EXAM FUNCTIONAL PROGRAMMING
Tuesday the 29th of September 2015, 11.00 h. - 13.00 h.

Name: Student number:

Before you begin: Do not forget to write down your name and student number above. If necessary, explain your answers (in English or Dutch). For multiple choice questions, clearly circle what you think is the (one and only) best answer. Use the empty boxes under the other questions to write your answer and explanations in. Use the blank paper provided with this exam only as scratch paper (kladpapier). At the end of the exam, only hand in the filled-in exam paper. Answers will not only be judged for correctness, but also for clarity and conciseness. A total of one hundred points can be obtained; divide by 10 to obtain your grade. Good luck!

In any of your answers below you may (but do not have to) use the following well-known Haskell functions/operators: *replicate*, *id*, *concat*, *foldr* (and variants), *map*, *filter*, *const*, *all*, *any*, *flip*, *fst*, *snd*, *not*, (.), *elem*, *take*, *drop*, *takeWhile*, *dropWhile*, *head*, *tail*, (++), *lookup*, *max*, *min* and all members of the type classes *Eq*, *Num*, *Ord*, *Show* and *Read*.

1. (i) Write a function *allNats* :: [*Int*] -> *Bool* that returns *True* if and only if all values in the input list are larger than zero. You are *not* allowed to use explicit recursion.

. . . /6

allNats xs = *all* (>0) *xs*

but also

allNats xs = *minimum xs* > 0

If foldr is used instead, you get

allNats′ xs = *foldr* (\ *x y* -> (*x* > 0) && *y*) *xs*

(ii) Using explicit recursion (no higher-order functions!) implement the function *pairUp* that pairs subsequent elements of a list: *pairUp* [1, 2, 3, 4] returns [(1, 2), (3, 4)]. If the list has an odd number of elements, the final element of the list should be *ignored*. So *pairUp* [1, 2, 3, 4, 5] is also [(1, 2), (3, 4)].

. . . /8

```
pairUp :: [a] -> [(a, a)]
pairUp [] = []
pairUp [x] = []
pairUp (x1 : x2 : xs) = (x1, x2) : pairUp xs
```

(iii) The function *rldecode* performs run length decoding. For the input $[1, 2, 3, 4]$ we expect it to generate the list [2,4,4,4]: the first number tells us how many of the second end up in the output, the third number says how many of the fourth we want, and so on. If the input list contains negative numbers or zero, then the empty list should be returned.

We implement this function using the functions *allNats* and *pairUp* as follows:

```
rldecode :: [Int] -> [Int]
rldecode xs =
  if allNats xs then
    foldr f e arg
  else
    []
  where f ...
        e ...
        arg ...
```

Given suitable definitions for f, e and *arg*.

. . . /**8**

```
f (x, y) xs = replicate x y ++ xs
  -- or (++) . (\(x, y) -> replicate x y) for example
e = []
arg = pairUp xs
```

A completely different solution that I had not thought of, but does get all points is something like:

```
f = (++)
e = []
arg = map g (pairUp xs)
  where g (x, y) = replicate x y
```

(iv) Give the type of the function *flip* that when given an arbitrary binary function f, returns the function that takes the arguments for f in reverse order.

. . . /**5**

```
flip :: (a -> b -> c) -> (b -> a -> c)
```

or

```
flip :: (a -> b -> c) -> b -> a -> c
```

2. (i) Define a datatype *CountTree a* with the following properties:

   a. the leaves of the tree do not contain any information
   b. in the branches, the tree splits in two
   c. in every branch there is a piece of information of (the same) type a, and
   d. in every branch there is an additional value of type integer

.../7

```
data CountTree a = Leaf
    | Branch (CountTree a) a Int (CountTree a)
```

Quite a few people had something like

```
Leaf | (a, Int) :> (CountTree a, CountTree a)
```

which is also allowed, but :> must have exactly two arguments (hence the added tuples). [*CountTree a*] is a bit too roomy instead of (*CountTree a*, *CountTree a*).

(ii) Write a function

```
height :: CountTree a -> Int
```

that computes how high the tree is. The height of a *Leaf* is zero, the height of a non-leaf is one more than the maximum height of the subtrees.

.../7

```
height :: CountTree a -> Int
height Leaf = 0
height (Branch ct1 _ _ ct2) = 1 + max (height ct1) (height ct2)
```

(iii) Write a function

```
insert :: Float -> CountTree Float -> CountTree Float
```

that adds an element to a count tree, even if it is already present.

The function may assume the tree has the following properties, but must also ensure that the tree that it returns has those same properties: (A) the values in the branches are in order, meaning that the value in a node is *larger then or equal to* the values in the left subtree, and *smaller* than all the values in the right subtree. (B) the additional integer values indicate how many values are contained in the LEFT subtree of each node.

.../8

```
insert :: Float -> CountTree Float -> CountTree Float
insert flt Leaf = Branch Leaf flt 0 Leaf
insert flt (Branch ct1 val cnt ct2) =
  if flt <= val then Branch (insert flt ct1) val (cnt + 1) ct2
  else Branch ct1 val cnt (insert flt ct2)
```

(iv) Write a function *insertU* that behaves almost the same as *insert* but that only insert the float argument if it does not yet occur in the tree. In your implementation, you may visit every branch at most once. Hint: implement a helper function *insertHelp* :: .... -> (*CountTree Float*, *Bool*) that returns not only the modified tree, but some additional information as well.

[. . . /6] One possible solution is:

```
insertU :: Float -> CountTree Float -> CountTree Float
insertU flt ct = fst (insertHelper ct)
  where
    insertHelper :: CountTree Float -> (CountTree Float, Bool)
    insertHelper Leaf = (Branch Leaf flt 0 Leaf, True)
    insertHelper t @@ (Branch ct1 val cnt ct2) =
      if flt == val then (t, False)   -- do not insert. value's there
      else
        if flt < val then    -- go left
          let (ctl, grown) = insertHelper ct1
          in (if grown then Branch ctl val (cnt + 1) ct2
              else Branch ctl val cnt ct2, grown)
        else    -- go right
          let (ctr, grown) = insertHelper ct2
          in  (Branch ct1 val cnt ctr, grown)
```

3. (i) A very efficient definition of *segs* for all segments of a list is

$$segs\ xs = [\,] : [\,t \mid i \leftarrow inits\ xs, t \leftarrow tails\ i, not\ (null\ t)\,]$$

What is *segs* $[1,2,3,4]$?

[. . . /4] $[[\,],[1],[1,2],[2],[1,2,3],[2,3],[3],[1,2,3,4],[2,3,4],[3,4],[4]]$ (order does not matter)

(ii) Explain in your own words how this function computes what it is supposed to compute. If necessary illustrate by a small example.

[. . . /6] The code conceptually consists of a nested loop: first the inits are computed that represent all ways of cutting a piece off at the end, then tails is used to do the same on the other side. This leads to many duplicate nulls (for t), so we remove those, and add one [] afterwards.

4. [. . . /**20**] **c, b, c, c**

The following multiple choice questions are each worth 5 points.

(i) Let f be any function of type $Int \rightarrow Bool$. Which expression has the same value as the following list comprehension?

$$[f\ x \mid x \leftarrow [2\,..\,8], odd\ x]$$

a. $filter\ odd\ (map\ f\ [2\,..\,8])$
b. $f\ (map\ odd\ [2\,..\,8])$
c. $map\ f\ (filter\ odd\ [2\,..\,8])$
d. $filter\ f\ (map\ odd\ [2\,..\,8])$

(ii) I $[id, const\ 2]$ is well-typed
II $\backslash x \rightarrow [even\ x, fib\ x]$, where *fib* is the well-known Fibonacci function, is well-typed

a. Both I and II are true
b. Only I is true
c. Only II is true
d. Both I and II are false

(iii) Given that $until :: (a \rightarrow Bool) \rightarrow (a \rightarrow a) \rightarrow a \rightarrow a$ what is the type of *foldr until*?

a. $(b \rightarrow Bool) \rightarrow b \rightarrow [b \rightarrow b] \rightarrow b$
b. $[a \rightarrow a] \rightarrow [a \rightarrow Bool] \rightarrow [a]$
c. $(a \rightarrow a) \rightarrow [a \rightarrow Bool] \rightarrow a \rightarrow a$
d. The expression is type incorrect.

(iv) I In an expression with only associative operators you can omit all parentheses.
II That the operator $(++)$ is defined as right-associative helps increase computational efficiency.

a. Both I and II are true
b. Only I is true
c. Only II is true
d. Both I and II are false

5. Determine the type of *until even*. You should not just write down the type below, but also explain how you arrived at that type (for example, in the way that this is done in the lecture notes of this course).

. . . /**15** See p. 121, sec. 5.5 in the reader.