CHALMERS
EXAMINATION / TENTAMEN

Course code/kurskod	Course name/kursnamn			
TDA555	Introduction to functional programming			
Anonymous code Anonym kod		Examination date Tentamensdatum	Number of pages Antal blad	Grade Betyg
TDA555-0038-RBS		2023-08-15	10	4

* I confirm that I've no mobile or other similar electronic equipment available during the examination.
Jag intygar att jag inte har mobiltelefon eller annan liknande elektronisk utrustning tillgänglig under eximinationen.

Solved task Behandlade uppgifter No/nr		Points per task Poäng på uppgiften	Observe: Areas with bold contour are to completed by the teacher. Anmärkning: Rutor inom bred kontur ifylles av lärare.
1	X	G	
2	X	G	
3	X	G	
4	X	G	
5	X	G	
6	X	G	
7	X	G	
8	X	G.	
9	X	U	
10			
11			
12			
13			
14			
15			
16			
17			
Bonus poäng			
Total examination points Summa poäng på tentamen		7/1	

CHALMERS | Anonymous code / Anonym kod: TDA555-0038-RBS | Points for question (to be filled in by teacher) / Poäng på uppgiften (ifylles av lärare): 6 | Consecutive page no. / Löpande sid nr: 1 | Question no. / Uppgift nr: 1

a) f [1, 0, 1] = {def f}

= sum (g o [1, 0, 1]) = {def g line 2}

= sum (1 * 2^0 : g (0+1) [0, 1]) = {def g line 2}

= sum (1^1 : 0 * 2^1 : g (1+1) [1]) = {def g line 2}

= sum (1 : 0^2 : 1 * 2^2 : g 3 [1]) = {def g line 1}

= sum (1 : 0 : 4 : [])

= sum [1, 0, 4] = {def sum}

= 5

b) f :: Num a => [a] -> a (general case)

or f :: [Int] -> Int (specific case)

my answer f :: Num a => [a] -> a

8

5

a) groupLetters :: [Char] → [[Char]]

groupLetters
= group . sort . map toLower .
filter (isAlpha && (not . isDigit))

{- 1. Filter all characters that are letters
I am unsure if the isAlpha function
means alpha numeric so I have
a redundant && (not . isDigit) check.

2. Then we lower case all letter
filtered out and sort them.

3 Lastly we group the letters next to eachoth
by equality. According to definition
of group.
-}

b) count :: String → [(Char, Int)]
count cs = [(head ls, length ls) |
ls ← groupLetters cs]

{- After groupLetters cs all letters are
group in non-empty groups of same
letters, we can then map each group to
the tuple of (letter, count of letter)
by using head, and length on the groups.

5

a)
```
data Truth
  = Val Bool
  | And Truth Truth
  | Or Truth Truth
  | Negate Truth
  | Implies Truth Truth
  | Var Char
```

```
type (∧) = And
type (∨) = Or
type (=>) = Implies
type (¬) = Negate
```

b) `exp = (Var 'a') ∧ ((Var 'b') => Val False)`

-1 I could have had separate values for True and False as elements of type Truth but I decided to have the Val constructor instead to avoid name clashes with Haskell True and False. but this does mean that you can have normal Haskell boolean operators to build a Bool for the Val constructor. -3

5

```
readPerson :: IO Person
readPerson = do
    name <- readUser (not . null) id "Name: "
    age  <- readUser (all isDigit) read "Age: "
    yn   <- readUser ('elem' ["y","n"]) convert
                "Would you like marketing material (y/n): "
    return $ Person name age yn
  where
    readUser :: (String -> Bool) -> (String -> a)
                -> String -> IO a
    readUser p f q = do
      putStr q
      a <- getLine
      if p a then return (f a) else do
          putStrLn "Invalid Input"
          readUser p f q
    convert :: String -> Bool
    convert "y" = True
    convert "n" = False
```

8

{- The p parameter is a predicate on the input string.
The f parameter is the conversion function from string to type a.
The q parameter is the question to ask the user.

CHALMERS | Anonymous code / Anonym kod: TDA555-0030-RBS | Points for question (to be filled in by teacher) / Poäng på uppgiften (ifylles av lärare): 6 | Consecutive page no. / Löpande sid nr: 5 | Question no. / Uppgift nr: 5

```
data Set a = Set [a]

empty :: Set a
empty = Set []

mkSet :: Eq a => [a] -> Set a
mkSet = Set . nub

elemSet :: Eq a => a -> Set a -> Bool
elemSet x (Set xs) = x `elem` xs

union :: Eq a => Set a -> Set a -> Set a
union (Set xs) (Set ys) = Set $ nub $ xs ++ ys

intersection :: Eq a => Set a -> Set a -> Set a
intersection (Set xs) (Set ys) = Set
                                 [z | z <- xs, z `elem` ys]
```

CHALMERS | Anonymous code / Anonym kod: TDA555-0037-RBS | Points for question (to be filled in by teacher) / Poäng på uppgiften (ifylles av lärare): 6 | Consecutive page no. / Löpande sid nr: 6 | Question no. / Uppgift nr: 6

a)
```
prop_notLarger :: String -> Bool
prop_notLarger cs = length cs >= Sum (map snd $ count cs)
```

b)
```
prop_contains :: String -> Bool
prop_contains cs = and
  [c `elem` cs | (c,_) <- count cs]
```

a)
```
mapFolder :: (File -> File) -> Folder -> Folder
mapFolder f (Folder name files subdirs)
    = Folder name (map f files)
                  (map (mapFolder f) subdirs)
```

b)
```
makeLower :: Folder -> Folder
makeLower = mapFolder toLower
```

```
data State = St
  { cursor  :: Int
  , content :: String
  , clip    :: String   -- clipboard
  } deriving Show

startState :: State
startState = St { cursor = 0, content = "", clip = "" }

showState :: State -> String
showState = content


runKey :: State -> Key -> State
runKey (St cur content clip) key =
  case key of
    Chr c  -> St (cur+1) (before ++ c : after) clip
    Del    -> St (check (cur-1)) (take (length before - 1)
                                  before ++ after) clip
    GoLeft -> St (check (cur-1)) content clip
    GoRight-> St (check (cur+1)) content clip
    Copy   -> St (cur - length word) content word
    Paste  -> St cur (content ++ clip) clip
  where
```

Answer only one question on this page. Do not write on the back of this paper
Behandla endast en uppgift på detta blad. Skriv ej på baksidan

5

```
where
    (before, after) = splitAt cur content
    word = ~~if ' ' `notElem` before then~~
           ~~before~~ else
           reverse . takeWhile (not . isSpace)
             $ reverse before

rest = if ' ' `notElem` before then ""
       else reverse . dropWhile (not isSpace)
              $ reverse before

check :: Int -> Int
check n | n < 0 = 0
        | n > length content = length content
        | otherwise = n

-- check that new value of cursor is
   within bounds.

run :: [Key] -> String
run = showState . foldl runKey startState

-- The state can probably be defined differently
and our computations made easier.
```

a)

```
foldFolder :: ((Name, [File]) -> b -> b) -> b -> Folder
                                              -> b

foldFolder f z (Folder n fs [])
    = f (n, fs) z

foldFolder f z (Folder name files subdirs)
    = f (name, files) $
        foldr g z subdirs
  where
    g :: Folder -> b -> b
    g folder z = foldFolder f z folder
```

this will not work, need to 'thread' z.

b)

```
allFiles :: Folder -> [File]
allFiles = foldFolder (\(_, fs) z -> fs ++ z)
           []
```