following property:

```
prop_duplicatesRemoved :: [Integer] -> Bool
prop_duplicatesRemoved xs = not (duplicates (removeDuplicates xs))
```

Does this property guarantee that `removeDuplicates` behaves correctly? If not, what is missing?
(`removeDuplicates` is actually a standard function, called `nub`).

2.2.4 Comprehensions (TDA555)

Describe what does the function

```
pairs :: [a] -> [b] -> [(a,b)]
pairs xs ys = [(x,y) | x<-xs, y<-ys]
```

do.

A Pythagorean triad is a triple of integers (a,b,c) such that

```
a^2 + b^2 == c^2
```

Define a function of `n` that will find all Pythagorean triads with `a<=b<=c<=n`.

2.2.5 Permutations (TDA555)

A *permutation* of a list is another list with the same elements, but in a possibly different order. For example, `[1,2,1]` is a permutation of `[2,1,1]`, but not of `[1,2,2]`. Write a function

```
isPermutation :: Eq a => [a] -> [a] -> Bool
```

that returns `True` if its arguments are permutations of each other.

2.2.6 Shortest and Longest (Chakravarty)

Determine the shortest and the longest string in a list. E.g.:

```
Main> shortestAndLongest ["abc"]
("abc","abc")
Main> shortestAndLongest ["This", "sentence", "is","ridiculous"]
("is","ridiculous")
```

2.2.7 Mystery (Thompson)

What does the following function do?

```
mystery xs = foldr (++) [] (map (\y -> [y]) xs)
```

Final Remark

Don't worry if you are not done with all the exercises within 2 hours. The idea is to give you some material for practicing your Haskell, feel free to skip what you don't like. Solutions can be found on respective source material web page.

For each exercise, look at your solution and ask a question: can I write this function differently? Can I use list comprehension here? Can I use mapping here? Can I use case-based definition? In most situations the answer is positive. You can then try to judge beauty of your solution, not only its formal correctness. Is always a short solution preferred to a longer one? Why or why not?