EXAM FUNCTIONAL PROGRAMMING
Tuesday the 30th of September 2014, 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; if you run out of space, you can use the empty sixth page of the exam. Use the empty 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: *id*, *concat*, *foldr* (and variants), *map*, *filter*, *const*, *flip*, *fst*, *snd*, *not*, (.), *elem*, *take*, *drop*, *takeWhile*, *dropWhile*, *head*, *tail*, (++), *lookup* and all members of the type classes *Eq*, *Num*, *Ord*, *Show* and *Read*.

1. (i) Write a function $intersperse :: a \to [a] \to [a]$, which places its first argument between the elements of its second argument; i.e. *intersperse* `'a'` `"xyz"` should return `"xayaz"`. You *must* use direct recursion.

...**/8** There are quite a few possible solutions. The most straightforward one is e.g.:

$$\begin{aligned} &intersperse\ a\ (x : y : ys) = x : a : intersperse\ a\ (y : ys) \\ &intersperse\ \_\ xs \qquad\qquad\ = xs \end{aligned}$$

The second case takes care of the empty *xs* and an *xs* of length 1.

(ii) Give an alternative definition of *intersperse* without direct recursion, using higher-order functions.

...**/8** Alternative solutions are:

$$\begin{aligned} &intersperse\ a\ xs = tail\ \ (\ foldr\ (\lambda x\ r \to a : x : r)\ \ [\,]\ xs) \\ &intersperse\ a \quad\ = tail\ \ .\ foldr\ (\lambda x\ r \to a : x : r)\ \ [\,] \\ &intersperse\ a \quad\ = tail\ \ .\ foldr\ (\lambda x \to (a{:})\ .\ (x{:}))\ [\,] \\ &intersperse\ a \quad\ = tail\ \ .\ foldr\ ((a{:})\ .\ )\ .\ (:)) \qquad [\,] \\ &intersperse\ a \quad\ = tail\ \ .\ concat\ .\ map\ (\lambda x \to [a, x]) \end{aligned}$$

2. The operator (.) composes two functions. We want to generalise this and implement a function that composes a list of functions *compoR2L* of type $[(a \to a)] \to a \to a$. For example, the call *compoR2L* $[f, g, h]$ v computes the value $f\ (g\ (h\ v))$.

   (i) Implement *compoR2L* using *foldr*.

   $\boxed{\ldots/\mathbf{8}}$

   $compoR2L :: [a \to a] \to (a \to a)$
   $compoR2L = foldr\ (\ .\ )\ id$

   (ii) Implement *compoL2R*, that composes functions in the opposite direction. In other words, *compoL2R* $[f, g, h]$ v equals $h\ (g\ (f\ v))$.

   $\boxed{\ldots/\mathbf{8}}$

   $compoL2R :: [a \to a] \to (a \to a)$
   $compoL2R = foldl\ (flip\ (\ .\ ))\ id$

   Using (i) and some kind of list-reversal is also okay. Or should I say use foldl?

   (iii) What do you get when you evaluate *compoL2R* $[not, even]$ 3?

   $\boxed{\ldots/\mathbf{5}}$ A type error since *not* and *even* do not have the same type.

3. Given is the following definition of a so-called *Trie a*

   data *Trie a* = *Leaf a* | *Branch a* [(*Char*, *Trie a*)]

   The idea of a *Trie* is that *every branch and leaf* contains a payload value of type *a*, and that the children of a branch are indexed by a value of type *Char*. An example of a *Trie Int* is the following:

```
ex = Branch 40 [('a', Branch 20 [('a', Leaf 1),
                                 ('b', Leaf 2)]),
                ('b', Branch 30 [('a', Leaf 3),
                                 ('c', Leaf 4)])
     ]
```

   (i) Write a function *sumIntTrie* :: *Trie Int* $\to$ *Int* that adds all the payloads (of type *Int*) together. For example, *sumIntTrie* (*Leaf* 3), should return 3, and *sumIntTrie ex* should return 100.

$\ldots/\mathbf{10}$

$$
\begin{aligned}
&\mathit{sumIntTrie} :: \mathit{Trie}\ \mathit{Int} \to \mathit{Int}\\
&\mathit{sumIntTrie}\ (\mathit{Leaf}\ v) = v\\
&\mathit{sumIntTrie}\ (\mathit{Branch}\ i\ \mathit{children}) =\\
&\quad i + \mathit{sum}\ (\mathit{map}\ (\mathit{sumIntTrie}\ .\ \mathit{snd})\ \mathit{children})
\end{aligned}
$$

(ii) Write a function $\mathit{searchTrie} :: [\mathit{Char}] \to \mathit{Trie}\ a \to \mathit{Maybe}\ a$, that follows a path down the tree as indicated by the first argument, and just returns the payload of the *branch or leaf* it reaches in this way, and *Nothing* otherwise. For example, $\mathit{searchTrie}\ []\ \mathit{ex}$ gives $\mathit{Just}\ 40$, $\mathit{searchTrie}\ [\texttt{'b'}]\ \mathit{ex}$ gives $\mathit{Just}\ 30$, $\mathit{searchTrie}\ [\texttt{'b'}, \texttt{'a'}]\ \mathit{ex}$ returns $\mathit{Just}\ 3$, and $\mathit{searchTrie}\ [\texttt{'b'}, \texttt{'a'}, \texttt{'h'}]\ \mathit{ex}$ returns *Nothing*. Here you may use a function $\mathit{clookup} :: \mathit{Char} \to [(\mathit{Char}, b)] \to \mathit{Maybe}\ b$ such that $\mathit{clookup}\ c\ \mathit{ps}$ returns v if (c, v) is the first pair in ps in which c is the first component, and *Nothing* if no such pair exists.

$\ldots/\mathbf{10}$

$$
\begin{aligned}
&\mathit{searchTrie} :: [\mathit{Char}] \to \mathit{Trie}\ a \to \mathit{Maybe}\ a\\
&\mathit{searchTrie}\ []\ (\mathit{Leaf}\ v) = \mathit{Just}\ v\\
&\mathit{searchTrie}\ []\ (\mathit{Branch}\ v\ \_) = \mathit{Just}\ v\\
&\mathit{searchTrie}\ \_\ \ (\mathit{Leaf}\ \_) = \mathit{Nothing}\\
&\mathit{searchTrie}\ (x : \mathit{xs})\ (\mathit{Branch}\ \_\ \mathit{chds}) =\\
&\quad \mathbf{case}\ \mathit{clookup}\ x\ \mathit{chds}\ \mathbf{of}\\
&\qquad \mathit{Nothing}\ \ \to \mathit{Nothing}\\
&\qquad (\mathit{Just}\ \mathit{child}) \to \mathit{searchTrie}\ \mathit{xs}\ \mathit{child}
\end{aligned}
$$

(iii) A problem with the definition of *Trie* is that the type system does not forbid values like $\mathit{Branch}\ 45\ [(\texttt{'a'}, \mathit{Leaf}\ 33), (\texttt{'a'}, \mathit{Leaf}\ 78)]$. Give a definition for $\mathit{Trie}\ a$ that does not have that problem.

$\ldots/\mathbf{5}$ Use a function: $\mathbf{data}\ \mathit{Trie}\ a = \mathit{Leaf}\ a \mid \mathit{Branch}\ a\ (\mathit{Maybe}\ (\mathit{Char} \to \mathit{Trie}\ a))$ but $\mathbf{data}\ \mathit{Trie}\ a = \mathit{Leaf}\ a \mid \mathit{Branch}\ a\ (\mathit{Char} \to \mathit{Trie}\ a)$ is also okay.

4. $\boxed{\dots/\mathbf{20}}$ **(a), (a), (c), (d)**
The following multiple choice questions are each worth 5 points.

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

$$[f\ x \mid x \leftarrow [1\,..\,6], even\ x]$$

a. $map\ f\ (filter\ even\ [1\,..\,6])$
b. $filter\ even\ (map\ f\ [1\,..\,6])$
c. $f\ (map\ even\ [1\,..\,6])$
d. $filter\ f\ (map\ even\ [1\,..\,6])$

(ii) Given is the following unnecessarily complicated function definition:

$$\begin{aligned}
&f\ g = f\ 0\ g\\
&\quad \mathbf{where}\ f\ g\ h \mid g < length\ h = foldr\ (const\ (+1))\ 0\ (h\ !!\ g) + f\ (g+1)\ h\\
&\qquad\qquad\qquad\ \mid otherwise \quad = 0\\
&\qquad\quad const\ f\ g = f
\end{aligned}$$

Which of the following implementations is equivalent?

a. $sum\ .\ map\ length$
b. $foldr\ (+)\ 0\ .\ map\ (const\ 1)$
c. $foldr\ ((+)\ length)\ 0$
d. $foldl1\ (+)\ .\ map\ length$

(iii) I Both function application and the $\to$ in function types associate to the left so that Currying becomes possible.

II Function application has precedence over all operators.

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) What is the type of $map\ .\ foldr$?

a. $(a \to a \to a) \to [a] \to [[a] \to a]$
b. $(a \to a \to a) \to [b] \to [b \to a]$
c. $(b \to a \to a) \to [b] \to [[a] \to a]$
d. $(b \to a \to a) \to [a] \to [[b] \to a]$

5. (i) Explain why the expression $\lambda x \to (x\ [\texttt{'1'}], x\ \texttt{'1'})$ is type incorrect.

$\boxed{\dots/\mathbf{5}}$ For $x\ [\texttt{'1'}]$ and $x\ \texttt{'1'}$ to both be type correct, we should derive a polymorphic type for x. But x is lambda-bound, not let-bound, and in that case x can not have a polymorphic type.

(ii) Determine the type of *map filter*. 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).

$\boxed{\dots/\mathbf{15}}$ See p. 123, sec. 5.10 in the reader. In case of *map filter* we have

$$\begin{aligned} &\mathit{map} :: (a \to b) \to [\,a\,] \to [\,b\,] \\ &\mathit{filter} :: (c \to \mathit{Bool}) \to [\,c\,] \to [\,c\,], \end{aligned}$$

where we have chosen fresh type variables in the type of *filter*. The function *map* gets a single argument:

$$\mathit{map} :: \underbrace{(a \to b)}_{\text{argumenttype}} \to \underbrace{[\,a\,] \to [\,b\,]}_{\text{resulttype}}.$$

matching the types of the arguments,

$$\underbrace{a \to b}_{\text{argumenttype of } \mathit{map}} \quad \equiv \quad \underbrace{(c \to \mathit{Bool}) \to [\,c\,] \to [\,c\,]}_{\text{type of } \mathit{filter}}$$

$$\Rightarrow \left\{ \begin{array}{lcl} a & \equiv & c \to \mathit{Bool} \\ b & \equiv & [\,c\,] \to [\,c\,], \end{array} \right.$$

gives:

$$\mathit{map} :: \underbrace{(\cancelto{(c \to \mathit{Bool})}{a} \to \cancelto{[c] \to [c]}{b})}_{\text{argumenttype}} \to \underbrace{[\cancelto{(c \to \mathit{Bool})}{a}] \to [\cancelto{[c] \to [c]}{b}]}_{\text{resulttype}}.$$

and thus

$$\mathit{map} :: \underbrace{((c \to \mathit{Bool}) \to [\,c\,] \to [\,c\,])}_{\text{type of } \mathit{filter}} \to \underbrace{[\,c \to \mathit{Bool}\,] \to [[\,c\,] \to [\,c\,]]}_{\text{type of } \mathit{map\ filter}}$$

Finally we get for *map filter*

$$\mathit{map\ filter} :: [\,c \to \mathit{Bool}\,] \to [[\,c\,] \to [\,c\,]],$$

or:

$$\mathit{map\ filter} :: [\,a \to \mathit{Bool}\,] \to [[\,a\,] \to [\,a\,]].$$