Lund University
Department of Computer Science

EDAN40: Functional Programming
13th January 2015, 14–19

Exam

1. Rewrite the following definitions into a point-free form (i.e., `f = ..`, `g = ..`), using neither lambda-expressions nor list comprehensions nor enumeration nor `where` clause nor `let` clause:

   - `f x y = (3 + x) / y`
   - `g x y = [ y z | z <- [1..x]]`

2. Give the types for the following operator expressions

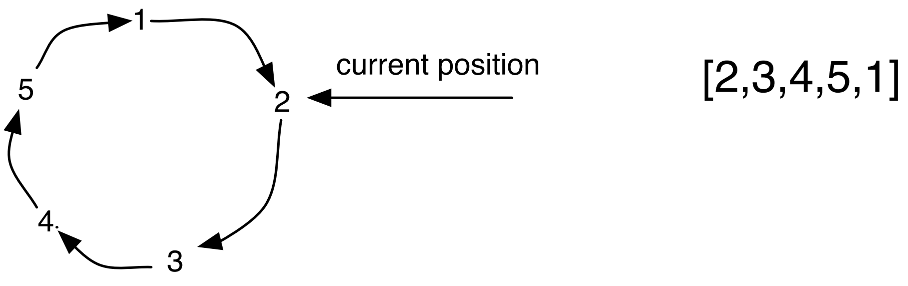
   - The Haskell smiley: `(8-)`
   - Haskell goggles: `(+0).(0+)`
   - Haskell wheels: `(.)(.)`
   - A Haskell treasure: `(($)$($))`
   - Haskell swearing: `([]>>=)(\_->[(>=)])`

3. Explain the concept of a *spark* in Haskell. How does it relate to the following three functions

   `seq, pseq, par ::  a -> b -> b`

   Explain what they do.

4. Define a type `CircList` (or `CL` for short, if you prefer) defining a circular list of arbitrary length (and, say, holding integer elements). You have to make sure that the *current position* is well-defined and accessible for the operations defined for this type. The picture below illustrates the concept and its possible representation using a standard list (with the assumption that the first element defines the current position and that the last position in the list is glued to the first one in a circular fashion):



   Please note that it is your task to define the appropriate type constructor! Define then for this type the following functions:

```
perimeter    :: CircList -> Int
currentelem  :: CircList -> Maybe Int
nextelem     :: CircList -> Maybe Int
previouselem :: CircList -> Maybe Int
insert       :: CircList Int -> CircList
takefromCL   :: Int CircList -> [Int]
```

returning the number of elements in the list, returning the current element in the list, returning the next element in the list, returning the previous element in the list, inserting an element between the current and the previous element in the list, and taking n first elements of the circular list, respectively. You may, and are actually encouraged to, define any helper functions you deem appropriate.

Finally, define a predicate

```
equalcirclist :: CircList CircList -> Bool
```

yielding `True` if and only if both list contain the same elements in the same order, but not necessarily with the same current position. E.g., if we used the standard list representation for circular lists (assuming the end is glued to the beginning) then

```
equalcirclist (CircList [1, 2, 3, 4, 5]) (CircList [3, 4, 5, 1, 2]) = True
equalcirclist (CircList [1, 2, 3, 4, 5]) (CircList [3, 4, 5, 2, 1]) = False
equalcirclist (CircList [1, 2, 3, 4, 5]) (CircList [3, 4, 5, 1, 2, 3]) = False
```

5. What is the type of the following function:

```
g x y = do
          a <- x
          b <- y
          return (a,b)
```

   What is the value of:

```
g (Just 5) Nothing
g [1, 2, 3] [5, 6, 7]
g (Just "I am") (Just "Charlie")
```

6. Define the standard function

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

   using `foldr`, so that your definition looks as follows:

```
filter p = foldr ...
```

Good Luck!