EXAM FUNCTIONAL PROGRAMMING

Tuesday the 1st of October 2016, 08.30 h. - 10.30 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 100 points can be obtained. Good luck!

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

1. LISTS

(i) Define a function *vouw* :: [*a*] -> [(*a*, *a*)] that takes a list *xs*, and pairs the first with the last, the second with the one-but-last, and so on. If the list has odd length, the middle element is paired with a copy of itself. So *vouw* [1, 2, 3, 4, 5] = [(1, 5), (2, 4), (3, 3)] and *vouw* [1, 2, 3, 4, 5, 6] = [(1, 6), (2, 5), (3, 4)]. Implement this function without using recursion yourself, and without list comprehensions.

. . . /8

```
vouw :: [a] -> [(a, a)]
vouw xs = zip l1 l2
  where
    n = length xs
    l1 = take ((n + 1) `div` 2) xs
    l2 = take ((n + 1) `div` 2) (reverse xs)
```

(ii) Define a function *ulh* that takes a list *xs*, and pairs each element with all other elements in *xs*. *ulh* [1, 2, 3, 2] = [(1, 2), (1, 3), (1, 2), (2, 1), (2, 3), (2, 2), (3, 1), (3, 2), (3, 2), (2, 1), (2, 2), (2, 3)], in that order. Do this using recursion and without list comprehensions. Moreover, of the above listed functions you may only use (++). You may define your own local definitions.

. . . /9

```
ulh :: [a] -> [(a, a)]
ulh xs = help [] xs
  where
    help _ [] =                   []
    help prev (next : rest) = combine next (prev ++ rest) ++
                              help (prev ++ [next]) rest
    combine x []       = []
    combine x (y : ys) = (x, y) : combine x ys
```

(iii) Would (ii) be easy to do with a list comprehension? If so, explain how; if not, explain what makes it hard.

$\boxed{\ldots/\mathbf{4}}$ This seems quite hard to do. It is easy of course to one-by-one take every element out of a list, but finding all elements that precede it, finding all elements that follow it is not straightforward at all. Since the list compr. as it were iterates through the list for you, it is cumbersome if you want to compute something during the iteration.

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

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

Here, the constructor :/\: represents conjunction $\wedge$, and the constructor *Not* represents the negation symbol $\neg$.

(i) Give the value of type *Prop* that represents the proposition $(\neg v \wedge w) \wedge$ `tt` where v and w represent variables, and `tt` represents the value true.

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

```
(Not (Vari 'v') :/\: Vari 'w') :/\: Cons True
```

(ii) Write an evaluator *eval* :: (*Char* –> *Bool*) –> *Prop* –> *Bool* that takes a function that maps variables to booleans, and a proposition, and returns the boolean value of that proposition.

$\boxed{\ldots/\mathbf{10}}$

```
eval :: (Char -> Bool) -> Prop -> Bool
eval env p = heval p
  where
    heval (Cons b) = b
    heval (Vari v) = env v
    heval (Not p) = not (heval p)
    heval (p :/\: q) = heval p && heval q
```

(iii) For all propositions p, $\neg(\neg p) = p$. Write a simplifier *simpl* :: *Prop* –> *Prop* that uses this (and only this) equality as much as it can to simplify boolean propositions. For example, *simpl prop* = *Cons True* :/\: *Not* (*Vari* `'v'`) where *prop* = *Not* (*Not* (*Not* (*Not* (*Cons True*)) :/\: *Not* (*Vari* `'v'`))).

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

```
simpl :: Prop -> Prop
simpl (Cons b) = Cons b
simpl (Vari v) = Vari v
simpl (Not (Not p)) = simpl p
simpl (Not p) = Not (simpl p)
simpl (p :/\: q) = simpl p :/\: simpl q
```

(iv) We also have some equalities for conjunction ($\wedge$): $p \ \wedge$ `ff` $=$ `ff` $=$ `ff` $\wedge \ p$ and $p \ \wedge$ `tt` $= p = true \ \wedge \ p$. Now, *simpl prop* = *Not* (*Vari* `'v'`) for the *prop* given in (iii). Extend *simpl* to apply also these optimisations.

$\boxed{\ldots/\mathbf{6}}$ Replace the case for :/\: by

```
simpl' (p :/\: q) = let ps = simpl' p
        qs = simpl' q
  in
    case (ps, qs) of
      (Cons False, _) -> Cons False
      (_, Cons False) -> Cons False
      (Cons True, _) -> qs
      (_, Cons True) -> ps
      otherwise -> ps :/\: qs
```

3. SUBLISTS
In this question we deal with a function *subs* :: [*a*] –> [[*a*]] which returns all the sublists of the argument list (i.e., all the lists that result by deleting elements from the argument list in any possible way).

(i) How many sublists does [1,2,3,4] have?

$\boxed{\ldots/\mathbf{4}}$ 16 = 2^4. Deleting elements in every possible way includes deleting nothing at all.

(ii) Explain how you can compute *subs* (*x*:*xs*) from *subs xs* (for example by using concrete values for *x* and *xs*).

$\boxed{\ldots/\mathbf{4}}$ If you take all the lists that are sublists of *xs*, then all you need to do is to take these, and a copy of each of them but with *x* cons'ed at the front and put these together.

(iii) Now, write the function *subs* :: [*a*] –> [[*a*]] exploiting sharing where you can.

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

```
subs []       = [[]]   -- 1 pt
subs (x : xs) = map (x:) subs_xs ++ subs_xs
   where subs_xs = subs xs
```

No sharing: -1.

4. MULTIPLE CHOICE $\boxed{\ldots/\mathbf{20}}$

d, a, c, c

The following multiple choice questions are each worth 5 points.

(i) I For a left-associative operator $\oplus$, the expression $a \oplus b \oplus c$ should be interpreted as $a \oplus (b \oplus c)$.

II When we say `infix :<>:` in Haskell, we mean that the operator :<>: is associative, so that we can write expressions like a :<>: b :<>: c.

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

(ii) I [*const* `'2'`, (*flip const*) `"2"`], where *flip* flips the first two arguments of a function, is well-typed.

II \ x –> [(*flip* . *flip*) x, *id*] 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) What is the type of *map* (`'}'`:) . *map reverse* . *map* (`'{'`:)?

a. [[a]] –> [[a]]
b. [[*String*]] –> [[*String*]]
c. [[*Char*]] –> [[*Char*]]
d. The expression is type incorrect.

(iv) The function *intersperse* :: a –> [a] –> [a] puts its first argument between all the elements of a non-empty list. Thus *intersperse* `'a'` `"xyz"` results in `"xayaz"`. Which definition is correct, assuming the argument *as* is not empty?

a. *intersperse a as* = *foldr* (\ e r –> (e : a : r)) [] *as*
b. *intersperse a as* = *foldl* (\ r e –> (a : e : r)) [] *as*
c. *intersperse a as* = (*tail* . *concat* . *map* (\ x –> [a, x])) *as*
d. *intersperse a as* = *tail* [(a : e) | e <– *as*]

5. TYPE INFERENCE
   Determine the type of *foldr concatMap*, where $concatMap :: (a \mathrel{-\!\!>} [b]) \mathrel{-\!\!>} [a] \mathrel{-\!\!>} [b]$. 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).

$\boxed{\ldots/\mathbf{15}}$ The type of *foldr* is $(a \mathrel{-\!\!>} b \mathrel{-\!\!>} b) \mathrel{-\!\!>} b \mathrel{-\!\!>} [a] \mathrel{-\!\!>} b$ (2 pts). The type of *concatMap* is given, but we now need to make sure our type variables are disjoint, so we can choose $(c \mathrel{-\!\!>} [d]) \mathrel{-\!\!>} [c] \mathrel{-\!\!>} [d]$.
For *foldr concatMap*: we get $a \mathrel{-\!\!>} b \mathrel{-\!\!>} b == (c \mathrel{-\!\!>} [d]) \mathrel{-\!\!>} [c] \mathrel{-\!\!>} [d]$. So $b = [d]$, $b = [c]$, so $d = c$. Then $a = c \mathrel{-\!\!>} [d]$ becomes $a = c \mathrel{-\!\!>} [c]$ and we had already $b = [c]$. Substiting this into $b \mathrel{-\!\!>} [a] \mathrel{-\!\!>} b$ gives $[c] \mathrel{-\!\!>} [c \mathrel{-\!\!>} [c]] \mathrel{-\!\!>} [c]$, or if you want $[a] \mathrel{-\!\!>} [a \mathrel{-\!\!>} [a]] \mathrel{-\!\!>} [a]$.