Programming in Haskell – Homework Assignment 1

UNIZG FER, 2013/2014

Handed out: October 11, 2013. Due: October 24, 2013 at 17:00

Note: Define each function with the exact name specified. You can (and in most cases you should) define each function using a number of simpler functions. Unless said otherwise, a function may not cause runtime errors and must be defined for all of its input values. Use the `error` function for cases in which a function should terminate with an error message. Problems marked with a star ($\star$) are optional.

1. Define `splitAt'` as a function that behaves exactly like `Prelude.splitAt` but is implemented without it. (Hint: see `Prelude.splitAt`'s source code defined within the `Prelude` module on Hackage.)

   ```
   splitAt' 1 [1..10] ⇒ ([1], [2..10])
   splitAt' (-3) [1..10] ⇒ ([], [1..10])
   splitAt' 3 [1..] ⇒ ([1,2,3], [4..])
   splitAt' 7 [] ⇒ ([], [])
   ```

2. A common mistake when checking whether a list is empty is to use the `length` function, which will hang on infinite lists. The proper way to do it is to use the `null` function, which only checks for the existence of a single element. Define a function `longerThan n` that checks whether a list has more than `n` elements. Make sure that it never hangs on infinite lists. (Hint: `drop`.)

   ```
   longerThan 0 "abc" ⇒ True
   longerThan 0 [] ⇒ False
   longerThan 1000 [1..] ⇒ True
   longerThan (-5) any ⇒ True
   ```

3. Define the `palindrome` function that checks whether a list is a palindrome. A palindrome is any list of elements that is invariant under reversion (i.e., equal when reversed). Define this function without using `reverse`! (Hint: use `head`, `last`.)

   ```
   palindrome "level" ⇒ True
   palindrome [1,2,3] ⇒ False
   palindrome [] ⇒ True
   ```

4. Define the `pangram` function for checking whether a given string is a pangram. A pangram is any sequence of characters that contains every letter of the alphabet at least once. Assume the input string consists of only spaces (' ') and lower-case letters of the English alphabet. (Hint: `nub`, `sort`.)

5. (a) Define the `rotl` function for rotating a list of elements to one side. A positive number rotates the list to the left, while a negative rotates the list to the right.

```
rotl 1 "abc" ⇒ "bca"
rotl (-4) "abc" ⇒ "cab"
rotl 0 "abc" ⇒ "abc"
rotl 1234 [] ⇒ []
```

(b)⋆ Redefine the function so that it also works on infinite lists. The function should hang (i.e., not compute; henceforth denoted with '⊥') only when rotating an infinite list to the right.

```
rotl (-1) [1..] ⇒ ⊥
```

6. Define the `put` function for overwriting an element at a list position with a given value. A position can be negative: in that case we start counting from the end of the list. A position greater than the list's length can also be given: in that case the position "wraps over" the edge of the list. If the input list is empty, insert the element into the list. (Hint: `cycle`.)

```
put 7 0 [1..3] ⇒ [7,2,3]
put 'x' 4 "abc" ⇒ "axc"
put 7 any [] ⇒ [7]
put 7 (-2) [1..5] ⇒ [1,2,3,7,5]
```

7. Define the `median` function for computing the median of a list of values. A median is the middle (or center) value of a sorted list of values. If the list is of even length, the median equals the mean of the two middle values. Return zero when given an empty list. (Hint: use '`/`' for division.)

```
median [3,2,4,10,1] ⇒ 3.0
median [10,0,6,2,8,3,9,9,3,5] ⇒ 5.5
median [] ⇒ 0.0
median [1..] ⇒ ⊥
```

8. Let's use lists as sets, even though this isn't recommended for performance reasons. Define `intersect'` and `difference` that implement the usual set operators. Assume the list elements don't have an ordering (i.e., don't use operators such as `<` or `<` to compare them, and also don't sort them), but do assume we can test them for equality. Both input and output lists may have duplicate elements. (Hint: use list comprehensions.)

```
intersect' "mio" "mao" ⇒ "mo"
intersect' [1..3] [1..] ⇒ [1..3]
intersect' [1..]  [1..3] ⇒ [1,2,3,⊥]
intersect' [] [1..] ⇒ []
intersect' [1..]  [] ⇒ []

difference "mio" "mao" ⇒ "i"
difference [4,3] [1,3,2,12] ⇒ [4]
difference [4,3] [1..] ⇒ []
difference [4,3] [5..] ⇒ [⊥]
difference [] [1..] ⇒ []
difference [1..]  [] ⇒ [1..]
```

9. A matrix can be represented as a list of equal-length sublists. Define the following functions over such a data structure:

(a) function `allGreaterThan` that checks whether all elements of a matrix are greater than a given value;

```
allGreaterThan 2 [[1..3], [4..6]] ⇒ False
allGreaterThan 2 [[3..5], [4..6]] ⇒ True
allGreaterThan 3 [[1,2], [7]] ⇒ error "malformed matrix"
allGreaterThan 0 [] ⇒ error "malformed matrix"
```

(b) function `getElem` that returns the element at a given position;

```
getElem (0,2) [[1..3],[4..6]] ⇒ 3
getElem (7,-2) [] ⇒ error "indices out of bounds"
```

(c) function `sumElems` that returns the sum of all elements of a matrix;

```
sumElems [[1..3],[4..6]] ⇒ 21
sumElems [[1..4]] ⇒ 10
sumElems [[5],[]] ⇒ error "malformed matrix"
sumElems [] ⇒ error "malformed matrix"
```

(d) function `trace` that returns the sum of the diagonal elements of a matrix.

```
trace [[1..3],[4..6],[7..9]] ⇒ 15
trace [[1..3],[4],[7,7]] ⇒ error "malformed matrix"
trace [[1..7]] ⇒ error "asymmetrical matrix"
```

10. Define the following functions that operate on perfect two-level binary trees. (Hint: use `fst` and `snd`, or come up with something cooler.)

(a) `sumTree` that returns the sum of the tree's values.

```
sumTree((1,7),(5,2)) ⇒ 15
```

(b) function `outermost` that returns a pair consisting of the leftmost and rightmost elements of the tree;

```
outermost ((1,2),(3,4)) ⇒ (1,4)
```

(c) function `leaves` that returns a list of the leaf elements of the tree;

```
leaves ((1,2),(6,5)) ⇒ [1,2,6,5]
```

(d) function `maxLeaf` that returns the tree's maximum element.

```
maxLeaf ((1,6),(3,19)) ⇒ 19
```

11. A directed graph can be represented as a list of edges, while each edge can be represented with a tuple (a, b), where edge directs from a to b. Define `detectCycles` that returns a list of all cycles of size 1 or 2. Return the cycle as a tuple (a, b), where $a \leq b$.

```
detectCycles [(1,2),(2,3),(3,4)] ⇒ []
detectCycles [(1,2),(2,3),(3,4),(2,1)] ⇒ [(1,2)]
detectCycles [(1,1),(2,2),(3,4),(4,3)] ⇒ [(1,1),(2,2),(3,4)]
detectCycles [(3,2),(2,3)] ⇒ [(2,3)]
```