[20201001] INFOFP - Functioneel programmeren - 1 - UITHOF

Cursus: BETA-INFOFP Functioneel programmeren (INFOFP)

Tijdsduur:	2 uur
Aantal vragen:	6

[20201001] INFOFP - Functioneel programmeren - 1 - UITHOF

Cursus: Functioneel programmeren (INFOFP)

Aantal vragen: 6

1 In this question, we study the following puzzle: given a list of integer numbers, find a correct way of inserting arithmetic operators and parentheses such that the result is a correct equation. Example: With the list of numbers **[2,3,5,7,11]** we can form the equations (((2 - 3) + 5) + 7) = 11 or 2 = (((3 * 5) + 7) / 11) (and ten others!).
Division should be interpreted as operating on rationals, and division by zero should be avoided.

a. a. We define a type of binary operators:

data Op = Plus | Minus | Multiply | Divide deriving Enum

Using **Op**, define a parameterized data type **Expr a** of **a**-expressions, which are either constants of type **a** (labelled **Const**) -- or a binary operator applied to two existing **a**-expressions (labelled **Binary**).

b. b. Define a **Show** instance for **Expr a** that correctly prints an equation, using parentheses where appropriate. You do not need to worry about redundant parentheses and associativity of operators. You may assume that **Op** has already been made an instance of **Show**, where **Plus**, **Minus**, **Multiply**, and **Divide** are, respectively, shown as **"+"**, **"-"**, **"*"**, and **"/"**.

c. c. Write a function **splits** that generates all possible splittings of a list of length at least 2 into a pair of *non-empty* lists, while retaining the order of elements.
splits has the specification that, for **zs** of length greater than or equal to 2, **(x:xs, y:ys) = splits zs** if and only if **(x:xs) ++ (y:ys) = zs**.

Hint:

Define an appropriate helper function **splits'** such that **splits = tail . splits'** .

d. We define a type synonym

type Value = Rational

Next, we define a function apply that tries to evaluate a binary operator on two values and fails in case of division by zero:

```
apply :: Op -> Value -> Value -> Maybe Value
apply Plus x y      = Just (x + y)
apply Minus x y     = Just (x - y)
apply Multiply x y = Just (x * y)
apply Divide x 0    = Nothing
apply Divide x y    = Just (x / y)
```

Given a list of numbers, we wish to generate all expressions we can build from this list by inserting operators and parentheses, paired with the value they evaluate to. We exclude ill-formed expressions which contain a division by zero.

```
exprs :: [Integer] -> [(Expr Integer, Value)]
exprs [n] = [(Const n, fromInteger n)]
exprs ns = [(Binary op e1 e2, v) | (ns1, ns2) <- splits ns,
                                    (e1, v1) <- d. ........................ () ,
                                    (e2, v2) <- e. ........................ () ,
                                          op <- [Plus, Minus, Multiply, Divide],
```

f. .... () **<-**

g. .... () **]**

Please complete the gaps in the code above.

h. e. Without using a list comprehension, write a function **equalsFive :: [Integer] -> [Expr Integer]** that generates all expressions using integers from some list that evaluate to 5.

2 In this question, we will ask you to write, in several different ways, a function to remove multiples of 5 and 7 from a list.

a. a. Write a predicate **multipleFS** on **Int**, such that **multipleFS n** equals **True** iff **n** is a multiple of 5 or 7 (or both), put differently, iff **n** is divisible by 5 or divisible by 7 (or divisible by both 5 and 7).

b. b. Write a function named **EM1** that takes a list of **Int**'s as its only parameter. Your function should return a new list of **Int**'s where all of the **Int**'s that are multiples of 3 or 5 have been removed. Implement your solution using recursion. You may not use list-- comprehensions or higher order functions in your solution.

Test cases that you may want to consider include:

EM1 [] should return **[]**
EM1 [11, 13, 17, 19] should return **[11, 13, 17, 19]**
EM1 [3, 5, 6] should return **[]**
EM1 [1..20] should return **[1,2,4,7,8,11,13,14,16,17,19]**

c. c. Write a function named **EM2** that performs the same task as **EM1** using list comprehensions. You may not use recursion or higher order functions in your solution.

d. d. Write a function named **EM3** that performs the same task as **EM1** and **EM2** by calling one or more higher order functions built into Haskell’s prelude module. You may not use recursion or list comprehensions in your solution.

3 In this question, we will implement a so-called "multimap", that is, an associative array data structure which can store for each key not one value, but any number of multiple values. Further, the same value can be associated multiple times with a given key and the values associated with a key can be associated in many different orders which we distinguish. For example, when we lookup a key, we retrieve the most recently inserted value (if it exists), and when we delete a key, we only remove the most recently inverted value.
You may make use of the **Map k v** data type which is a type of associate arrays which associate keys of type **k** with values of type **v**.

Recall that such **Map k v** types can be accessed using the following API:
insertMap :: Ord k => k -> v -> Map k v -> Map k v
deleteMap :: Ord k => k -> Map k a -> Map k a
lookupMap :: Ord k => k -> Map k a -> Maybe a
emptyMap :: Map k a

Complete the holes in the following implementation.

```
type MultiMap k v = Map.Map k [v]

insert :: Ord k => k -> v -> MultiMap k v -> MultiMap k v
insert k v m = case lookupMap k m of Just vs ->
a. ........................................................................................................................ ()
                                                Nothing -> insertMap k [v] m

delete :: Ord k => k -> MultiMap k a -> MultiMap k a
delete k m = case lookupMap k m of  b.  ......................... ()  -> insertMap k vs m
                                                                _ -> deleteMap k m

lookup :: Ord k => k -> MultiMap k a -> Maybe a
lookup k m = case lookupMap k m of Just (v : vs) ->  c.  ......................... ()
                                                        _ -> Nothing

empty :: MultiMap k a
empty = emptyMap
```

4 Please answer the questions below. You will receive 1pt for each question correctly answered, -1pt for each wrong answer, and 0pt for each question answered with "Don't know".

a. a. For each of the following expressions, please indicate whether it is correct that they evaluate to the list **[1,2,3,4,5]**.

		Correct	Incorrect	Don't know
		A	B	C
[f | f <- [1..10], g <- [1..10], f <= 5]	1	○	○	○
[d \`div\` 2 | d <- [1..10], (d + 1) \`div\` 2 == d \`div\` 2]	2	○	○	○
[c + 1 | c <- [1..10], c < 4]	3	○	○	○
map (+1) [b \`div\` 2 | b <- [1..10], b \`mod\` 2 == 1]	4	○	○	○
filter (\x -> 5 > x) [1..21]	5	○	○	○
map (2+) (filter (>= -1) [-5 .. 3])	6	○	○	○
[a | a <- [1..10], a < 5]	7	○	○	○

b. b. Please mark all well-typed definitions.

		Well-typed	Ill-typed	Don't know
		A	B	C
moo f x = let y = f x in if x == y then y else moo f y	1	○	○	○
friet = let f g = (g [], g 0) in f (\x->x + 1)	2	○	○	○
foo = (\y -> y) (\y -> y)	3	○	○	○
baz g = g g	4	○	○	○
bar f = f (bar f)	5	○	○	○

c. c. The function **interspерse :: a −> [a] −> [a]** puts its first argument between all the elements of a non-empty list. Thus **intersperse ’,’ "xyz"** results in **"x,y,z"**. Which definitions are correct, assuming the argument as is not empty?

		Correct	Incorrect	Don't know
		A	B	C
intersperse a as = foldr (\ e r −> (e : a : r)) [ ] as	1	○	○	○
intersperse a as = foldl (\ r e −> (a : e : r)) [ ] as	2	○	○	○
intersperse a = foldr (\x ys -> x : if null ys then [ ] else a : ys) [ ]	3	○	○	○
intersperse a = tail . concat . map (\ x −> [a, x])	4	○	○	○
intersperse _ [a'] = [a'] **intersperse a (a' : as) = a' : a : intersperse a as**	5	○	○	○
intersperse a as = tail [(a : e) | e <− as]	6	○	○	○

5 We consider a type of natural numbers

```
data Nat = Zero | Succ Nat
```

We think of **Zero** as the number 0 and **Succ n** as the number n + 1. On this type, we can define a function

```
foldN :: (a -> a) -> a -> Nat -> a
foldN f e Zero = e
foldN f e (Succ n) = f (foldN f e n)
```

analogous to **foldr** on **[a]**.

Define the type

```
data Dummy = D
```

Observe that **Nat** is essentially the type **[Dummy]** "in disguise", in the sense that we have functions

```
listToNat :: [()] -> Nat
natToList :: Nat -> [()]
```

such that

listToNat . natToList equals **id** and **natToList . listToNat** equals **id** .

a. a. Using direct recursion, implement a function

plus :: Nat -> Nat -> Nat

that adds two natural numbers.

b. b. Using **foldN** and without using direct recursion, write a function

mult :: Nat -> Nat -> Nat

that multiplies two natural numbers.

c. c. Using a fold and without using direct recursion, implement

listToNat :: [Dummy] -> Nat

d. d. Please implement

natToList :: Nat -> [Dummy]

For 2 points: implement it using direct recursion

OR

For 3 points: implement it using **foldN** and without using direct recursion.

e. Consider the following definition

```
mystery i = snd (foldN (\(x, _) -> (Succ x, x)) (Zero, Zero) i)
```

The type of **mystery** is **e.** .... () .
Let us use the shorthand notation 0 for **Zero**, 1 for **Succ Zero**, 2 for **Succ (Succ Zero)**, etc. .
If we evaluate **mystery** on 7, we obtain the answer **f.** .... () and if we evaluate it on 0, it returns **g.** .... () .
Please write your answers using numerals.

6 Determine the type of the following expressions or demonstrate that they are not well-typed.
Hint: **const x _ = x**

a. a. **foldr const id**

b. b. **flip foldr True (&&)**