EXAM FUNCTIONAL PROGRAMMING (WITH ANSWERS)
Tuesday the 1st of October 2013, 8.30 h. - 10.30 h.

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, circle what you think is the best answer. Use the empty boxes under each question to write your answer and explanations in. 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!

1. (i) What is the result of evaluating the following expression:

   $(foldr\ (-)\ 0\ .\ map\ length)\ [\texttt{"Rood"}, \texttt{"Wit"}, \texttt{"Blauw"}]$

.../**8** *map length* results in $[4,3,5]$ earning you 4 points. Passing it to *foldr* results in $4-(3-(5-0))$ which equals 6. If you show the right expression and the right value you get 4 more points. If you miscalculate the expression, but you have the right expression you get 3 points. If you only give a number and it is wrong, then you get 0 points. If you give the right answer but no explanation at all how you got it, you get 4 points.

   (ii) The function *nrOfPositives* is defined as follows:

   $nrOfPositives = length\ .\ filter\ (>0)$

   Define this function with *foldr* by supplying the correct definitions of e and op, according to the following recipe:

   $nrOfPositives = foldr\ op\ e$
   where .....

.../**15** The implementation computes the number of positive integers in a list. For e there is only one correct answer: $e = 0$. This gives you 5 points. op can be implemented in very many ways. A rather explicit one is $op = \lambda x\ r \rightarrow$ **if** $x > 0$ **then** $r + 1$ **else** r This earns you the other 10 points. If you, for example, interchange the role of x and r, then you get 7. Another good implementation of the above op is with sections: $op\ x \mid x > 0 = (1+)$ $\quad \mid otherwise = id$ Many students wrote $(+0)$ instead of id. That is okay, too. This implementation has a lot of room for small mistakes, since the section syntax is easy to get wrong.

2. Consider the following datatype for representing boolean expression (propositions), where variable names consist of a single character:

```
data Prop = Cons Bool
          | Vari Char
          | Not Prop
          | Prop :/\: Prop
          | Prop :\/: Prop
          | Prop :->: Prop
```

   (i) Give the value of type *Prop* that represents the proposition $p \rightarrow (p \vee q)$

…/**6** The fully correct answer is $(Vari\ \texttt{'p'}) :->: ((Vari\ \texttt{'p'}) :\backslash/: (Vari\ \texttt{'q'}))$
For the answer *Cons True* (which has the same logical meaning, but is not a representation of the formula), you still get 3 points. Answers like *True* or *Bool* are completely wrong.
There are many ways to get the answer partly right. Basically, I counted the number of correct kinds of tokens. Often made mistakes: not parenthesizing the or part, writing types in the values, like $Vari\ Char :->:$ (note that although all *Vari* expressions really need to have them as well, I did not subtract points for that).

(ii) Write a function $vars :: Prop \to [Char]$ that takes a proposition and finds all the variables that occur in the proposition (you may choose to delete duplicates, or not).

…/**14** A correct answer is

$$
\begin{aligned}
&vars :: Prop \to [Char] \\
&vars\ (Cons\ \_) = [\,] \\
&vars\ (Vari\ c) = [c] \\
&vars\ (Not\ p) = vars\ p \\
&vars\ (p :/\backslash: q) = vars\ p +\!\!+ vars\ q \\
&vars\ (p :\backslash/: q) = vars\ p +\!\!+ vars\ q \\
&vars\ (p :->: q) = vars\ p +\!\!+ vars\ q
\end{aligned}
$$

Forgetting the parentheses around some of the patterns costs 1 pt (not one each!). Typically you get points for each case that you had right (and that includes not only the right hand side, but also the left hand side).
Some people overabstracted writing

$$vars\ (p\ \_\ q) = vars\ p +\!\!+ vars\ q$$

which they thought would cover all binary operators. Alas, Haskell does not allow this. The penalty is not high in this case, because it still shows that you do understood the recursive process.
Instead of patterns, you can also use a **case** expression. However, quite a few used guards resulting in things like:

$$
\begin{aligned}
&vars\ p \mid v \equiv Vari = v \\
&\qquad\quad \mid v \equiv (p :/\backslash: q) = vars\ p +\!\!+ vars\ q
\end{aligned}
$$

This is wrong. If you need to introduce new identifiers for parts of the matched pattern *never* use guards. Guards are just boolean expressions. Of course, in this case you could still get points for the correct right hand side (in the example v the right hand side is wrong, but the case of $:/\backslash:$ is right).
A number of students used an accumulating parameter. That is fine, too. But then $vars\ p +\!\!+ vars\ q$ should become something like $vars\ q\ (vars\ p\ acc)$ and not $vars\ p\ acc +\!\!+ vars\ q\ acc$. The latter duplicates variables much more than necessary, and although I did not ask to delete duplicates, the latter implementation indicates that you do not understand the principle of the accumulating parameter all that well.
Typical for this exercise is that students forget the *Not* and/or the *Cons* altogether. Another frequently made mistake was that students wrote something like

$$vars\ (Prop\ p :/\backslash: Prop\ q) = vars\ p +\!\!+ vars\ q$$

Remember that you typically do not see types show up inside Haskell code.
There were two students who actually deleted duplicates, and one did it incorrectly: as a rule try not to do more than you must, unless it makes your life easier.
Generally, throughout this entire exercise and the next students tend to mix case: variables start with capitals, and type variables as well. Try not to do this, as this suggests that you do not know what you are doing. There is a major difference between *prop* and *Prop*. This is one aspect where it pays off to be clear and precise in your writing.

(iii) On the next page, write an evaluator $beval :: Prop \to [(Char, Bool)] \to Maybe\ Bool$. It takes a proposition and an environment *env*. The latter contains pairs of values, like (`'p'`, *True*),

that give values to variables. In order to compute the truth value of the proposition, you must first verify that *env* has a truth value for every variable in that proposition (Hint: use (ii)). If this is not the case, you should return *Nothing*. Otherwise you should return *Just v* where *v* is the truth value computed for the proposition.

. . . /**14** Again there are many possibilities here. This is one.

```
beval :: Prop → [(Char, Bool)] → Maybe Bool
beval p env =
  if alldefined (vars p) env then Just (ceval p) else Nothing
  where
    alldefined [] env = True
    alldefined (v : vs) env = case lookup v env of
      Nothing → False
      Just _ → alldefined vs env
    ceval (Cons b) = b
    ceval (Vari c) = fromJust (lookup c env)
    ceval (Not p) = not (ceval p)
    ceval (p :/\: q) = ceval p ∧ ceval q
    ceval (p :\/: q) = ceval p ∨ ceval q
    ceval (p :->: q) = ceval (Not p) ∨ ceval q
    }
```

It is okay to reuse functions like *elem*, and operators like (\\). It is also okay to verify the *alldefined* property during the evaluation traversal. However, some students then forget that a *Maybe* value is returned by the recursive calls, and that these need to be unpacked. Some other often made mistakes: instead of verifying that each element of *vars p* is in a first element of a pair of *env*, they do this the other way around. Many students forgot the operators for *or* and *and*. A way out is then to use *and* [*ceval p*, *ceval q*]. This is okay (albeit not very elegant). If you can't remember what they are called, and it is easy to write it down yourself, and do just that (in fact, a small number of students did it like that).
As with the previous exercise, the pattern matching is sometimes done in the guards.
There are many many ways in which to implement the definedness test (if done correctly it gives you 4 points, the evaluation gives you 10). It was probably also the hardest thing to get right without the ability to test the implementation, since there are many ways in which you can subtly get it wrong.

---

3. (i) Assuming that the prelude does not provide you with a definition for function composition (.), write your own. Also provide a type signature for (.) .

. . . /**9** I was surprised to see so many wrong answers here. The type of

$$(\ .\ ) \mid is \mid (b \to c) \to (a \to b) \to a \to c$$

A correct type gives you 4 points, an incorrect one zero, unless I believed that you made a honest typo.
The implementation varies, but not so much:

```
( . ) f g x = f (g x)
```

is the simplest, but the following are also fine:

```
(f . g) x = f (g x)   -- many forget the parentheses on the left
( . ) f g = λx → f (g x)
```

and variants with local definitions.
A correct implementation gives you 5 points. You get none if it is incorrect, unless, again, it looks like a simple typo.

(ii) Explain what it means that you can regard (.) as a function that takes 1, 2 or 3 parameters.

…/**8** One aspect here that you should mention is that in Haskell every function takes only one argument, and currying can be used to model functions of multiple arguments. This gets you half the points.
If you then go on and explain concretely that the types of the arguments of (.) above have type $b \to c$, $a \to b$ and a, and/or that the type of (.) can for example be understood as $(b \to c) \to (a \to b) \to (a \to c)$ which is suggestive of the case that you pass only the first two arguments to return a function of type $a \to c$, then you earn another 4 points.

(iii) Given the type signature $sum :: [Int] \to Int$, is the expression

if *True* **then** $[sum, length]$ **else** $[id]$

type correct? If so, what is its type. If not, explain why not.

…/**10** This was not an easy one for many. The expression is not type correct. The reason is that the type of the then and else parts must correspond, but the type of the then part is $[Int] \to Int$, while that of the else part is $a \to a$. Clearly, if we match the arguments, a must be $[Int]$, but then the result types can never match. Only if you also provide the right reason, do you get all points.
Even if you give the wrong answer (or the right one, but not the right reason), you still get five points if your answer shows that you can perform type inference, for example by saying that while *length* has type $[a] \to Int$, the type of the then part is $[Int] \to Int$ because that is what *sum* demands.
Quite a few students came up with a type like $Bool \to Int$ or some such, because of the boolean condition, and the fact that the functions can return an Int. But note that there is no boolean argument anywhere, and that the function returns a list of functions. There is no reason for these functions to be applied here; Haskell is a higher-order language.
Some students remarked that the else part is never reached, and then thought that lazy evaluation would make it okay since the expression can only return a value of type $[Int] \to Int$. However, Haskell is strongly typed and the type inferencer will not perform a control flow analysis to decide what is reachable and what is not. So whether a piece of code is reachable or not, it must all be type correct. Note that laziness has nothing to do with this issue, the same thing could be said of any strict language.

4. (i) Determine the type of *map map*. 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).

…/**16** The answer to this one can be found in the lecture notes. It is also allowed to provide your own explanation of the correct type, which is $[a \to b] \to [[a] \to [b]]$ or any consistent renaming thereof.
For providing only the type of *map* you get 2 points, for also providing another copy of that type (with other type variables) you get an additional 3. Many students provided not the type of *map map*, but the type of the first *map*. That cost them 4 points. If you only provide the exact type, but no explanations, I subtracted 5. If you made a wrong match (but substituted the results of the match correctly), you lose 4 points.