EXAM FUNCTIONAL PROGRAMMING
Thursday the 5th of November 2015, 17.00 h. - 20.00 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: *zipWith*, *zip*, *id*, *concat*, *foldr* (and variants), *map*, *filter*, *const*, *all*, *any*, *flip*, *fst*, *snd*, *not*, (.), *elem*, *take*, *drop*, *takeWhile*, *dropWhile*, *head*, *tail*, *repeat*, *replicate*, (++), *lookup*, *max*, *min* and all members of the type classes *Eq*, *Num*, *Ord*, *Show* and *Read*.

1. (i) To commemorate the 200th birthday of George Boole, define a type class, call it *BoolA* with a single argument *a*, to represent the concept of a boolean algebra. It should support functions that represent binary conjunction (*andb*), binary disjunction (*orb*) and unary negation (*notb*), as well as two constants *bot* and *top*. Here *top* represents the neutral element of *andb*, and *bot* that of *orb*.

   .../**6**

```
class BoolA a where
  andb :: a -> a -> a
  orb :: a -> a -> a
  notb :: a -> a
  bot :: a
  top :: a
```

   (ii) Give an instance definition for *BoolA Bool*.

   .../**6**

```
instance BoolA Bool where
  andb = (&&)
  orb = (||)
  notb = not
  bot = False
  top = True
```

(iii) This exercise was disqualified from the exam due to its incorrect phrasing. The consequence is that everyone gets 4 points for this question, and on top of that, those that did have something sensible can get additional points.

(iv) Give an instance definition for values of type $[a]$ (for any a that is an instance of *BoolA*) that extends the operations to lists elementwise. For example, $andb\ [True, False]\ [True, True, False] = [andb\ True\ True, andb\ False\ True] = [True, False]$. Note that the length of the result is the same as the minimum of the length of the two arguments. Again make sure that *top* and *bot* are in fact neutral for the right operator.

$\boxed{\dots/\mathbf{6}}$

instance *BoolA a* $\Rightarrow$ *BoolA* $[a]$ **where**
  andb = *zipWith andb*
  orb = *zipWith orb*
  notb = *map notb*
  bot = *repeat bot*
  top = *repeat top*

(v) Why is it not a good idea to have *andb* and *orb* return a list that is as long as the longest (this could then be done by correctly padding the shorter list)? In which situation would this not be a problem?

$\boxed{\dots/\mathbf{4}}$ *top* and *bot* are defined as infinite lists (or actually, lengths of arbitrary length). This is really useful and can be done because of lazy evaluation. But if we take the max, then we can't use infinite lists, and in particular, we cannot use *top* and *bot*.

2. (i) Implement the function $minimum :: (Ord\ a) \Rightarrow [a] \rightarrow a$ for lists that computes the smallest element of a list. It should display a nice run-time error message when you pass the empty list. Also, you must implement it with a fold (choose one of *foldr*, *foldr1*, *foldl*, and *foldl1*).

$\boxed{\ldots/\mathbf{6}}$

```
minimum [] = error "minimum was passed an empty list"
minimum xs = foldl1 min xs
  -- or
minimum xs = foldl1 (\x y -> if x < y then x else y) xs
```

(ii) Which of the other three could you have used as well? Which do you think are most suitable to use, and why?

$\boxed{\ldots/\mathbf{3}}$ Any would do, but the foldl1 and foldr1 can be considered more elegant, since they take the default from the list which you know will be there. Some people remarked that the foldr/foldr1 version is faster, which is also an acceptable answer.

(iii) Reflect on why it would or would not be a good idea to use a strict fold function, such as *foldl'*.

$\boxed{\ldots/\mathbf{3}}$ Yes, it is a good idea. If you try it out you will in fact see it is much faster.

(iv) A fellow student has defined a function $sort :: (Ord\ a) \Rightarrow [a] \rightarrow [a]$ for sorting a list. Define a QuickCheck property that verifies that the first element of a sorted list is the smallest in the list. Make sure the QuickCheck property never crashes: only non-empty lists may contribute to the test set.

$\boxed{\ldots/\mathbf{6}}$

```
prop_minimum' xs = not (null xs) ==> head (sort xs) == minimum xs
```

3. $\boxed{\ldots/\mathbf{20}}$ **b,a,d,a**

The following multiple choice questions are each worth 5 points.

(i) What is the type of *flip foldr*, where $\mathit{flip} :: (a \rightarrow b \rightarrow c) \rightarrow b \rightarrow a \rightarrow c$ switches the arguments a and b of a function.

a. $(a \rightarrow c) \rightarrow [a \rightarrow (a \rightarrow c) \rightarrow c] \rightarrow a \rightarrow c$
b. $b \rightarrow (a \rightarrow b \rightarrow b) \rightarrow [a] \rightarrow b$
c. It is type incorrect, since *foldr* takes three arguments.
d. It is type incorrect, but for another reason then the one listed under (c).

(ii) Which expressions are equivalent, i.e., can replace each other in any context?

a. $\mathit{takeWhile}\ p\ .\ \mathit{dropWhile}\ p$ and $\mathit{dropWhile}\ p\ .\ \mathit{takeWhile}\ p$
b. $\mathit{takeWhile}\ p\ .\ \mathit{dropWhile}\ q$ and $\mathit{takeWhile}\ (\backslash x \rightarrow q\ x\ \&\&\ p\ x)$
c. $\mathit{dropWhile}\ p\ .\ \mathit{takeWhile}\ q$ and $\mathit{takeWhile}\ q\ .\ \mathit{dropWhile}\ p$
d. $\mathit{dropWhile}\ p\ .\ \mathit{dropWhile}\ q$ and $\mathit{takeWhile}\ (\backslash x \rightarrow p\ x\ \&\&\ q\ x)$

(iii) I A general purpose language cannot be embedded in another general purpose language.

II A major advantage of shallow over deep EDSLs is that you can optimize EDSL programs before running them.

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

(iv) Assume $\perp$ is an expression that always crashes, and $f :: \mathit{Int} \rightarrow \mathit{Int} \rightarrow \mathit{Int}$ is a function that always crashes. Which of the following statements is false:

a. $\mathit{seq}\ (f\ 1)\ 3$ crashes.
b. $(\backslash x \rightarrow ())\ \perp$ equals $()$.
c. $\mathit{head}\ (\mathit{map}\ \perp\ [1,2])$ crashes.
d. For basic types, *seq* and *deepseq* have the same behaviour.

4. Given the following Haskell definition where $g :: \mathit{Int} \rightarrow \mathit{Maybe\ Int}$ and $h :: a \rightarrow \mathit{Maybe}\ a$

```
f w = do
  x <- g w
  let xs = do
    z <- [1,2]
    v <- ['a','b']
    return (z, v)
  y <- h (snd (head xs))
  return y
```

Complete the following explanation by filling in the gaps:

In the *Maybe* monad, *Nothing* signals failure and Just a successful computation. In the above program, the type of x is *Int*, the type of y is *Char*, the type of xs is $[(\mathit{Int}, \mathit{Char})]$, and the type of f is $\mathit{Int} \rightarrow \mathit{Maybe\ Char}$. If we call f and would print the value of xs to the screen then we'd see $[(1,\texttt{'a'}),(1,\texttt{'b'}),(2,\texttt{'a'}),(2,\texttt{'b'})]$. $\boxed{\ldots/\mathbf{10}}$

5. The following definitions make *Maybe* into a monad:

$$
\begin{array}{ll}
(1)\ f \gg\!= g & = \textbf{case } f \textbf{ of} \\
 & \quad \textit{Nothing} \;\text{->}\; \textit{Nothing} \\
 & \quad \textit{Just } x \;\;\text{->}\; g\ x \\
(2)\ \textit{return}\ x = \textit{Just}\ x &
\end{array}
$$

The following is often called the third monad law:

$$(m \gg\!= f) \gg\!= g = m \gg\!= (\backslash\, x \text{ -> } f\ x \gg\!= g)$$

(i) Prove that the law holds if m equals *Nothing*. Hint: to simplify the proof you may first want to prove Lemma A that says that $\textit{Nothing} \gg\!= f = \textit{Nothing}$.

$\boxed{\ldots/\mathbf{5}}$ Note: this is 11.9 from the exercises.

It is easiest to first prove Lemma A that says that $\textit{Nothing} \gg\!= f = \textit{Nothing}$:

$$
\begin{array}{ll}
 & \textit{Nothing} \gg\!= f \\
=== & (1) \\
 & \textbf{case } \textit{Nothing} \textbf{ of} \\
 & \quad \textit{Nothing} \text{ -> } \textit{Nothing} \\
 & \quad \textit{Just}\ x \text{ -> } f\ x \\
=== & (\textbf{case}\ ) \\
 & \textit{Nothing}
\end{array}
$$

Then the proof goes like this:

$$
\begin{array}{ll}
 & (\textit{Nothing} \gg\!= f) \gg\!= g \\
=== & (\textit{LemmaA}) \\
 & \textit{Nothing} \gg\!= g \\
=== & (\textit{LemmaA}) \\
 & \textit{Nothing} \\
=== & (\textit{LemmaA}) \\
 & \textit{Nothing} \gg\!= (\backslash\, x \text{ -> } f\ x \gg\!= g)
\end{array}
$$

(ii) Prove that the law holds for $m = \textit{Just}\ x$. Again, it may be wise to first prove a Lemma B that $\textit{Just}\ x \gg\!= f = f\ x$.

$\boxed{\ldots/\mathbf{6}}$

$$
\begin{array}{ll}
 & \textit{Just}\ x \gg\!= f \\
=== & (1) \\
 & \textbf{case } \textit{Just}\ x \textbf{ of} \\
 & \quad \textit{Nothing} \text{ -> } \textit{Nothing} \\
 & \quad \textit{Just}\ x' \text{ -> } g\ x' \\
=== & (\textbf{case}\ , \textit{and } x' \textit{ equals } x) \\
 & g\ x \\
 & (\textit{Just}\ x \gg\!= f) \gg\!= g \\
=== & (\textit{Lemma B}) \\
 & f\ x \gg\!= g \\
=== & (\textit{expansion}) \\
 & (\backslash\, x' \text{ -> } (f\ x' \gg\!= g))\ x \\
=== & (\textbf{case}\ ) \\
 & \textbf{case } \textit{Just}\ x \textbf{ of} \\
 & \quad \textit{Nothing} \text{ -> } \textit{Nothing} \\
 & \quad \textit{Just}\ x'' \text{ -> } (\backslash\, x' \text{ -> } (f\ x' \gg\!= g))\ x'' \\
=== & (1) \\
 & \textit{Just}\ x \gg\!= (\backslash\, x' \text{ -> } (f\ x' \gg\!= g))
\end{array}
$$

Some people saw that you can in fact replace the last two steps with Lemma B again. Of course, that is correct too.

6. Induction

(1) $[] \quad ++ \; ys = ys$
(2) $(x : xs) ++ ys = x : (xs ++ ys)$,

(3) $foldr\ op\ e\ [] = e$
(4) $foldr\ op\ e\ (x : xs) = op\ x\ (foldr\ op\ e\ xs)$

(i) Given that op is an associative binary operator, and e a neutral element of op, prove by induction that

$$foldr\ op\ e\ (xs ++ ys) = op\ (foldr\ op\ e\ xs)\ (foldr\ op\ e\ ys)\ .$$

$\boxed{\ldots/\mathbf{15}}$ See 13.2: 1 for each equation (8 in all), 3 for the right choice of induction variable, 2 for starting each case correctly.