[20261112] INFOFP - Functioneel programmeren - 1 - USP

Cursus: BETA-INFOFP Functioneel programmeren (INFOFP)

Tijdsduur: 2 uur en 30 minuten

Aantal vragen: 5

[20261112] INFOFP - Functioneel programmeren - 1 - USP

Cursus: Functioneel programmeren (INFOFP)

Aantal vragen: 5

1 In this question, we work with a simple language of arithmetic expressions containing integer literals, variables, addition and subtraction. We will first manipulate these expressions and then translate them into a sequence of instructions.

```
data Op = Add | Sub deriving (Eq, Show)

data Expr a = Lit Int | Var a
            | Bin Op (Expr a) (Expr a)
  deriving (Eq, Show)
```

Lit n is an integer literal
Var x is a variable labelled x
Bin Add l r and **Bin Sub l r** represents l + r and l - r, respectively.

We will now consider a function

```
substitute :: a.  ..............................................(1 pt.)
```

that operates on expressions. **substitute replaceBy e** replaces each variable **Var x** in the original expression **e** by the result of **replaceBy x**, preserving literals and operations. Do not substitute inside inserted expressions. Replacements may use a different type of variable label. For example:

```
substitute (\x -> Bin Add (Var x) (Lit 1)) (Bin Sub (Var "u") (Lit 2)) ==
Bin Sub (Bin Add (Var "u") (Lit 1)) (Lit 2)
```

a. [1pt] Fill in the most general type of **substitute** above.

b. [4pt] Complete the definition:

```
substitute _ (Lit n) = b.  ......................................(1 pt.)
substitute replaceBy (Var x) = c.  ..............................(1 pt.)
substitute replaceBy (Bin op l r) = d.  .........................(2 pt.)
```

c. [2pt] Give a **Functor** instance for **Expr** using **substitute**, without using direct recursion.

2 pt. **e.**

We translate expressions into lists of instructions:

```
data Instr a = Push Int | Load a | Apply Op deriving (Eq, Show)
```

Use exactly these translation rules:

- **Lit n** generates **Push n**.
- **Var x** generates **Load x**.
- **Bin op l r** generates the code for **l**, then the code for **r**, then **Apply op**.

These rules are all you need; no compiler or assembly knowledge is required.

d. [4pt] Complete the definition of **compileOnto** below. Use **(:)** and recursive calls to **compileOnto**; do not use concatenation or call compile.

```
compileOnto :: Expr a -> [Instr a] -> [Instr a]
compileOnto (Lit n) rest = Push n : rest
compileOnto (Var x) rest = Load x : rest
compileOnto (Bin op l r) rest = f. ..............................(4 pt.)

compile :: Expr a -> [Instr a]
compile e = compileOnto e []
```

For example:

```
compileOnto (Bin Sub (Lit 9) (Bin Add (Var "x") (Lit 2))) [Load "z"] ==
[Push 9, Load "x", Push 2, Apply Add, Apply Sub, Load "z"]
```

2 For a list **xs** of length n, **suffixTotals xs** contains the sums of **drop i xs** for positions **i** = 0, 1, ..., n, including the empty suffix.

```
suffixTotals [4, -1, 3] == [6, 2, 3, 0]
suffixTotals [] == [0]
```

a. [3pt] Complete the implementation of **suffixTotals** below; use no direct recursion. You need not add a case to step for an empty accumulator.

```
suffixTotals :: [Int] -> [Int]
suffixTotals = foldr step  a.  ..................................(1 pt.)
  where
    step x totals@(s : _) =  b.  ..................................(2 pt.)
```

b. [5pt] Implement

```
largeSuffixes :: Int -> [Int] -> [Int]
```

to return, in increasing order, the starting positions of suffixes whose total is strictly greater than limit. The empty suffix at position **length xs** is included when it qualifies.

```
largeSuffixes 2 [4, -1, 3] == [0, 2]
largeSuffixes (-1) [] == [0]
```

Use **suffixTotals**, **map**, **filter** and **zip**. You may use lambdas, composition, projections and enumerations. Do not use direct recursion or a list comprehension.

5 pt. **c.**

c. [3pt] Implement

```
specSuffixTotals :: ([Int] -> [Int]) -> [Int] -> Bool
```

so that **specSuffixTotals candidate xs** returns **True** exactly when **candidate xs** correctly computes the suffix totals for every suffix of **xs**. You may assume the **candidate** returns a finite, fully defined list, possibly empty or of the wrong length. Do not call **suffixTotals**. Your predicate must safely reject an empty output.

For example:

```
specSuffixTotals (const [6, 2, 3, 0]) [4, -1, 3] == True
specSuffixTotals (const [6, 2, 4, 0]) [4, -1, 3] == False
```

Hint: You may use

```
tails :: [a] -> [[a]]
tails [] = [[]]
tails xs@(_:ys) = xs : tails ys
```

to compute all suffixes of a list.

```
specSuffixTotals candidate xs =  d.  ..............................(3 pt.)
```

3 We will develop a function **labelNames** that can label names ---words that start with a capital--- in a sentence. The main idea will be to scan through the sentence while maintaining a table with the names that we have seen and their corresponding label (which will just be an **Int**). We can use the following type to represent the table:

```
type NameTable = (Int, [(String, Int)])
```

The first argument of the tuple is the first available label (the smallest unused non-negative integer). The second argument is an association list with the names that we have seen before, together with their labels. When we see a new name, we will assign it the first available label and prepend it to the association list.

When we see a repeated name, all names retain their existing label. Recall that **State s a** describes a computation returning an **a** and maintaining state of type **s**. You may use:

```
get :: State s s -- read state
put :: s -> State s () -- replace state
runState :: State s a -> s -> (a, s) -- (result, final state)
lookup :: Eq k => k -> [(k, v)] -> Maybe v
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
```

mapM performs the actions from left to right and returns their results in that order.

We can thus represent the (intermediate) labeling computation using a **State NameTable a** type (for some appropriate choice of type **a**).

a. [4pt] Complete the implementation of **processName name**, which returns the label assigned to **name**. If the **name** is already in the table, return its existing label and leave the state unchanged. Otherwise, appropriately assign it the next available label and return the assigned label. You can assume the given **name** is a valid name.

```
processName :: String -> State NameTable Int
processName name = do
  (next, table) <- get
  case lookup name table of
    Just old -> a.  ..............................................(1 pt.)
    Nothing -> do
        b.  ......................................................(2 pt.)
        c.  ......................................................(1 pt.)
```

For example:

```
runState (do _ <- processName "Oak"
             _ <- processName "Elm"
             processName "Oak" )
         (0, [])
  == (0, (2, [("Elm", 1), ("Oak", 0)]))
```

b. [3pt] Rewrite the definition of

```
rememberPair :: String -> String -> State NameTable (String, Int)
```

below using **(>>=), (>>)**, lambdas and **return**. You may not use **do** notation. Ordinary **let** ... **in** ... is allowed.

```
rememberPair first second = do i <- processName first
                               let labelled = (first, i)
                               processName second
                               return labelled
```

3 pt. **d.**

c. [3pt] Complete the following implementation of **labelNames** that takes a sentence as a **String**, starts from **(0, [])**, and returns the final association list, with one pair per distinct name. Keep the table order produced by **processName** (reverse order of first appearance).

```
labelNames :: String -> [(String,Int)]
labelNames sentence =
  ( e. ..................................................... (1 pt.) )
    (runState
      ( f. ..................................................... (1 pt.)
        [ w | w <- words sentence, startsWithCapital w ])
      g. ..................................................... (1 pt.) )
```

where

```
startsWithCapital :: String -> Bool
startsWithCapital (c:_) = isUpper c
startsWithCapital _ = error "absurd"
```

d. [3pt] Implement

```
processSentence :: IO Int
```

that reads a one line sentence from standard input, labels the names using the **labelNames** function, prints the names with their labels, and returns the number of distinct names. Print no prompts or additional output. You may use that:

```
getLine :: IO String
print :: Show a => a -> IO ()
```

Input Oak Elm Oak prints **[("Elm",1),("Oak",0)]** and returns **2**; an empty line prints **[]** and returns **0**.

3 pt. **h.**

4 Using the definitions below, prove the following claims. Justify each reasoning step with the label of the definition used, or I.H. for an induction hypothesis. State any induction hypotheses explicitly, and write each reasoning step separately.

```
pre :: (a -> b) -> (b -> c) -> a -> c
pre f h x = h (f x)                          -- a

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

total :: [Int] -> Int
total [] = 0                                 -- c
total (x:xs) = x + total xs                  -- d

withSum :: [Int] -> (Int -> b) -> b
withSum [] k = k 0                           -- e
withSum (x:xs) k = withSum xs (preAdd x k) -- f

preAdd :: Int -> (Int -> b) -> Int -> b
preAdd x k s = k (x + s)                     -- g
```

a. [6pt] For **g :: a -> b** and **f :: b -> c**, prove

```
pre (f . g) = pre g . pre f
```

Both sides have type **(c -> d) -> a -> d**.

Hint: Recall that function application is left-associative, i.e. that the expression **h k x** means **(h k) x**.

6 pt. **a.**

b. [8pt] Prove by induction on xs that, **<u>for every</u>** function **k :: Int -> b**,

```
withSum xs k = k (total xs)
```

8 pt. **b.**

5

a. [1pt] Using the definitions below, choose the first WHNF reached for

```
tail (map pairUp [undefined, 4]).
```

You may use that

```
pairUp :: Int -> (Int, Int)
pairUp x = (x, x + 1)

map f [] = []
map f (x:xs) = f x : map f xs

tail (_:xs) = xs
```

A correct answer gives 1pt, an incorrect answer gives -1pt, "I don't know" gives 0pt.

1 pt. **a.**

a. undefined

b. (4, 5) : []

c. pairUp 4 : map pairUp []

d. map pairUp [4]

e. I don't know

b. [1pt] Fill in the first WHNF reached by

```
makeBox undefined
```

where

```
makeBox :: Int -> Maybe Int
makeBox x = Just (x + 1)
```

First WHNF: **b.** ..... (1 pt.)

c. [4pt] For each expression below, fill in the result, or write undefined if evaluation fails or does not terminate.

```
((+) (undefined :: Int)) `seq` True
```

Result: **c.** ..... (1 pt.)

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

Result: **d.** ..... (1 pt.)

```
let k x rest = if x == 2 then True else rest in
  foldr k False (cycle [1..2])
```

Result: **e.** ..... (1 pt.)

```
let k acc x = acc || x == 2 in foldl k False (cycle [1..2])
```

Result: **f.** ..... (1 pt.)

You may use that **cycle [1..2]** repeats **1, 2, 1, 2, ...** forever and that

```
foldr f z [] = z
foldr f z (x:xs) = f x (foldr f z xs)

foldl f z [] = z
foldl f z (x:xs) = foldl f (f z x) xs
```

d. [1pt] Complete **forcePair** below using **seq**: forcing its result to WHNF must force both components to WHNF and return the original pair.

```
forcePair :: (a, b) -> (a, b)
forcePair (x, y) = g. ..... (1 pt.)
```