[20251106] INFOFP - Functioneel programmeren - 1 - USP

Cursus: BETA-INFOFP Functioneel programmeren (INFOFP)

Tijdsduur: 2 uur en 30 minuten

Aantal vragen: 6

[20251106] INFOFP - Functioneel programmeren - 1 - USP

Cursus: Functioneel programmeren (INFOFP)

Aantal vragen: 6

1 In this question, we test your understanding of equational reasoning about functional programs.

11 pt.

[11pt] Using the following definitions

```
data Tree = Leaf | Node Tree Tree

mirror :: Tree -> Tree
mirror Leaf = Leaf                                  -- a
mirror (Node l r) = Node (mirror r) (mirror l)      -- b

(.)       :: (b -> c) -> (a -> b) -> (a -> c)
(f . g) x = f (g x)                                   -- c
```

please prove the following claim, where you justify every reasoning step by marking it with the letter (a - c) of some definition or by marking it as I.H. to refer to an induction hypothesis. Please clearly state any induction hypotheses you use. Please do not combine multiple reasoning steps but justify each step separately as I.H. or (a - c).

Claim:

```
(mirror . mirror) t = t
```

2 In this question, we test your understanding of programming on novel algebraic data types.

Consider the following simplified representation of html.

```
data Tag = Div        -- container
         | P          -- paragraph
         | H Int      -- H1,..,H6 tags
         | Span       -- inline
         | Img String -- image with the source URL
         | Br         -- newline
         deriving (Show)

type AttributeName = String

data HtmlP v = TextNode String
             | Elem Tag (Map AttributeName v) [HtmlP v]
             deriving (Show)

type Html = HtmlP String
```

where

```
Map k v
```

is a type of associative array. In particular, recall that there are the functions

```
lookup :: Ord k => k -> Map k v -> Maybe v
```

to (try to) find an element with a given key in the Map, and

```
fromList :: Ord k => [(k,v)] -> Map k v
```

to construct a Map k v from a list of key,value pairs.
We can then represent a snippet of a Html document as:

```
myHtml :: Html
myHtml = Elem Div (fromList [])
                  [ Elem (Img "header.jpg") (fromList []) []
                  , Elem (H 1) (fromList [("id","myTitle")])
                               [TextNode "My FP Webpage"]
                  , TextNode "FP is "
                  , Elem Span (fromList [("class","emphasize")])
                              [TextNode "Wonderful!"]
                  , Elem Div (fromList [("id","bar")])
                             [ Elem (Img "lambda.jpg") (fromList
[("id","lambda")]) []
                             , Elem Div (fromList [("id","empty")]) []
                             ]
                  ]
```

(this would correspond to a html page with a header image, a title, a single line of text, and and a div (a container) with another image (and a nested empty div)).
a. [6pt] As you may know, an html element may have an attribute with AttributeName "id". Write a function

```
collectIds :: Html -> [String]
```

that collects all **Divs** that have AttributeName "id" in an html document. For example, we have:

```
> collectIds myHtml
["bar","empty"]
```

6 pt. **a.**

b. [4pt] Write a function

```
replaceImages :: Html -> Html
```

that replaces every image by the text "image: <url to the image>". For example, on the above snippet we get:

```
> replaceImages myHtml
Elem Div (fromList [])
         [ TextNode "image: header.jpg"
         , Elem (H 1) (fromList [("id","myTitle")])
                      [TextNode "My FP Webpage"]
         , TextNode "FP is "
         , Elem Span (fromList [("class","emphasize")])
                     [TextNode "Wonderful!"]
         , Elem Div (fromList [("id","bar")])
                    [ TextNode "image: lambda.jpg"
                    , Elem Div (fromList [("id","empty")]) []
                    ]
         ]
```

4 pt. **b.**

c. [5pt] The 'Span' tag is supposed to be used as an "inline" container (e.g. so that we can mark up a word in a sentence), rather than contain arbitrarily nested documents. Write a function

```
verifySpans :: Html -> Bool
```

that checks that every 'Span' tag contains only 'TextNode _'s as its children (and thus no Elem _ _ _'s). So in particular, we have:

```
> verifySpans myHtml
True
```

but on the following malformed Html document

```
malformedHtml = Elem Div (fromList [])
```

```
                            [ Elem Span (fromList [("id","bar")])
                                        [ Elem (Img "lambda.jpg") (fromList
[("id","lambda")]) []
                                        , Elem Div (fromList [("id","empty")])
[]
                                        ]
                            ]
```

we would get

```
> verifySpans malformedHtml
False
```

5 pt. **c.**

d. [4pt] Please give a functor instance for 'HtmlP'. You may assume that we already have a 'Functor' instance for 'Map k'.

4 pt. **d.**

3 In this question, we test your understanding of IO and monads.

a. [5pt] Write a function

```
promptUntil :: (String -> Bool) -> IO [String]
```

that repeatedly asks the user for a line of input (using getLine) until a given predicate is satisfied; it should collect all inputs for which the predicate returned False in a list.

5 pt. **a.**

b. [4pt] Desugar the following code into bind notation

```
sumInt :: IO Int
sumInt = do ss <- promptUntil null
            let n = sum (map read ss)
            print n
            return n
```

4 pt. **b.**

4 In this question, we test your understanding of specification and testing.
a. [3pt] Consider the function

```
chunksOf :: Int -> [a] -> [[a]]
```

that splits a list into chunks of length n > 0, where the last chunk may be shorter than length n.
For example,

```
chunksOf 3 "abcdefgh" == ["abc", "def","gh"]
chunksOf 2 "abc"      == ["ab", "c"]
chunksOf 2 "abcd"     == ["ab", "cd"]
chunksOf 3 ""         == []
```

Complete the following definition of 'chunksOf' that uses 'unfoldr'. Intuitively; 'unfoldr f s' applies the function f on the current "state" s, to generate the next element in the list (if such an element should exist) as well as the new state. If the function f returns Nothing this signals the end of the list.

```
unfoldr     :: (s -> Maybe (a, s)) -> s -> [a]         -- a useful helper
unfoldr f s = case f s of Nothing      -> []
                          Just (a, s') -> a : unfoldr f s'

chunksOf n = unfoldr step where
             step [] = a.  .....(1 pt.)
             step xs = b.  .....(2 pt.)
```

b. [3pt] We would like to build a function 'spec' that could be used to test whether our implementation of 'chunksOf' is correct. In particular, we would like it to detect the error in the following candidate implementation.

```
chunksOf' n xs | length xs < n = []
chunksOf' n xs                 = take n xs : chunksOf' n (drop n xs)
```

Explain in natural language what an implementation of spec should check. (Not just to find the error in 'chunksOf'', but to test full correctness of any implementation.)

3 pt. **c.**

c. [2pt] Please give the type of 'spec', where 'spec' should *not* take a candidate implementation of 'chunksOf' as one of its arguments. (In its implementation, we would instead simply call the global definition of 'chunkOf' that we have written.). Recall that, in order to run the spec using QuickCheck, it should be able to generate inputs and check whether the result matches the expected result.

```
spec :: d.  .....(2 pt.)
```

5 In this question, we test your understanding of lazy evaluation.

a. [6pt] Indicate, for each of the following expressions what their WHNF is. If the expression is already in WHNF, please copy the original expression. If the expression crashes in its evaluation to WHNF, please write "undefined".

```
head (drop 2 [1,2,undefined,4])
```

a. ..... (1 pt.)

```
length (take 0 (undefined : [1..]))
```

b. ..... (1 pt.)

```
foldr (||) True [True, undefined, False]
```

c. ..... (1 pt.)

```
foldl (||) False [False, undefined, True]
```

d. ..... (1 pt.)

```
seq (head [], 5) 99
```

e. ..... (1 pt.)

```
foldr (:) [] [1..10] : map ((:[]) . (+1)) [ j | j <- [9..]]
```

f. ..... (1 pt.)

b. [2pt, bonus] Indicate, for each of the following more challenging expressions what their WHNF is. If the expression is already in WHNF, please copy the original expression. If the expression crashes or diverges in its evaluation to WHNF, please write "undefined".

```
foldr (&&) True lst where lst = True : False : lst
```

g. ..... (1 pt.)

```
foldl (&&) True lst where lst = True : False : lst
```

h. ..... (1 pt.)

6 In this question, we test your knowledge of basic FP and higher order functions. Run length encodings can be used to compress a list. The idea is to encode repeated consecutive elements using a pair of the element and its frequency. The run length encoding of a list is then a list of such element and frequency pairs. For example,

```
"aaabccccaa"
```

has run length encoding

```
[('a', 3), ('b', 1), ('c', 5), ('a', 2)]
```

[4pt] Complete the following implementation of a function that constructs a run length encoding, by using a fold:

```
rle :: Eq a => [a] -> [(a, Int)]
rle = foldr f e where
  e = a. ..... (1 pt.)
  f a [] = b. ..... (1 pt.)
  f a l@((a', i):rest) | a == a' = c. ..... (1 pt.)
                       | otherwise = d. ..... (1 pt.)
```