# Exam Functional Programming – November 3rd 2014

| | |
|---|---|
| Name | |
| Student number | |
| I study CS/AI/Other | |

- Write **neatly** and carefully. Use a pen (no pencil!) with black or blue ink.
- Write your answers in the answer boxes. If you need more space, use the back side of the sheet and make a reference to it.
- You can score 90 points. You get 10 points for free, yielding a maximum of 100 points in total. Your exam grade is the obtained points divided by 10.
- If you need auxiliary lemmas in a proof, then prove the validity of these lemmas as well.

You may use throughout the entire exam the following functions + lemmas:

```
[]     ++ ys             =  ys
(x:xs) ++ ys             =  x : (xs++ys)

map f  []                =  []
map f (x:xs)             =  f x : map f xs

foldr f z []             = z
foldr f z (x:xs)         = f x (foldr f z xs)

sum []                   = 0
sum (x:xs)               = x + sum xs

reverse []               = []
reverse (x:xs)           = reverse xs ++ [x]

head (x:xs)              = x
tail (x:xs)              = xs

length []                = 0
length (x:xs)            = 1 + length xs

f . g                    = \x -> f (g x)

zip (x:xs) (y:ys)        = (x,y) : zip xs ys
zip  xs     ys           = []

-- Lemma associativity of ++ (may be used without proof):
-- (xs ++ ys) ++ zs = xs ++ (ys ++ zs) = xs ++ ys ++ zs

-- Lemma concatenation with []  (may be used without proof):
-- xs ++ [] = xs
```

1. (5× 2=10 points)

**(a)** What is the type of the standard Haskell function `zip`?

```
zip (x:xs) (y:ys)        = (x,y) : zip xs ys
zip  xs     ys           = []
```

**(b)** What is the type of the standard Haskell function `concat`?

```
concat = foldr (++) []
```

**(c)** What is the type of the following Haskell function `uncurry`?

```
uncurry f = (\(a,b) -> f a b)
```

**(d)** What is the type of the following Haskell function `plus1`?

```
plus1 = map (+ 1)
```

**(e)** What is the type of the following Haskell function `f`?

```
f = sum.h.g
g = (\x -> (head x, (head.reverse) x))
h (x,y) = [x,y]
```

2. (10 points) A Dutch Citizen Service Number (DCSN) has always 9 digits and the first digit can be a 0. Many websites use the following rudimentary check to validate the correctness of the (9 digit) number $ABCDEFGHI$. First compute $X = 9 \times A + 8 \times B + 7 \times C + 6 \times D + 5 \times E + 4 \times F + 3 \times G + 2 \times H - 1 \times I$. Note that the last digit has a negative weight. If $X$ is a multiple of 11, then the number $ABCDEFGHI$ passes the test, otherwise it is invalid.

   Write a Haskell functie `isDCSN` (including its type) that determines whether its argument passes the test described above.

3. (3+3+4=10 points)

- Write a function `relPrimePairs n` that returns the list of pairs `(i,j)` where `1<i<j<=n` and `i` and `j` have no common factor (you may use the function `gcd` that computes the greatest common divisor of its two arguments). The implementation of `relPrimePairs` must be a list comprehension.

- Given are the Haskell definitions of `suits`, `cards` and `honours`:

```
suits = ["Clubs", "Diamonds", "Hearts", "Spades"]
cards = map show [2..10]
honours = ["J","Q","K","A"]
```

  Write a list comprehension for `deck`, where `deck` is

```
[("Clubs","2"),("Clubs","3"),("Clubs","4"),("Clubs","5"),("Clubs","6"),("Clubs","7"),
("Clubs","8"),("Clubs","9"),("Clubs","10"),("Clubs","J"),("Clubs","Q"),("Clubs","K"),
("Clubs","A"),("Diamonds","2"),("Diamonds","3"),("Diamonds","4"),("Diamonds","5"),
("Diamonds","6"),("Diamonds","7"),("Diamonds","8"),("Diamonds","9"),("Diamonds","10"),
("Diamonds","J"),("Diamonds","Q"),("Diamonds","K"),("Diamonds","A"),("Hearts","2"),
("Hearts","3"),("Hearts","4"),("Hearts","5"),("Hearts","6"),("Hearts","7"),("Hearts","8"),
("Hearts","9"),("Hearts","10"),("Hearts","J"),("Hearts","Q"),("Hearts","K"),("Hearts","A"),
("Spades","2"),("Spades","3"),("Spades","4"),("Spades","5"),("Spades","6"),("Spades","7"),
("Spades","8"),("Spades","9"),("Spades","10"),("Spades","J"),("Spades","Q"),("Spades","K"),
("Spades","A")]
```

- Use a list comprehension and the function `zip` to write a Haskell function `locations n xs` that returns the list of all indexes `i` such that the `i`th element of `xs` is `n` (i.e. `xs!!i == n`). Note that the first elelement of a list has index 0. You are not allowed to use the indexing operator `!!`.
  Example: `locations 0 [x ‘mod‘ 10 | x <- [1..50]]` should yield `[9,19,29,39,49]`.

4. (3+3+4=10 points)

- The function `iterate` creates an infinite list where the first item is calculated by applying the function its first argument on its secod argument, the second item by applying the function on the previous result and so on. For example, `iterate (2*) 1` yields the infinite list `[2,4,8,16,32,64,128,256,512,...]`. Give a Haskell implementation (including its type) of the function `iterate`.

- Define the infinite list `ints`, which is the list of all integers. It should be ordered in such a way that you can find any given integer after searching a finite number of elements in `ints`. In other words, this is not going to work: `ints = [0..] ++ [-1, -2..]`

- Given is the definition of the infinite list of primes:

```
primes  = sieve [2..]
  where sieve (p:xs)  = p:sieve [x | x <- xs, x ‘mod‘ p /= 0]
```

  Use `primes` to define the infinite list `composites` of non-primes. So, `take 10 composites` should yield `[4,6,8,9,10,12,14,15,16,18]`. Note that we skip the value 1.

5. (10 points) We are used to write expressions using *infix* notation. For instance, we write `10 - (4 + 3) * 2`. The downside of this notation is that we have to use parentheses to denote precedence. *Reverse Polish Notation* (RPN) is another way of writing down expressions, and does not need parentheses. In RPN, every operator follows its operands, therefore RPN is also called *postfix notation*. The above expression in RPN is: `10 4 3 + 2 * -`
Evaluating such an expression goes as follows. We keep pushing numbers onto a stack, until we encounter the first operator. So, when we encounter the +, the stack contains [3, 4, 10] (here, the head of the list is the top of the stack). We replace the two top numbers from the stack by their sum. The stack is now [7, 10]. Next, we push 2 on the stack (so, [2, 7, 10]). Now, we encounter an operator again, we pop 2 and 7 off the stack, apply the operator and push the result to the stack yielding [14, 10]. Finally, there is a −. We pop 10 and 14 from the stack, subtract 14 from 10 and push that back. The number on the stack is now -4, which is the final result.

   We use the following data type for representing RPN literals:

```
data RPN = Value Integer | Plus | Minus | Times |Div
```

   Write a Haskell funtion `rpn :: [RPN] -> Integer` that evaluates a RPN expression to an `Integer`.
   Two examples:

```
rpn [Value 10,Value 2, Div]                               ↦  5
rpn [Value 10,Value 4, Value 3, Plus, Value 2, Times, Minus]  ↦  -4
```

6. (15 points) The abstract data type (ADT) `Fifo tp` implements a simple data type for the storage of elements of the type `tp`, from which elements are retrieved in the same order as in which they are inserted: FIFO stands for *First In First Out queue*.

   Implement a `module Fifo` such that the concrete implementation of the type Fifo is hidden from the user.

   The following operations on the data type `Fifo` must be implemented:

   - `empty` returns an empty queue.
   - `isEmpty` returns `True` for an empty queue, otherwise `False`.
   - `insert`: returns the queue that is the result of inserting an element.
   - `top`: returns the 'oldest' element of the queue.
   - `remove`: returns the queue that is obtained by removing the 'oldest' element.

7. (10 points) Given is the data type `Tree`:

```
data Tree a = Leaf a | Node a (Tree a) (Tree a)
```

Given are the functions `leaves` and `nodes`:

```
  leaves (Leaf _)     = 1
  leaves (Node a l r) = leaves l + leaves r
  nodes  (Leaf _)     = 0
  nodes  (Node a l r) = 1 + nodes l + nodes r
```

Prove for all finite trees `t`:     `leaves t = nodes t + 1`

8. (15 points) Given are the definitions of the functions `rev1`, `shunt`, and `rev2`:

```
rev1 :: [a] -> [a]
rev1 [] = []
rev1 (x:xs) = (rev1 xs) ++ [x]

shunt :: [a] -> [a] -> [a]
shunt [] ys = ys
shunt (x:xs) ys = shunt xs (x:ys)

rev2 :: [a] -> [a]
rev2 xs = shunt xs []
```

Prove that `rev1 xs = rev2 xs` for all finite lists `xs`.